

Mikrokontrolery STM32

Układy peryferyjne i nie tylko

(poradnik dla początkujących)

autor: *szczywronek* (szczywronek@gmail.com)

wersja: 1.9

ostatnia modyfikacja: 26.01.2016

Ziemia Obiecana, 31.07.2015

I n w e n t a r z

1. Wstęp formalny	7
1.1. Co to właściwie jest? („Quo vadis Domine?”)	7
1.2. Ja („Quicquid Latine dictum sit, altum videtur”)	8
1.3. Ty („Bene tibi!”)	8
1.4. Uwagi końcowe („Cogitationis poenam nemo patitur”)	10
2. Wstęp techniczny	12
2.1. Od przybytku głowa (nie) boli („Divitiae non sunt bonum”)	12
2.2. O co chodzi z tym rdzeniem? („Nil mirari, nil indignari, sed intelligere”)	14
2.3. Hardware („Nemo sine vitiis est!”)	17
2.4. Piśmiennictwo („Littera docet, littera nocet.”)	19
2.5. Software („Ex malis eligere minima oportet”)	24
2.6. Słowo o bibliotekach („Relata refero”)	25
2.7. Wskazówki przed startowe („Ave, Caesar, morituri te salutant”)	30
3. Porty I/O („Ardua prima via est”)	36
3.1. Ogarnąć nóżki w F103 (F103)	36
3.2. Nieśmiertelnie Blinkający Hell of World (F103)	40
3.3. Atomowo macham nogą i nie tylko	44
3.4. Cortex-M4 wybieram Cię! (F334 i F429)	48
3.5. Wybór funkcji alternatywnej (F334 i F429)	52
3.6. Remapping funkcji alternatywnych (F103)	57
3.7. Elektryczna strona medalu	58
3.8. Skompensuj mi celę (F429)	60
4. Bit banding („Gemma gemmarum”)	61
4.1. Ale o co chodzi?	61
4.2. Jak to działa?	62
4.3. Jasne i proste jak cała matematyka	64
4.4. Makro do bit bandingu	66
4.5. Kiedy stosować bit banding	69
4.6. A gdzie jest haczyk?	69
5. Wyjątki („Macte animo, iuvenis!”)	72
5.1. Trochę zbędnej teorii o trybach pracy rdzenia	72
5.2. Poznajmy wyjątki w Cortexie	74
5.3. Mechanika działania wyjątków	77
5.4. Priorytety i wywłaszczanie	79

5.5. Funkcje pomocnicze	82
5.6. Priorytety przykład praktyczny	84
6. Licznik systemowy SysTick („Magnum opus”)	88
6.1. Blink me baby one more time	88
7. Blok EXTI i przerwania zewnętrzne („Facta sunt verbis difficilora”)	93
7.1. EXTI (F103)	93
7.2. EXTI (F429)	99
7.3. EXTI (F334)	101
8. Liczniki („Festina lente!”)	105
8.1. Wstęp	105
8.2. Blok zliczający, prosty timer	106
8.3. Update Event (UEV), buforowanie rejestrów i One Pulse Mode	108
8.4. Schemat blokowy licznika	109
8.5. Licznik pędzony z zewnątrz	111
8.6. Filtrowanie sygnałów zewnętrznych	115
8.7. Tryb enkodera	117
8.8. Taktowanie licznika innym licznikiem	119
8.9. Podsumowanie źródeł taktowania	120
8.10. Blok porównujący - PWM	121
8.11. Blok porównujący - przerwania	128
8.12. Blok porównujący - podsumowanie	130
8.13. Blok przechwytyjący - Input Capture	131
8.14. Synchronizacja sygnałem zewnętrznym	133
8.15. PWM Input Mode	136
8.16. Synchronizacja kilku liczników	139
8.17. Liczniki ogólnego przeznaczenia i podstawowe	141
8.18. Luźne uwagi na koniec	142
8.19. Różnice między F103 i F429 (F103 i F429)	144
8.20. Licznikowe indywidualum (F334)	147
9. Battery Backup Domain (“Non omnis moriar”)	159
9.1. Wstęp	159
9.2. Backup Registers (F103)	160
9.3. RTC (F103)	163
9.4. Backup Registers (F429)	166
9.5. Backup SRAM (F429)	169

9.6. RTC (F429)	171
9.7. Backup Registers i RTC (F334)	175
10. Układy watchdog („Duo cum faciunt idem, non est idem”)	176
10.1. Watchdog niezależny IWDG	176
10.2. Watchdog okienkowy WWDG	180
10.3. Porównanie układów watchdog	185
10.4. Układy watchdog w mikrokontrolerze F334 (F334)	186
11. Reset, zasilanie i tryby oszczędzania energii („Difficilis in otio quies”)	188
11.1. Reset	188
11.2. Zasilanie (F103)	190
11.3. Zasilanie (F429)	191
11.4. Zasilanie (F334)	191
11.5. Debugowanie a tryby uśpienia	191
11.6. Tryby obniżonego poboru mocy (F103)	193
11.7. Tryby obniżonego poboru mocy (F429)	198
11.8. Tryby obniżonego poboru mocy (F334)	206
12. Mechanizm DMA („Annuntio vobis gaudium magnum: habemus DMA”)	207
12.1. Z czym to się je?	207
12.2. DMA (F103)	208
12.3. DMA (F429)	215
12.4. DMA (F334)	224
13. Przetwornik ADC („Superflua non nocent”)	227
13.1. ADC wstęp (F103)	227
13.2. Tryby pracy pojedynczego przetwornika ADC (F103)	230
13.3. Czas próbkowania (F103)	242
13.4. Tryby pracy dwóch przetworników ADC (F103)	245
13.5. Bajery (F103)	251
13.6. Ogólne spojrzenie na tryby pracy przetwornika ADC (F103 i F429)	256
13.7. Różnice w STM32F429 (F429)	257
13.8. Różnice w STM32F334 (F334)	272
13.9. Końcowe uwagi	291
14. Przetwornik DAC („Omne ignotum pro magnifico”)	293
14.1. Wstęp (parametry i tryby pracy)	293
14.2. Zadania praktyczne (F103)	298
14.3. Zadania praktyczne (F429)	307

14.4. Odrobina odmiany (F334)	308
14.5. Uwagi końcowe	308
15. Interfejs USART („Volenti nihil difficile”)	309
15.1. USART (F103)	309
15.2. USART (F429)	314
15.3. USART (F334)	316
16. Pamięć Flash i bootowanie („Variatio delectat”)	323
16.1. Pamięć Flash	323
16.2. Tryby uruchamiania i bootloader	325
16.3. Bajty konfiguracyjne (F103)	328
16.4. Bajty konfiguracyjne (F429)	332
16.5. Bajty konfiguracyjne (F334)	333
16.6. Kod w pamięci SRAM - po co?	333
16.7. Funkcja w pamięci sram - jak?	338
16.8. Cały program w pamięci sram - jak?	344
17. System zegarowy („Finita est comoedia”)	353
17.1. Wstęp	353
17.2. System zegarowy (F103)	355
17.3. System zegarowy (F429)	363
17.4. System zegarowy (F334)	368
18. Hashing randomly encrypted CRC (“Contra facta non valent argumenta”)	371
18.1. Wstęp z niespodzianką	371
18.2. Nic nie dzieje się przypadkiem (F429)	372
18.3. Suma kontrolna CRC32 (F103 i F429)	380
18.4. Sprzętowe CRC8, CRC16 i dowolny wielomian (F103 i F429)	389
18.5. Suma kontrolna CRC (F334)	389
19. Interfejs SPI (“Potius sero quam numquam”)	395
19.1. Wstęp (F103 i F429)	395
19.2. Master ŚPI (F103 i F429)	398
19.3. Master i Slave w jednym SP(al)I domu (F103 i F429)	405
19.4. Pół-puplex na dwa mikrokontrolery (F103 i F429)	410
19.5. CRC w SPI (F103 i F429)	420
19.6. Outsider (F334)	421
20. Komparator i wzmacniacz („Nec temere, nec timide”)	426
20.1. Komparator analogowy (F334)	426

20.2. Wzmacniacz operacyjny (F334)	436
Dodatek 1: Funkcja konfigurująca porty (F103)	450
Dodatek 2: Funkcja konfigurująca porty (F429)	453
Dodatek 3: Makro do bit bandingu	459
Dodatek 4: To bit band or not to bit band	463
Dodatek 5: Atrybut interrupt (F103, GCC)	466
Dodatek 6: Przerwanie widmo	468
Dodatek 7: Gwałt na Nucleo w dwóch aktach	470
21. Errata i changelog („Hominis est errare, insipientis in errore perseverare”)	474
21.1. Błędy zauważone w wersji 1.0 Poradnika	474
21.2. Błędy zauważone w wersji 1.1 Poradnika	480
21.3. Błędy zauważone w wersji 1.2 Poradnika	483
21.4. Błędy zauważone w wersji 1.3 Poradnika	485
21.5. Zmiany dokonane w wersji 1.4 Poradnika	486
21.6. Zmiany dokonane w wersji 1.5 Poradnika	487
21.7. Zmiany dokonane w wersji 1.6 Poradnika	489
21.8. Zmiany dokonane w wersji 1.7 Poradnika	495
21.9. Zmiany dokonane w wersji 1.8 Poradnika	504

1. WSTĘP FORMALNY

1.1. Co to właściwie jest? („*Quo vadis Domine?*”¹)

Poznając mikrokontrolery STM32 postanowiłem, w miarę na bieżąco, spisywać to co uznałem za warte utrwalenia lub odkrywcze. Pierwotnie miały to być notatki tylko dla mnie. Skoro jednak to „coś” powstało to może warto się podzielić? A nuż komuś pomoże (albo zaszkodzi i będzie mniejsza konkurencja na rynku pracy :]). Postanowiłem więc zebrać wszystko do kupy i na tej bazie stworzyć niniejszy Poradnik. Tak oto powstało toto. Wiele z opisanych tu rzeczy wydaje się teraz oczywiste. Nie były jednak takie, gdy zaczynałem przygodę z STM32 (i ogólnie mikrokontrolerami). Proszę więc wybaczyć, czasem, infantylne porównania i opisy - może komuś pomogą zrozumieć jakieś zagadnienie tak jak i mnie pomagały :) Swoją drogą mam cichą nadzieję, że po opublikowaniu tego Poradnika, zgodnie z regułą Cunninghama², i ja się sporo nauczę :)

Nie zamierzam, z założenia, tłumaczyć dokumentacji STMa, ani dopisywać do niej przykładów wykorzystujących każdy możliwy tryb każdego z peryferiów. Przynajmniej na początku chciałbym pokazać jak sobie samemu radzić w dokumentacji, pomóc w przełamaniu bariery językowej która czasem utrudnia załapanie całkiem prostych rzeczy... które ktoś idiotycznie nazwał i zamotał opis. Potem, w miarę oswajania z mikrokontrolerem i dokumentacją, wszystko staje się proste i sprowadza do ustwienia paru bitów zgodnie z dokumentacją. I tak na dobrą sprawę nie ma co opisywać i tłumaczyć, więc i formuła Poradnika będzie się stopniowa zmieniała. Mniej gadaniny, więcej przykładów, kilka zdań o możliwych trybach, pułapkach i tyle. Tzn. taki jest plan. Czy się uda – to już nie mnie oceniać.

Poradnik poświęcony jest **podstawom**. Zdecydowanie nie będzie tu przykładów obsługi, dajmy na to, kart pamięci SD. Dlaczego? Obsługa interfejsu do komunikacji z kartą (np. interfejsu SDIO) jest prosta i każdy kto przebrnie przez ten Poradnik da sobie radę sam. Tyle że poza obsługą SDIO trzeba jeszcze okieźnać kontroler karty, bez tego nic się nie działa. A kontroler karty nie jest przedmiotem tego poradnika! Podobnie ma się sprawa np. z USB, wyświetlaczami graficznymi czy innymi RTOSami. Nie podoba się? To trudno...

Luźna dygresja: swoją drogą karty SD i interfejs USB łączy to, że można przeczytać całą oficjalną dokumentację dotyczącą tych tworów i dalej nie mieć zielonego pojęcia jak to właściwie ugryźć w praktyce :)

1 „Dokąd idziesz Panie?”

2 “The best way to get the right answer on the Internet is not to ask a question, it's to post the wrong answer.”

1.2. Ja („Quicquid Latine dictum sit, altum videtur”³)

Hobbystą jestem i nic tego nie zmieni! Niedzielnym kierowcą świata mikrokontrolerów. Nie mam wykształcenia elektronicznego, programistycznego, mechatronicznego i tym podobnego. W tym dokumencie opisuję wszystko tak, jak udało mi się to zrozumieć i językiem takim, jakim ja rozumiem. Języka staram się nie kaleczyć, czyli nie popełniać błędów wynikających z takiego czy innego roztrzepania. Neologizmów oraz wszelkiej maści naciągnięć językowych nadużywam z lubością i pełną premedytacją. Dodatkowo, po przeczytaniu naraz kilkuset stron anglojęzycznych datasheetów, mózg ulega osobliwemu zakwaszeniu co owocuje pojawianiem się w tekście określeń typu *trygierz*, *trygier*⁴ czy *zakapturzenie*⁵. Jak się nie podoba to wiadomo... ja nikogo nie zmuszam.

Początki na pewno nie są proste - tyle nowości... Dotychczas bawiłem się tylko mikrokontrolerami ATmega oraz ATTiny i moim największym osiągnięciem był zegar na matrycy led (ale na swój sposób wyjątkowy bo bez multipleksowania!). Tak jakoś się zadziało, że chcąc nie chcąc wkręciłem się w pewien projekt i zostałem zmuszony gwałtem do użycia mikrokontrolera STM32. Od około 2 tygodni⁶ „googluję” w Internecie poszukując informacji, poradników, książek itd. itp. o mikrokontrolerach STM32 i wszystkim co się z nimi wiąże. I mam coraz większy mętlik w głowie. Przerażają mnie możliwości, ilość rzeczy jakie należy konfigurować, biblioteki, środowiska programistyczne i wizja spalenia płytka za ponad 100zł⁷ (po spaleniu Atmegi cierpiąła głównie duma, nie kieszeń). Wierzę jednak, że uda mi się przebrnąć przez ten galimatias.

1.3. Ty („Bene tibi!”⁸)

W kwestii Czytelnika zakładam przede wszystkim dwie rzeczy: że godzi się na to, że dokument ten może zawierać błędy, niedopowiedzenia, pokrętne opisy omijające sedno problemu wynikające z niewiedzy i lenistwa autora, oraz że Czytelnik umie programować w języku C i ogarnia jakieś mikrokontrolery (najlepiej AVR⁹, bo będę je czasem traktował jako coś w rodzaju punktu odniesienia). Może, tak jak ja, przechodzisz złotą drogę AVR → STM32?

Jak to jest z tym ogarnianiem języka i mikrokontrolera? Wiedza niezbędna (krótki test):

- Kiedy i dlaczego stosować słówko *volatile*?

3 „Cokolwiek powiesz po lacinie, brzmi mądrze.”

4 dziwadło językowe powstałe z wduszenia angielskiego słowa *trigger* (wyzwalacz, spust) w realia języka polskiego od ang. *capture* (przechwycić, zdobyć)

6 odniesienie czasowe było aktualne kiedyś tam - gdzieś w kwietniu 2013... Wtedy zaczynałem „od zera” z STM :)

7 oczywiście nie trzeba kupować drogich płytEK. Za 15zł można mieć płytKE z mikrokontrolerem STM32F103 i darmową dostawą do domu (ebay)

8 „Dobrze Tobie! (pozdrowienie)”

9 wszelkie nawiązania do mikrokontrolerów Atmel AVR odnoszą się do rodzin ATTiny i ATmega (chyba, że wyraźnie napisano inaczej)

- Co to jest rejestr mikrokontrolera, co to jest przerwanie?
- Rozumiesz różnicę między komplikacją, linkowaniem, flashowaniem?
- Zdajesz sobie sprawę z tego, że Eclipse niczego nie komplikuje?¹⁰
- Wiesz czym jest stos i przynajmniej słyszałeś o stercie?
- Nie gubisz się jeśli w programie pojawiają się wskaźniki i struktury?
- Czujesz prawo Ohma i coś w życiu elektronicznego spłodziłeś?¹¹
- Potrafisz przeanalizować listing asemblera mając do dyspozycji opis rozkazów i wiedzę „co ten kod powinien robić”?
- Nie gubisz się w operacjach bitowych i logicznych?
- Potrafisz zapisać w C operację¹²: *PORTD = _BV(5);* nie stosując żadnych liter (inaczej mówiąc: czy wiesz czym jest PORTD? i nie chodzi mi o to, że to jest rejestr wyjściowy portu D, tylko czym jest zapis PORTD dla kompilatora i jego pomocników)?

Jeśli pytania nie wprowadzają Cię w zbytnie zakłopotanie to możesz czytać dalej. Jakby co, to proszę do-studiować we własnym zakresie – STFW. Trudno jest poznawać nowe mikrokontrolery, kiedy nie ogarnia się programowania ogólnie :) Uprzedzam: wszelkie przykładowe kody w Poradniku, będą napisane tak aby:

- mnie się szybko pisało
- ukazywały sedno zagadnienia
- Tobie się łatwo analizowało

Stąd czasem pojawią się jakieś prymitywne konstrukcje o dyskusyjnej elegancji. No i w kodach raczej nie będzie komentarzy. Zamiast tego będą omówione w tekście. Przypominam, że to nie jest poradnik C czy dobrego stylu programowania :) Sam bym taki z chęcią przeczytał...

Na koniec jeszcze potruję trochę. Najważniejsza jest praca samodzielna. Możesz wierzyć lub nie, ale czasem dosyć niewinne zdanie z tego Poradnika, kosztowało mnie kilka dni szukania w dokumentacji i Internecie. Serio. Nie przesadzam. Jak bum cyk cyk! Kilka rzeczy wyjaśniło się dopiero po napisaniu testowego programu i obserwacji wyników. Ten Poradnik zapewne da się przeczytać w jeden dzień przy odrobinie zaparcia - kilka razy sam to zrobiłem przed publikacją (a czytanie własnego tekstu jest strasznie nudne bo niczym nie zaskakuje...). Ale gwarantuję, że nic

¹⁰ pytanie dotyczy tylko osób korzystających z tego IDE :)

¹¹ i nie wybuchło... od razu

¹² przykład AVRowy, jeśli ktoś nie bawił się AVRami to niech improwizuje

dzięki temu nie zyskasz a tylko nabawisz się bólu głowy. Poradnik stanowi tylko wstęp ułatwiający start. Dalej musisz radzić sobie sam z wszystkimi wątpliwościami i pytaniami jakie się pojawią :)

W każdym rozdziale opisującym jakieś peryferium, zamieściłem „zadania domowe” z przykładowymi rozwiązaniami. Przyłóż się do tych zadań. Wiem, że na początku to się wydaje jakiś kosmos, ale próbuj za każdym razem! I nie na „odwal się”, tylko solidnie. Spędzenie, na początku, godziny czy dwóch nad migającym LEDem to jest nic! Nie działa? Trudno. Poczekaj, prześlij się z problemem, poszukaj jeszcze raz w dokumentacji, w Internecie. Umówmy się, że jeśli jakieś „zadanie domowe” Ci nie wychodzi, to przykładowe rozwiązanie przeczytasz po przynajmniej trzech dniach prób samodzielnego rozwiązania - ok? :) Nawet jeśli Twój kod nadal nie działa, to cały czas zdobywasz bezcenną umiejętność szukania informacji i rozwiązywania napotkanych problemów. Tylko proszę bez marudzenia że o STIMach jest mało informacji i ciężko znaleźć... O AVRach napisano już wszystko co się da, a na forach i tak co tydzień pojawia się pytanie o obsługę przycisku i miganie diodą. Praktycznie wszystko, co napisałem w tym Poradniku, można znaleźć w dokumentacji mikrokontrolera - zapamiętaj to!

1.4. Uwagi końcowe („*Cogitationis poenam nemo patitur*”¹³)

Poradnik udostępniam nieodpłatnie każdemu zainteresowanemu, do osobistego użytku edukacyjnego *only*. Zezwalam na wszelkiej maści nieodpłatne rozpowszechnianie fragmentów tudzież publiczne odwoływanie się do nich pod warunkiem zachowania informacji o Autorze i pochodzeniu tych fragmentów. Za nic nie biorę odpowiedzialności itd. itd. Poradnik czytasz na własną odpowiedzialność.

Mile widziany *feedback* w wątku **[STM32][C] - Poradnik dla początkujących (bez bibliotek)** na forum Elektroda.pl (<http://www.elektroda.pl/rtvforum/viewtopic.php?p=15126335>) lub na: szczywronek@gmail.com. W szczególności jeśli ktoś znajdzie błędy merytoryczne, językowe, jeśli coś jest napisane pokrętnie i nie wiadomo o co chodzi, brakuje czegoś ważnego, ktoś ma fajny przykład, opis do dodania albo wszystko jest wspaniale i po prostu chcesz mi postawić piwo albo zaoferować pracę¹⁴... Byle konstruktywnie. Uwagi typu „*brakuje opisu interfejsu FSMC*” można sobie darować, gdyż doskonale wiem o czym nie napisałem :) Proszę też nie traktować mojego *maila* jako *help-line*, gdy dioda nie migła.

Jeśli wszystko pójdzie dobrze, to najnowszą wersję Poradnika będzie można zawsze pobrać z forum Elektroda.pl z wspomnianego powyżej wątku. Za kopie umieszczone w innych miejscach nie odpowiadam i jestem im trochę nieprzychylny, gdyż nie nabijają mi punktów na forum (tak

13 „*Nikt nie ponosi odpowiedzialności za swoje myśli.*”

14 gdybym miał pracę to nie miałbym czasu na pisanie takich elaboratów :)

jestem egoistą i zbieram na pendrive'a). I tyle w temacie. W razie jakichś wątpliwości proszę śmiało pisać na podany adres mailowy.

2. WSTĘP TECHNICZNY

2.1. Od przybytku głowa (nie) boli („*Divitiae non sunt bonum*”¹⁵)

Mikrokontroler 32bitowy, *STM32*, *High-Density*, *Cortex*, *ARM*, *Discovery*... ja się na początku pogubiłem. Trzeba te pojęcia jakoś uporządkować.

Popularne *atmegi* i *attiny* należą do rodziny ośmioróżkowych mikrokontrolerów *AVR* produkowanych przez firmę *Atmel*. Rodzina dzieli się na kilka podrodzin (*attiny*, *atmega*, *atxmega*...). W każdej z podrodzin jest kilkanaście(-siąt) różnych układów różniących się peryferiami, obudowami etc. Jasne i proste, prawda?

Off topic! Zatrzymajmy się na chwilę przy tych ośmiu bitach. Zaraz wejdziemy w świat 32 bitowy. Ale co to właściwie znaczy, że mikrokontroler jest 8/32 bitowy? Kilka razy w Internecie spotkałem opinię, że jakiś AVR jest 8 bitowy bo ma 8 nóżek w porcie. Niestety nie jest to takie proste. STM32 jest 32 bitowy a nóżek w porcie ma 16. O „bitowości” mikrokontrolera decyduje to, na jakich danych operuje jednostka arytmetyczno-logiczna rdzenia¹⁶ (ALU). W ośmioróżkach, podstawowym typem danych jest typ zajmujący 8b (np. *uint8_t*). Wynika to z tego, że znakomita większość rozkazów arytmetycznych i logicznych operuje na danych 8b. Działanie na dłuższych danych jest rozbijane na kilka/kilkanaście działań 8b. Analogicznie w przypadku mikrokontrolerów 32 bitowych, podstawowym typem danych jest typ 32 bitowy. ALU w STMacie operuje na danych 32 bitowych. Koniec OT, wracamy do głównego wątku.

STM32 to rodzina 32 bitowych mikrokontrolerów produkowanych przez firmę *STMicroelectronics*¹⁷. Dotąd proste. Konkretnych układów wśród mikrokontrolerów STM32 jest całkiem sporo. Wejdź na: www.st.com → *Products* → *Microcontrollers* → *STM32 32-bit ARM Cortex MCUs (po lewej stronie strony)* i pogap się na wykres.

Podstawowy podział mikrokontrolerów STM32 jest związany z rdzeniem na jakim oparty jest dany mikrokontroler (oś odciętych na wykresie ze strony ST). W przypadku AVRów, rdzeniem jako takim, nikt się specjalnie nie przejmował. Tutaj jest troszkę inaczej o czym za chwilę powiem więcej. W tym momencie ważne jest że w STMacie wykorzystywane są rdzenie ARM¹⁸ Cortex M0,

15 „Bogactwo nie jest dobrem.”

16 jeśli ktoś nie miał do czynienia z asemblerem, to ma pełno prawa nic nie zrozumieć :)

17 www.st.com

18 www.arm.com

M0+, M3, M4 i M7. Rodzaj rdzenia zakodowany jest w oznaczeniu mikrokontrolera. Oznaczenie mikrokontrolera wygląda następująco:

STM32abcd...

gdzie:

- *STM32* to oznaczenie rodziny (analogia do prefiksu *AT* w *ATmega*, *ATtiny*...)
- a – oznaczenie typu :
 - *F* – podstawowy
 - *L* - o niskim poborze mocy)
- b – seria (rdzeń):
 - 0 – *Cortex-M0/M0+*
 - 1,2 – *Cortex M3*
 - 3,4 – *Cortex M4*
 - 7 - *Cortex M7*
- c, d – „linia mikrokontrolera” (im wyższy numer tym więcej bajerów na pokładzie)

Przykładowo: *STM32F103* → rdzeń *CM3* (*Cortex-M3*), typ podstawowy.

Uwaga! Kilka lat temu system oznaczania mikrokontrolerów uległ zmianie, więc oznaczenie starszych kostek może nie pasować do nowego schematu... bywa. Po wpisaniu w googlach „*stm32 oznaczenia*” - bez problemu można znaleźć dokładne informacje o oznaczeniach, jakby ktoś był zainteresowany tym nudnym tematem.

Po oznaczeniu zgodnym z powyższym opisem, pojawiają się jeszcze literki i cyferki związane z typem obudowy, wielkością pamięci – STFW. Jako osoba początkująca masz zapewne jedną czy dwie płytki rozwojowe z STMami - rozkoduj sobie oznaczenie posiadanych scalaczków i o reszcie zapomnij na razie ;)

Dodatkowo można się nadziać na takie nieszczęsne nazwy „linii” mikrokontrolerów *STM32F1xx*¹⁹:

¹⁹ żeby nie było, że nie mówiłem

- *Low density Value line devices* – chodzi o mikrokontrolery *STM32F100* z pamięcią flash z przedziału 16-32kB
- *Low density devices* - *STM32F101, STM32F102, STM32F103* z flashem 16-32kB
- *Medium density Value line devices* – *STM32F100* z flashem 64-128kB
- *Medium density devices* - *STM32F101, STM32F102, STM32F103* z flashem 64-128kB
- *High density Value line devices* – *STM32F100* z flashem 256-512kB
- *High density devices* – *STM32F101, STM32F103* z flashem 256-512kB
- *XL-density devices* – *STM32F101, STM32F103* z flashem 512-1024kB
- *Connectivity line devices* – *STM32F105, STM32F107*

Na koniec dwa ostatnie pojęcia „podstawowe”: *Discovery i Nucleo*. Są to nazwy serii oficjalnych płyt startowych, produkowanych przez ST. Płytki są stosunkowo tanie i dodatkowo każda ma na pokładzie programator i debugger (STLink). Za pomocą STLinków z płyt *Discovery/Nucleo* można programować/debugować inne mikrokontrolery, nie tylko ten w zestawie startowym. Poza tymi „oficjalnymi” zestawami rynek ocieka, rzecz jasna, całą masą mniej lub bardziej chińskich *def-bordów* z *STMami*.

Co warto zapamiętać z tego rozdziału?

- STM32 to 32 bitowe mikrokontrolery produkowane przez firmę STMicroelectronics
- mikrokontrolery STM32 mają rdzeń ARM²⁰ Cortex-Mx
- płytki *Discovery i Nucleo* mają wbudowane wszystko co potrzeba aby tanio, chyżo i skowyrnie rozpocząć zabawę z *STMami*

2.2. O co chodzi z tym rdzeniem? („Nil mirari, nil indignari, sed intelligere”²¹)

W przypadku AVRów sprawa była prosta bo Atmel produkował „cały” mikrokontroler. W przypadku STMów jest nieco inaczej. ST produkuje mikrokontroler, ale wkłada do niego „gotowy” rdzeń kupiony od ARM. Tzn. nie kupuje fizycznie kawałka krzemu, tylko licencję na użycie takiego Cortexa w swojej kostce. ST dokłada do tego rdzenia peryferia (pamięci, timery, liczniki, interfejsy, ADC i takie tam pierdółki) i pakuje to w ładną obudowę. Co z tego wynika w praktyce? Przede wszystkim pozorne zamieszanie w dokumentacji (tylko na początku po przesiadce z AVR) o którym

²⁰ ARM to jednocześnie nazwa architektury rdzenia i firmy która ją opracowała; holding ARM sprzedaje licencje pozwalające na implementację rdzeni ARM przez producentów układów (np. przez ST w kontrolerze STM32)

²¹ „Nie dziwić się, nie oburzać, lecz zrozumieć.”

powiem w rozdziale 2.4. Po drugie: nie ważne jaki mikrokontroler z Cortexem weźmiemy (*STM32*, jakieś *LPC*, czy coś od *Texasa*) – w każdym z nich „rdzeń” będzie działał tak samo.

A co toto ten rdzeń? I co za różnica jaki rdzeń? Dla użytkownika różnica jest tym większa im bardziej niskopoziomowo lubi (lub musi) programować. Od rdzenia zależy np. lista rozkazów *asm* dostępnych w danym mikrokontrolerze czy sposób obsługi wyjątków (przerwań). Na razie nie będziemy wdawać się w szczegóły. Może później jak starczy zapału. Warto natomiast zapamiętać, że im wyższy numer rdzenia (ten po „M”) tym większe możliwości i większa wydajność:

- *M3* to najstarszy i taki „podstawowy” *Cortex*
- *M4* to *M3* plus dodatkowo wsparcie dla przetwarzania sygnałów (specjalne rozkazy umożliwiające wykonanie kilku operacji arytmetycznych naraz) oraz opcjonalnie *FPU* (koprocessor matematyczny wspomagający procek w obliczeniach zmiennoprzecinkowych)
- *M0* to taki budżetowy *Cortex* pozbawiony bajerów²², *low-power*, *low-cost*, wieje nudą...
- *M0+* to taki trochę poprawiony *M0* – generalnie aplikacje *low-power*
- *M7* to najmocniejszy z *Cortexów* mikrokontrolerowych, nie chce mi się szukać szczegółów (dla zainteresowanych: https://en.wikipedia.org/wiki/ARM_Cortex-M)

Rdzeń *Cortex* poza samym mózgiem (*ALU*, *CPU* czy jak to się tam nazywa) składa się z kilku bloków/układów peryferyjnych, z których najważniejsze to:

- licznik systemowy – *SysTick*
- kontroler przerwań – *NVIC*
- *System Control Block*²³ - *SCB*
- koprocessor arytmetyczny - *FPU*
- układy związane z debugowaniem

Przypominam, że w każdym mikrokontrolerze z rdzeniem *Cortex*, te układy będą działały identycznie!

Na razie poprzestańmy na tym. Nie chciałbym zanudzać rzecząmi, które przy pierwszych miganiach diodą do niczego się nie przydadzą a działają zniechęcająco. Tylko sygnalizuję zagadnienie. Dla rzadnych wiedzy – STFW, RTFM. W googlach jest sporo ładnych obrazków porównujących np. zestaw instrukcji każdego z *Cortexów*. Na tym etapie jednak nie jest to

²² np. nie obsługuje dzielenia sprzętowego – lipa jakąś

²³ nie mam pomysłu na tłumaczenie które nie brzmi idiotycznie

specjalnie potrzebna wiedza. A ja... chciałbym już uciec z tego wstępu i przejść do czegoś „żywego”.

Jeszcze tylko taka uwaga. Proszę nie myśleć ciepło o rdzeniu np. CM0 - bo skoro jest prosty to łatwo się będzie nauczyć. To nie do końca tak działa. Np. Cortex-M4 ma wsparcie dla przetwarzania sygnałów – dodatkowe rozkazy *SIMD*. Brzmi to strasznie groźnie. Jednak póki nie grzebiemy w *asm* i (jako początkujący) nie skupiamy się na ekstremalnej optymalizacji kodu, to średnio nas to wszystko obchodzi. To problem kompilatora by kod z C przetrawić w taki sposób, ażeby wykorzystać potencjał rdzenia (te dodatkowe rozkazy). Po prostu spora część tych zaawansowanych mechanizmów jest zwykle niewidoczna dla programisty. Oczywiście jest też druga strona medalu – im potężniejszy rdzeń tym bardziej rozbudowane peryferia mikrokontrolera, ale nie ma co się bać.

Jedną z zalet architektury ARM Cortex-M, jaką można zauważyć od razu po przesiadce z AVR jest wspólna, liniowa przestrzeń adresowa²⁴. O co chodzi? W AVR pamięci *SRAM*, *Flash*, *EEPROM* miały oddzielne przestrzenie adresowe. Każda pamięć adresowana była „od zera”. Tzn. jakaś wartość mogła być w pamięci SRAM pod adresem 0; w pamięci Flash pod adresem 0 i w EEPROMie pod zerowym adresem. I mało który z AVRowych hobbystów przypuszczał, że w ogóle może być inaczej (ze mną włącznie). Takie rozwiązanie ma jednak drobną niedogodność: odczytaj mi wartość spod adresu 0x0032... ale – zapytasz - z której pamięci (z której przestrzeni adresowej)? Przy odrębnych przestrzeniach adresowych, sam adres nie wystarcza aby coś odczytać – trzeba jeszcze wiedzieć o której przestrzeni chodzi (nie wystarczy znać numeru mieszkania, potrzebny jeszcze numer klatki schodowej). Co to oznacza w praktyce? W praktyce następstwem tego są wszystkie cyrki jakich się dokonuje aby odczytać stałe z pamięci Flash i EEPROM. Spróbuj np. napisać jedną uniwersalną funkcję (na AVR), która jako argument przyjmuje wskaźnik na coś we Flashu lub SRAMie...²⁵.

A jak może być inaczej? A np. tak jak w Cortexie. Tutaj jest jedna wspólna przestrzeń adresowa. Tzn. w uproszczeniu (proszę jednym okiem czytać Poradnik, a drugim podglądać sobie mapkę pamięci - *Memory Map* - z *datasheetu* mikrokontrolera - mapka naszym przyjacielem!²⁶):

- pamięć Flash zajmuje adresy od 0x0800 0000 do 0x1FFF FFFF
- pamięć SRAM to adresy od 0x2000 0000 do 0x3FFF FFFF
- rejesty układów peryferyjnych zajmują pamięć od 0x4000 0000 do 0x5FFF FFFF

24 nie jestem pewny czy poprawnie to nazywam

25 od niedawna w GCC pojawiło się pewne, częściowe „obejście problemu” - nazwane przestrzenie adresowe (*_flash*, *_memx*, *itp*)

26 jeśli nie wiesz jak znaleźć mapę pamięci to wróć tu po przeczytaniu rozdziału 2.4

dalej jest jeszcze kontroler zewnętrznej pamięci (*FSMC/FMC*) i rejestrów rdzenia. I to jest **piękne**. Dostajemy adres np. *0x2001 0300* i od razu wiemy, że to gdzieś w pamięci SRAM. Ta wiedza szczególnie przydaje się przy debugowaniu, od razu możemy wyłapać np. próbę zapisu do pamięci Flash (pamięć tylko do odczytu). Nie potrzeba żadnego cyrkowania aby określić o jaką przestrzeń chodzi. Wolność wskaźnikom! Możemy jednym i tym samym wskaźnikiem latać sobie po pamięci Flash, SRAM, rejestrach peryferiów i pamięci zewnętrznej. A stałe same lądują tam gdzie ich miejsce, czyli w pamięci Flash. Cytując Korę: „*Kocham, kocham, kocham!*”.

Co warto zapamiętać z tego rozdziału?

- rdzenie CM0 i CM0+ to proste rdzenie do aplikacji low-cost i low-power
- rdzeń CM3 to „podstawowy” Cortex
- rdzenie CM4 i CM7 są najszybsze i najbardziej rozbudowane
- rdzeń ma swoje układy peryferyjne (*SysTick, NVIC, FPU, ...*)
- wspólna przestrzeń adresowa!

2.3. Hardware („*Nemo sine vitiis est!*”²⁷)

Na początek polecam coś gotowego (zestaw ewaluacyjny / edukacyjny / uruchomieniowy / startowy / demonstracyjny / jak zwał tak zwał) z Cortexem-M3. Dobrze by płytka była wyposażona w:

- minimum 2 diody świecące do wykorzystania wedle uznania
- przynajmniej dwa przyciski do wykorzystania w programie
- osobny przycisk reset
- **wygodne** złącze zasilania, które nie wygląda jakby miało zaraz odpaść
- wbudowany programator/debugger lub wygodne złącze do podpięcia się z zewnętrznym urządzeniem
- wyprowadzone **wszystkie** nóżki mikrokontrolera (na listwy grzebieniowe czy coś w tym stylu)
- czytelny opis wszystkich wyprowadzeń z poprzedniego punktu (i na Boga! nie na spodniej stronie płytki!)
- rezonator kwarcowy (zwykły i zegarkowy)
- gniazdo baterii pastylkowej (to już wisienka na torcie)

²⁷ „*Nikt nie jest bez wad.*”

Wszelkie inne bajery są mile widziane pod warunkiem, że można je odłączyć. Uwierz mi... żyroskopy, akcelerometry i inne pierdoły są fajne tylko na ulotce reklamowej zestawu. W rzeczywistości najczęściej przeszkadzają bo blokują pin związanego z jakąś frapującą funkcją alternatywną (np. wyjście przetwornika cyfrowo analogowego).

Płytek jest bardzo dużo na rynku. Ja mam aktualnie -dwie- trzy:

- HY-mini STM32 z mikrokontrolerem STM32F103 (Cortex-M3, 256kB Flash, 48kB SRAM, przejściówka UART-USB, złącze kart micro-SD, odłączany wyświetlacz²⁸ TFT 3,2" 320x240 z panelem dotykowym)
- STM32F429-Discovery z mikrokontrolerem STM32F429 (Cortex-M4F, 2MB Flash, 256kB SRAM, programator/debugger na pokładzie, 2.4" QVGA TFT LCD z dotykiem, zewnętrzny SRAM 64Mb, żyroskop i jakieś pierdoły)
- Nucleo-F334R8 z mikrokontrolerem STM32F334R8T6 (Cortex-M4 z FPU, 64kB flash, 16kB sram, programator/debugger STLink V2-1 na pokładzie)

i na nich będę się dalej opierał. Dokładniej STM32F103 będzie (najczęściej) punktem wyjścia przy poznawaniu jakiegoś peryferium. A jeśli w STM32F429 lub STM32F334 wygląda ono inaczej, to zostaną omówione różnice. Wsparcie dla F334 zostało dopisane po publikacji Poradnika - opisy odnoszące się do tego mikrokontrolera mogą nie być tak rozbudowane jak w przypadku F103 i F429, gdyż nieco trudniej jest mi postawić się w roli kogoś kto dopiero zaczyna przygodę z STM32. Poza tym ileż razy można się zachwycać tymi samymi peryferialami :) Nie podoba się? To trudno.

HY-mini to wyrób chiński. Na początku wydawał się strasznie delikatny – kilka lat minęło i żyje nadal. Warto umyć płytę przed pierwszym użyciem z cynowych śmieci i poprawić luty większych elementów. Teraz żeby było śmieszniej: Discovery to firmowa płytką ST kupiona w „pewnej” hurtowni. W porównaniu z HY-mini, Discovery to straszny szajs :) Jest koszmarnie polutowana, jest pełno błędów w warstwie opisowej, wyświetlacz odpada od płytki (obowiązkowo trzeba go przykleić bo odpadnie permanentnie). Do tego pełny opis goldpinów jest na spodzie płytki, a same goldpiny (od góry) są jakieś krótkie i przewody połączeniowe z nich spadają. Nawet dokumentacja HY-mini jest przyjemniejsza niż Discoverki. „You get what you pay for” i tyle w temacie. Nucleo to najświeższa hurysa w moim rajskim ogrodzie. To co mnie zaskoczyło, to wielkość płytki. Na żywo wydaje się o wiele mniejsza niż na zdjęciach. Do wad Nucleo zaliczyłbym białą soldermaskę (ładnie się prezentuje, ale w ogóle nie widać ścieżek), brak opisów przy listwach pinowych oraz zbyt małą ilość przycisków i diod do wykorzystania w programie (po

²⁸ we wszystkich przykładach pokazanych w Poradniku, bazujących na tej płytce, wyświetlacz jest odłączony aby nie blokował pinów mikrokontrolera

jednej sztuce²⁹). Płytką prezentuje się o wiele lepiej niż Discovery. Jest schludnie polutowana, nie klei się od resztek topnika. Elementy nie są pokrzywione. Ogólnie - podoba mi się to maleństwo :)

Uwaga debugger obowiązkowo! Przy AVRkach można się było bawić w debugowanie za pomocą UARTu, diod itp. środków zastępczych. Przy STM nie wyobrażam sobie nauki bez możliwości podejrzenia co się dzieje. Nie i już! Debugger musi być. Daje nam możliwość wejścia do środka mikrokontrolera i podejrzenia co też on czyni – możemy zatrzymać program w dowolnym momencie, odczytać wartości rejestrów i zmiennych, wymusić jakieś wartości rejestrów itd.

2.4. Piśmiennictwo („*Littera docet, littera nocet.*”³⁰)

Żaden poradnik nie nauczy programowania mikrokontrolerów. Może pomóc na starcie, rozwiać jakieś wątpliwości, ułatwić, podpowiedzieć etc... Ale najważniejsze jest aby nauczyć się samemu zdobywać informacje, rozwiązywać problemy, zacząć ufać sobie, swoim pomysłem, bazować na dokumentacji i odczepniać się od poradników i kursów. Ja też nie urodziłem się z jakąś wiedzą na temat STMów, nie dostałem jej spod lady, tudzież z jakichś zagranicznych ksiąg tajemnych. Więc skąd?

Za darmo z Internetu³¹ :) Przede wszystkim z najpewniejszego źródła – czyli dokumentacji producenta. I tu mała, ale bardzo ważna uwaga (skupić się proszę). Atmel przyzwyczaił Nas do dokumentacji jedno-pedofowej. Tzn. wystarczyło ściągnąć notę konkretnego mikrokontrolera i było w niej wszystko. Totalnie wszystko. Informacje o rdzeniu, opisy instrukcji assemblera, przykłady kodów, mapy pamięci, rejestrów, opisy peryferiów, dane elektryczne i mechaniczne, errata. *All-in-one...* jak parówka. W STMachine jest odrobinę inaczej. I wbrew pozorom jest to nawet logiczne. Dokumentacji jest delikatnie mówiąc cholernie dużo. Ale spokojnie, nie wszystko jest potrzebne.

Najpierw taki mały przykład dla odprężenia: kupiliśmy używany samochód z nieoryginalnym radiem i alarmem. Gdzie będziemy szukać informacji jak ustawić radioodtwarzacz na naszą ulubioną stację – w instrukcji samochodu, instrukcji radioodtwarzacza, instrukcji alarmu czy PoRD (Prawo o Ruchu Drogowym)? Głupie pytanie, prawda? Oczywiście, że w instrukcji radia. W końcu producent samochodu nie wyprodukował radia tylko (on lub ktoś inny) wpakował gotowy produkt do swojego wyrobu. Czujesz już analogię do mikrokontrolera formy ST i rdzenia firmy ARM? Nigdzie się nie spieszamy, więc jeśli nie czujesz to podumaj nad tym chwilę – jak

29 chodzi mi po głowie mały gwałt na Nucleo - wlutowanie dodatkowego leda [kilka tygodni później] a słowo ciałem się stało - patrz dodatek 7

30 „*Littera uczy, littera szkodzi.*”

31 tak! w Internecie są nie tylko gołe baby... też byłem zdziwiony!

teraz załapiesz to potem nie będziesz błądzić w dokumentacji :) Idziemy dalej, czy w instrukcji obsługi samochodu są informacje jak jechać po rondzie (tak tak wiem... skrzyżowaniu o ruchu okrężnym)? No nie, dlaczego? Bo zasady ruchu drogowego są wspólne dla wszystkich samochodów i są opisane w innym dokumencie³². Ma to sens, prawda? Wyobrażasz sobie jaką grubą byłaby instrukcja obsługi samochodu gdyby zawrzeć w niej przepisy ruchu drogowego (i to dla różnych państw) i szczegóły techniczne budowy dróg, znaków itd... Tak samo jest z mikrokontrolerami STM. Początkowo nie jest łatwo się połapać bo wszystko jest nowe, inne, skomplikowane (tu akurat przyzwyczajenia z AVR przeszkadzają). Jak to więc jest z tymi eS-Te-eM-ami dokładnie?

Rdzeń został opracowany przez holding ARM³³, stąd też i ta firma przygotowała dokumentację rdzenia (jest jak radio samochodowe). Żeby było śmieszniej ST, czyli producent mikrokontrolera, oferuje swoje opracowanie dokumentacji rdzenia – nazywa się toto **Programming Manual** (w skrócie **PM**) i w zupełności wystarcza na początku. Jeśli ktoś jest żądnym szczegółowej wiedzy nt. rdzenia, to wtedy warto udać się na stronę ARMa i poszukać następujących pdfów (przykładowo dla *Cortexa M3*):

- *Cortex-M3 Devices Generic User Guide* – w miarę strawnie opisane wszystko co związane z rdzeniem – praktycznie to samo co w *Programming Manualu* od ST (przy czym *User Guide ARMa* czyta mi się jakoś przyjemniej niż *PM*, ale to tak OT)
- *Cortex-M3 Technical Reference Manual* – to samo co wyżej tylko bardziej szczegółowo
- *Cortex-M3 Software Developers Errata Notice* – errata rdzenia dla dociekliwych

Tutaj pojawia się mała pułapka (duże słowo) dotycząca układów peryferyjnych rdzenia (np. licznik *SysTick*, który zaraz pokochamy) – ich opis znajduje się w dokumentacji rdzenia a nie mikrokontrolera, bo są częścią rdzenia! Gwarantuję, że będzie się mylić na początku :)

Biblią, opisującą wszystkie peryferia mikrokontrolera (np. liczniki, porty I/O, ADC...) jest **Reference Manual** (w skrócie **RM**). Jest to dokument wspólny dla całej rodziny/linii mikrokontrolerów (kodeks drogowy wspólny dla wszystkich samochodów). Uwaga! W *RM* opisane są wszelkie peryferia jakie mogą się pojawić w mikrokontrolerach danej rodziny/linii. Samemu trzeba sprawdzić (w *datasheetie* konkretnej kostki) czy w „naszym” mikrokontrolerze dany układ jest zaimplementowany i jakie tryby pracy obsługuje. Inaczej można się naciąć na próbę uruchomienia np. drugiego przetwornika ADC, którego w konkretnym mikrokontrolerze nie ma³⁴ :)

32 to nie jest idealna metafora, ale jakoś nie mogę znaleźć lepszej

33 www.arm.com; ARM to jednocześnie nazwa architektury i firmy która ją opracowała

34 przykład z życia wzięty

Kolejną ważną pozycją jest ***datasheet*** – w nim znajdziemy wspomniane przed chwilą informacje o tym co konkretnie posiada dany mikrokontroler (jakie peryferia), ponadto datasheet zawiera typowe informacje elektryczno-mechaniczne.

Ostatni „niezbędny” pdf to ***errata*** (od ST, nie od ARM). Przy AVRach mało kto zaglądał do jakichś errat czy changelogów. Tutaj ze względu na stosunkową świeżość i stopień komplikacji mikrokontrolera, liczba błędów jest proporcjonalnie większa. Warto chociaż z grubsza przejrzeć erratę, żeby wiedzieć czego można się spodziewać. Wbrew pozorom, część baboli nie jest związana z jakimiś „egzotycznymi” trybami pracy peryferiów. Nawet w tym poradniku nadziejemy się na kilka punktów z erraty.

Dodatkowo ST wyprodukowało kilkadziesiąt krótszych dokumentów - not aplikacyjnych (***Application Note - AN***) - opisujących w sposób dokładniejszy, niż we wspomnianej już dokumentacji, jakieś konkretne zagadnienia (Atmel zresztą ma podobnie). Przykłady:

- *AN2629* - opisuje sposoby zmniejszania poboru energii
- *AN2548* - wybrane przykłady użycia *DMA*
- *AN2834* - sposoby poprawy dokładności pomiarów *ADC*
- *AN3116* - szczegółowe opisy, z przykładami, trybów pracy *ADC*
- *AN2586* - informacje użyteczne przy projektowaniu własnych *PCB* z układami STM32
- *AN4013* - opisy i szczegóły konfiguracji liczników
- ... mnóstwo innych dostępnych na stronie ST

Wszystkie wspomniane wyżej dokumenty są dostępne całkowicie darmowo. Co najwyżej wymagają utworzenia konta i zapłaszenia uwagi, że nie jesteśmy agentami wrogiego wywiadu czy innymi niegodziwcami. Ważne jest aby noty pobierać wyłącznie ze strony producenta (a nie pierwszy lepszy wynik z googli) – gwarantuje to, że będą w najnowszej wersji. Każdy dokument zawiera informacje o rewizji i opis zmian. Po roku leżenia na dysku, okazało się, że połowa moich zbiorów o STM32 się zdezaktualizowała. Zmiany czasem są „kosmetyczne”, czasem diametralne³⁵.

Krótki poradnik jak szukać dokumentów na stronie ST:

1. www.st.com
2. z górnej belki wybieramy *Products → Microcontrollers*

³⁵ jak np. opis kalibracji ADC: <http://www.elektroda.pl/rtvforum/viewtopic.php?p=13790376#13790376>
(wątek: [STM32] Kalibracja ADC. Jak często?)

3. z menu po lewo *STM32 32-bit ARM Cortex MCUs*
 4. z menu po lewo wybieramy interesującą nas serię mikrokontrolerów, potem konkretne oznaczenie
 5. po lewo mamy menu „*Resources*” - tam grzebiemy :)
-

W tym miejscu wspomnę jeszcze raz, żeby potem nie było rozczarowania, zanim zaczniesz cokolwiek robić z STMem zobacz w datasheetie co Twój mikrokontroler oferuje! Przykład: *Reference Manual RM0008* (kodeks drogowy) dotyczy mikrokontrolerów:

- *STM32F101...*
- *STM32F102...*
- *STM32F103...*
- *STM23F105...*
- *STM32F107...*

W rozdziale o licznikach znajdziemy opisy takich oto liczników:

- *Advanced Control Timers* (TIM1 i TIM8)
- *General-purpose Timers* (TIM2 to TIM5)
- *General-purpose Timers* (TIM9 to TIM14)
- *Basic Timers* (TIM6 i TIM7)

czyli w sumie mamy:

- dwa liczniki „zaawansowane” (TIM1, 8)
- dziesięć liczników „ogólnego zastosowania” (TIM2, 3, 4, 5, 9, 10, 11, 12, 13, 14)
- dwa liczniki „podstawowe” (TIM6, 7)

ale to nie znaczy, że każdy konkretny układ podlegający pod ten *Reference Manual* ma wszystkie te liczniki. Jeśli zerkiemy do *datasheetu* od STM32F103xC/xD/xE, to w rozdziale *Device Overview* znajdziemy tabelkę, która pokazuje, że ten mikrokontroler ma:

- dwa liczniki zaawansowane
- **cztery** liczniki ogólnego zastosowania
- dwa liczniki podstawowe

Ogólnych jakby ciut mniej. Informacje o tym, które konkretnie liczniki są w tym mikrokontrolerze, znajdziemy w rozdziale *Overview → Timers and watchdogs (TIM2,3,4,5)*. Czasem trzeba trochę powertować, ale idzie się przyzwyczaić :) Korzystając z RMa należy jeszcze zwrócić uwagę czy w tytule rozdziału nie ma informacji o tym jakiej linii mikrokontrolerów rozdział dotyczy! Np. w RM mogą być dwa rozdziały o konfiguracji jakiegoś układu – osobne dla różnych linii mikrokontrolerów. Patrz RM0008 i rozdziały:

- *Low-, medium-, high- and XL-density reset and clock control (RCC)*
- *Connectivity line devices: reset and clock control (RCC)*

Oba rozdziały dotyczą bloku RCC, tylko dla różnych linii mikrokontrolerów. Linie były omówione [tu](#). Na koniec, warto jeszcze sprawdzić czy w *erracie* nie ma czegoś ciekawego o peryferialu który chcemy wykorzystać :)

Jak dotąd w miarę proste prawda? Spokojnie. Mina Ci zrzędnie, drogi Czytelniku, gdy ściagniesz wspomniane dokumenty. Lekko licząc będzie tego ok 2-3 tysięcy stron. Luz. Przecież wszystkiego nie trzeba czytać. Gdy w AVRze korzystaliśmy z licznika to też nie czytaliśmy rozdziałów o UARTcie, ADC i wsparciu dla bootloaderów, prawda? Choć przekartkować pewnie nie zaszkodzi :)

A z czego warto korzystać poza oficjalną dokumentacją? Wydaje mi się, że:

- www.elektroda.pl → polecam wpisać w szukajce *STM32* i czytać po kolej jak leci żeby się oswoić trochę z tematem, słownictwem i najczęstszymi problemami... że dużo? To chyba dobrze... nikt nie mówił, że będzie łatwo i szybko
- www.freddiechopin.info → bardzo polecam w całej rozciągłości (opis konfiguracji środowiska, przykładowe projekty i narzędzia)
- <https://my.st.com/public/STe2ecommunities/mcu/default.aspx> – forum „wsparcia” ST
- www.youtube.com → kurs *Embedded Programming Lessons* na kanale *Quantum Leaps, LLC*; choć to może trochę za szczegółowe podejście jak dla początkujących hobbystów – oceń sam. Jak nie teraz to może później się przyda.
- www.google.pl → STFW

Są jeszcze książki dotyczące rdzeni (nie mikrokontrolerów) z serii „*Definitive Guide to ARM Cortex...*”. W żadnym razie nie są to pozycje obowiązkowe ale na pewno nie

zaszkodzą - wiedza o niskopoziomowym działaniu procesora bardzo przydaje się przy debugowaniu.

Na koniec przyjrzyjmy się jeszcze budowie dokumentu RM. Każdy rozdział opisujący jakiś układ peryferyjny zbudowany jest z grubsza podobnie i składa się z:

- krótkiego podsumowania funkcji układu – podrozdziały typu: *Introduction, Main features*
- szczegółowego opisu wszystkich dostępnych trybów pracy i konfiguracji – podrozdział: *Functional description*
- zestawienia (z opisem) wszystkich rejestrów konfiguracyjnych danego bloku – podrozdział: *Registers*

Przy bardziej rozbudowanych peryferiach podrozdziałów może być więcej, ale powyższe zawsze występują. Czemu to jest ważne? Bo łatwiej się odnaleźć w dokumentacji. Np. jeśli z grubsza wiemy co chcemy ustawić, tylko nie pamiętamy nazw konkretnych bitów konfiguracyjnych to nie ma potrzeby wertowania całego rozdziału – bo na końcu zawsze jest podrozdział *Registers* w którym każdy bit jest krótko opisany.

Co warto zapamiętać z tego rozdziału?

- rdzeń (i jego peryferia) opisane są w innej dokumentacji niż peryferia mikrokontrolera
- na początek trzeba się zaopatrzyć w cztery pdfy ze strony ST:
 - *Reference Manual* – opis peryferiów mikrokontrolera (działanie i konfiguracja)
 - *Programming Manual* – opis rdzenia i jego peryferiów
 - *Datasheet* – szczegóły konkretnej kostki, w tym opis elektryczno - mechaniczny
 - *Errata* – opis rzeczy, które działają... nie do końca poprawnie :)
- RM dotyczy kilku podrodzin mikrokontrolerów i opisuje wszystkie możliwe peryferia; nie każdy mikrokontroler będzie miał wszystkie opisane bloki
- trzeba czytać, czytać i... czytać – i to już, od zaraz :)

2.5. Software („*Ex malis eligere minima oportet*”³⁶)

Kolejna pułapka dla początkujących. Programów, środowisk, toolchainów oraz poradników ich konfiguracji jest sporo. Coś trzeba wybrać... Każdy szanujący się poradnik dla początkujących, zawiera rozdział o konfiguracji wybranego środowiska. Ten jest hardcore'owy i jedyny w swoim

³⁶ „Ze zlego należy wybierać mniejsze зло.”

rodzaju, więc w nim tego nie będzie³⁷. Dlatego, że to nie ma sensu. Dokładny opis środowiska, konfiguracji i obsługi to byłoby kilkadziesiąt/set stron, które za kilka miesięcy będą nieaktualne bo coś się zmieni, ikonka będzie gdzie indziej lub wyjdzie coś nowego i fajniejszego. Poza tym to jest IMO nudne... Zresztą opisów w sieci są dziesiątki – odsyłam więc do sieci z wskazaniem na:

- <http://www.freddiechopin.info/pl/artykuly/35-arm/59-arm-toolchain-tutorial>
- <http://www.elektroda.pl/rtyforum/viewtopic.php?p=10341774#10341774>

(wątek: *ARM toolchain - tutorial - jak to połączyć?*)

Uwaga! Najpierw wszystko czytamy, potem działamy! Wiem, że temat na Elektrodzie jest długi, ale... nikt nie mówił, że będzie łatwo. Jak komuś nie styknie cierpliwości na przeczytanie tego tematu to na rozwiązywanie problemów przy programowaniu też mu nie starczy.

Zgodnie z powyższym swoje środowisko oparłem o *Eclipse*, *GCC* i *OpenOCD*. Bo:

- za darmo
- bez ograniczeń
- działa na Linuksie³⁸
- nic nie narzuca (żadnych kreatorów, bibliotek i innych wodotrysków)
- bo tak i już

Nie jest to najłatwiejsza droga. Nie będę ukrywał, że uruchomienie i ogarnięcie środowiska oraz rozpracowanie przykładowego projektu ze strony *Freddiego Chopina* zajęło mi za pierwszym razem coś koło 5 dni. Skoro ja dałem radę to Ty też dasz. Z tego co widzę na forach, początkujący w dużej mierze uciekają do *CooCoxa* (kokosa), *Keila*, ewentualnie *Eclipse'a* z wtyczką dla *ARMów*... bo jest łatwiej wystartować. Ja nic nie narzucam – każde środowisko które działa i pasuje użytkownikowi jest ok. Byleby pozwoliło skupić się na programowaniu a nie walce z samym narzędziem.

2.6. Słowo o bibliotekach („*Relata referto*”³⁹)

Jeśli nie słyszałeś nic o bibliotekach w kontekście STMów to możesz się wstydzić. Bo to znaczy, że jeszcze nie zacząłeś czytać o STMacie w necie. Sio odrabiać zaległości – co najmniej tydzień wertowania netu za karę.

37 tak - można lamentować

38 i w ogóle działa :)

39 „*Powtarzam, co usłyszałem.*”

Temat bibliotek jest dosyć kontrowersyjny. O co chodzi? Sprawa wygląda następująco: firma ARM stworzyła standard *CMSIS*⁴⁰... właściwie to ja nie do końca czuję czym jest CMSIS, bo to niby standard, ale często spotykam pojęcie „biblioteka CMSIS”. To w końcu standard czy biblioteka? Mniejsza, jak ktoś chce to niech sobie szuka. Generalnie chodzi (chyba) o to, że CMSIS to standard który m.in. opisuje sposób dostępu do rejestrów układów peryferyjnych mikrokontrolera (poprzez struktury). Konkretny producent mikrokontrolera (np. ST), przygotowując plik nagłówkowy z definicjami rejestrów i bitów konfiguracyjnych, ma go stworzyć tak aby był zgodny ze standardem CMSIS. Ponadto CMSIS dostarcza kilka podstawowych funkcji związanych z obsługą rdzenia. Czyli:

- definicje rejestrów i bitów konfiguracyjnych peryferiów rdzenia (np. SysTicka)
 - funkcje związane z obsługą rdzenia (np. włącz/wyłącz przerwania)
 - funkcje *intrinsic* - proste funkcje w C (właściwie wstawki asm) umożliwiające wywołanie konkretnego rozkazu asm (np.: `NOP()`, `WFI()`, ...)

ST dokłada do powyższych kropek swoje biblioteki, zawierające definicje bitów i rejestrów konfiguracyjnych peryferiów (pliki nagłówkowe zgodne ze standardem CMSIS) oraz gotowe funkcje i narzędzia służące do obsługi peryferiów. Przykładem takiej biblioteki dostarczonej przez ST jest *Standard Peripheral Library (SPL)*⁴¹. Spora część przykładów w nacie i polskie książki o STM32 bazują na tym (po)tworku. I tu właśnie jest pies pogrzebany :) Korzystać z biblioteki czy programować bezpośrednio „na rejestrach”? Osobiście jako „wychowanek” Elektrody wyssałem z wiedzą również i niechęć do biblioteki SPL (czyli do tych funkcji obsługujących peryferia) i pochodnych. Nie wiem czy jest sens podawać argumenty za i przeciw, jak ktoś chce to bez problemu znajdzie tasiemcowe dyskusje na forach. Może tylko jeden: poniżej znajduje się kod (napisany z użyciem biblioteki) konfigurujący układ FSMC do współpracy ze sterownikiem wyświetlacza LCD w zestawie *HY-mini* (kod pochodzi z przykładów od producenta zestawu)⁴²:

```
1. | FSMC_NORSRAMInitTypeDef  FSMC_NORSRAMInitStructure;
2. | FSMC_NORSRATimingInitTypeDef  FSMC_NORSRATimingInitStructure;
3. | /* FSMC */
4. | FSMC_NORSRATimingInitStructure.FSMC_AddressSetupTime = 10;
5. | FSMC_NORSRATimingInitStructure.FSMC_AddressHoldTime = 0;
6. | FSMC_NORSRATimingInitStructure.FSMC_DataSetupTime = 10;
7. | FSMC_NORSRATimingInitStructure.FSMC_BusTurnAroundDuration = 0x00;
8. | FSMC_NORSRATimingInitStructure.FSMC_CLKDivision = 0x00;
9. | FSMC_NORSRATimingInitStructure.FSMC_DataLatency = 0x00;
10. | FSMC_NORSRATimingInitStructure.FSMC_AccessMode = FSMC_AccessMode_A;
11. | FSMC_NORSRAMInitStructure.FSMC_Bank = FSMC_Bank1_NORSRAM1;
12. | FSMC_NORSRAMInitStructure.FSMC_DataAddressMux = FSMC_DataAddressMux_Disable;
13. | FSMC_NORSRAMInitStructure.FSMC_MemoryType = FSMC_MemoryType_SRAM;
14. | FSMC_NORSRAMInitStructure.FSMC_MemoryDataWidth = FSMC_MemoryDataWidth_16b;
15. | FSMC_NORSRAMInitStructure.FSMC_BurstAccessMode = FSMC_BurstAccessMode_Disable;
```

40 Cortex Microcontroller Software Interface Standard

41 biblioteka SPL nie jest już rozwijana przez ST, zastąpił ją STM32Cube

42 proszę się nie przestraszać – chodzi mi tylko o porównanie dwóch kodów „na pierwszy rzut oka”

```

16. FSMC_NORSRAMInitStructure.FSMC_WaitSignalPolarity = FSMC_WaitSignalPolarity_Low;
17. FSMC_NORSRAMInitStructure.FSMC_WrapMode = FSMC_WrapMode_Disable;
18. FSMC_NORSRAMInitStructure.FSMC_WaitSignalActive = FSMC_WaitSignalActive_BeforeWaitState;
19. FSMC_NORSRAMInitStructure.FSMC_WriteOperation = FSMC_WriteOperation_Enable;
20. FSMC_NORSRAMInitStructure.FSMC_WaitSignal = FSMC_WaitSignal_Disable;
21. FSMC_NORSRAMInitStructure.FSMC_AsynchronousWait = FSMC_AsynchronousWait_Disable;
22. FSMC_NORSRAMInitStructure.FSMC_ExtendedMode = FSMC_ExtendedMode_Disable;
23. FSMC_NORSRAMInitStructure.FSMC_WriteBurst = FSMC_WriteBurst_Disable;
24. FSMC_NORSRAMInitStructure.FSMC_ReadWriteTimingStruct = &FSMC_NORSRAMTimingInitStructure;
25. FSMC_NORSRAMInit(&FSMC_NORSRAMInitStructure);
26. /* FSMC */
27. FSMC_NORSRAMTimingInitStructure.FSMC_AddressSetupTime = 3;
28. FSMC_NORSRAMTimingInitStructure.FSMC_AddressHoldTime = 0;
29. FSMC_NORSRAMTimingInitStructure.FSMC_DataSetupTime = 3;
30. FSMC_NORSRAMTimingInitStructure.FSMC_BusTurnAroundDuration = 0x00;
31. FSMC_NORSRAMTimingInitStructure.FSMC_CLKDivision = 0x00;
32. FSMC_NORSRAMTimingInitStructure.FSMC_DataLatency = 0x00;
33. FSMC_NORSRAMTimingInitStructure.FSMC_AccessMode = FSMC_AccessMode_A; /* FSMC */
34. FSMC_NORSRAMInitStructure.FSMC_WriteTimingStruct = &FSMC_NORSRAMTimingInitStructure;
35. FSMC_NORSRAMInit(&FSMC_NORSRAMInitStructure);
36.
37. /* Enable FSMC Bank1_SRAM Bank */
38. FSMC_NORSRAMCmd(FSMC_Bank1_NORSRAM1, ENABLE);

```

Jest tam wywoływana m.in. funkcja z biblioteki SPL: *FMSC_NORSRAMInit(...)*, więc dla uzupełnienia wrzućmy jeszcze jej kod:

```

1. void FMSC_NORSRAMInit(FSMC_NORSRAMInitTypeDef* FSMC_NORSRAMInitStruct) {
2. /* Check the parameters */
3. assert_param(IS_FSMC_NORSRAM_BANK(FSMC_NORSRAMInitStruct->FSMC_Bank));
4. assert_param(IS_FSMC_MUX(FSMC_NORSRAMInitStruct->FSMC_DataAddressMux));
5. assert_param(IS_FSMC_MEMORY(FSMC_NORSRAMInitStruct->FSMC_MemoryType));
6. assert_param(IS_FSMC_MEMORY_WIDTH(FSMC_NORSRAMInitStruct->FSMC_MemoryDataWidth));
7. assert_param(IS_FSMC_BURSTMODE(FSMC_NORSRAMInitStruct->FSMC_BurstAccessMode));
8. assert_param(IS_FSMC_ASYNCWAIT(FSMC_NORSRAMInitStruct->FSMC_AsynchronousWait));
9. assert_param(IS_FSMC_WAIT_POLARITY(FSMC_NORSRAMInitStruct->FSMC_WaitSignalPolarity));
10. assert_param(IS_FSMC_WRAP_MODE(FSMC_NORSRAMInitStruct->FSMC_WrapMode));
11. assert_param(IS_FSMC_WAIT_SIGNAL_ACTIVE(FSMC_NORSRAMInitStruct->FSMC_WaitSignalActive));
12. assert_param(IS_FSMC_WRITE_OPERATION(FSMC_NORSRAMInitStruct->FSMC_WriteOperation));
13. assert_param(IS_FSMC_WAITE_SIGNAL(FSMC_NORSRAMInitStruct->FSMC_WaitSignal));
14. assert_param(IS_FSMC_EXTENDED_MODE(FSMC_NORSRAMInitStruct->FSMC_ExtendedMode));
15. assert_param(IS_FSMC_WRITE_BURST(FSMC_NORSRAMInitStruct->FSMC_WriteBurst));
16. assert_param(IS_FSMC_ADDRESS_SETUP_TIME(FSMC_NORSRAMInitStruct->FSMC_ReadWriteTimingStruct->FSMC_AddressSetupTime));
17. assert_param(IS_FSMC_ADDRESS_HOLD_TIME(FSMC_NORSRAMInitStruct->FSMC_ReadWriteTimingStruct->FSMC_AddressHoldTime));
18. assert_param(IS_FSMC_DATASETUP_TIME(FSMC_NORSRAMInitStruct->FSMC_ReadWriteTimingStruct->FSMC_DataSetupTime));
19. assert_param(IS_FSMC_TURNAROUND_TIME(FSMC_NORSRAMInitStruct->FSMC_ReadWriteTimingStruct->FSMC_BusTurnAroundDuration));
20. assert_param(IS_FSMC_CLK_DIV(FSMC_NORSRAMInitStruct->FSMC_ReadWriteTimingStruct->FSMC_CLKDivision));
21. assert_param(IS_FSMC_DATA_LATENCY(FSMC_NORSRAMInitStruct->FSMC_ReadWriteTimingStruct->FSMC_DataLatency));
22. assert_param(IS_FSMC_ACCESS_MODE(FSMC_NORSRAMInitStruct->FSMC_ReadWriteTimingStruct->FSMC_AccessMode));
23. /* Bank1 NOR/SRAM control register configuration */
24. FSMC_Bank1->BTCR[FSMC_NORSRAMInitStruct->FSMC_Bank] = (uint32_t) FSMC_NORSRAMInitStruct->FSMC_DataAddressMux
25. | FSMC_NORSRAMInitStruct->FSMC_MemoryType
26. | FSMC_NORSRAMInitStruct->FSMC_MemoryDataWidth
27. | FSMC_NORSRAMInitStruct->FSMC_BurstAccessMode
28. | FSMC_NORSRAMInitStruct->FSMC_AsynchronousWait
29. | FSMC_NORSRAMInitStruct->FSMC_WaitSignalPolarity
30. | FSMC_NORSRAMInitStruct->FSMC_WrapMode
31. | FSMC_NORSRAMInitStruct->FSMC_WaitSignalActive
32. | FSMC_NORSRAMInitStruct->FSMC_WriteOperation
33. | FSMC_NORSRAMInitStruct->FSMC_WaitSignal
34. | FSMC_NORSRAMInitStruct->FSMC_ExtendedMode
35. | FSMC_NORSRAMInitStruct->FSMC_WriteBurst;
36.
37. if (FSMC_NORSRAMInitStruct->FSMC_MemoryType == FSMC_MemoryType_NOR) {
38.     FSMC_Bank1->BCR[FSMC_NORSRAMInitStruct->FSMC_Bank] |= (uint32_t) BCR_FACCEN_Set;
39. }
40.
41. /* Bank1 NOR/SRAM timing register configuration */
42. FSMC_Bank1->BTCR[FSMC_NORSRAMInitStruct->FSMC_Bank + 1] = (uint32_t) FSMC_NORSRAMInitStruct->FSMC_ReadWriteTimingStruct-
43. >FSMC_AddressSetupTime
44. | (FSMC_NORSRAMInitStruct->FSMC_ReadWriteTimingStruct->FSMC_AddressHoldTime << 4)
45. | (FSMC_NORSRAMInitStruct->FSMC_ReadWriteTimingStruct->FSMC_DataSetupTime << 8)
46. | (FSMC_NORSRAMInitStruct->FSMC_ReadWriteTimingStruct->FSMC_BusTurnAroundDuration << 16)
47. | (FSMC_NORSRAMInitStruct->FSMC_ReadWriteTimingStruct->FSMC_CLKDivision << 20)
48. | (FSMC_NORSRAMInitStruct->FSMC_ReadWriteTimingStruct->FSMC_DataLatency << 24)
49. | (FSMC_NORSRAMInitStruct->FSMC_ReadWriteTimingStruct->FSMC_AccessMode);
50.
51. /* Bank1 NOR/SRAM timing register for write configuration, if extended mode is used */
52. if (FSMC_NORSRAMInitStruct->FSMC_ExtendedMode == FSMC_ExtendedMode_Enable) {
53.     assert_param(IS_FSMC_ADDRESS_SETUP_TIME(FSMC_NORSRAMInitStruct->FSMC_WriteTimingStruct->FSMC_AddressSetupTime));
54.     assert_param(IS_FSMC_ADDRESS_HOLD_TIME(FSMC_NORSRAMInitStruct->FSMC_WriteTimingStruct->FSMC_AddressHoldTime));
55.     assert_param(IS_FSMC_ADDRESS_HOLD_TIME(FSMC_NORSRAMInitStruct->FSMC_WriteTimingStruct->FSMC_AddressHoldTime));
56.     assert_param(IS_FSMC_DATASETUP_TIME(FSMC_NORSRAMInitStruct->FSMC_WriteTimingStruct->FSMC_DataSetupTime));
57.     assert_param(IS_FSMC_CLK_DIV(FSMC_NORSRAMInitStruct->FSMC_WriteTimingStruct->FSMC_CLKDivision));
58.     assert_param(IS_FSMC_DATA_LATENCY(FSMC_NORSRAMInitStruct->FSMC_WriteTimingStruct->FSMC_DataLatency));
59.     assert_param(IS_FSMC_ACCESS_MODE(FSMC_NORSRAMInitStruct->FSMC_WriteTimingStruct->FSMC_AccessMode));
60.     FSMC_Bank1->BTWTR[FSMC_NORSRAMInitStruct->FSMC_Bank] = (uint32_t) FSMC_NORSRAMInitStruct->FSMC_WriteTimingStruct-
61. >FSMC_AddressSetupTime
62. | (FSMC_NORSRAMInitStruct->FSMC_WriteTimingStruct->FSMC_AddressHoldTime << 4)
63. | (FSMC_NORSRAMInitStruct->FSMC_WriteTimingStruct->FSMC_DataSetupTime << 8)
64. | (FSMC_NORSRAMInitStruct->FSMC_WriteTimingStruct->FSMC_CLKDivision << 20)
65. | (FSMC_NORSRAMInitStruct->FSMC_WriteTimingStruct->FSMC_DataLatency << 24)

```

```

66.     | FSMC_NORSRAMInitStruct->FSMC_WriteTimingStruct->FSMC_AccessMode;
67. } else {
68.     FSMC_Bank1E->BWTR[FSMC_NORSRAMInitStruct->FSMC_Bank] = 0x0FFFFFFF;
69.
70. }

```

I teraz pytanie w formie zagadki z niespodzianką: ile linijek kodu zajmie skonfigurowanie bloku *FSMC* w **identyczny sposób**, ale bez użycia *ułatwiającej i przyspieszającej pracę* biblioteki SPL? Odpowiedź poniżej – kod robiący to samo bezpośrednio na rejestrach:

```

1. /* Control Register, BCR1 */
2. FSMC_Bank1->BCR1[0] = FSMC_BCR1_MWID_0 | FSMC_BCR1_WREN;
3.
4. /* Timing Register, BTR1 */
5. FSMC_Bank1->BTR1[1] = 0x0a | (0x0a<<8);
6.
7. /* Write timing register, BWTR1 */
8. FSMC_Bank1E->BWTR1[0] = 0x0fffffff;
9.
10. /* Enable FSMC Bank1_SRAM Bank */
11. FSMC_Bank1->BCR1[0] = FSMC_BCR1_MBKEN;

```

I niech mi ktoś jeszcze raz powie, że z biblioteką jest szybciej. Mój kod to cztery zapisy do rejestrów, a kod z biblioteką? Pierdylion niepotrzebnych operacji. Dosyć OT.

Podsumowując: biblioteki i programy do konfigurowania peryferiów (*SPL*, *HAL*, *CubeMX...*) zostawmy tym, którzy ich potrzebują. My lubimy rejesty, prawda? Nic złego natomiast nie ma w CMSIS (standardzie opisu peryferiów, plikach nagłówkowych i funkcjach przygotowanych przez ARM) oraz plikach nagłówkowych (od ST) z definicjami rejestrów⁴³.

Jak i skąd to wszystko zdobyć? Plik nagłówkowy mikrokontrolera oczywiście ze strony ST. Dokładniej trzeba pobrać paczkę z narzędziem STM32Cube i wyłuskać z niej odpowiedni plik nagłówkowy. Przykładowo dla STM32F103:

- www.st.com
- *Products → Microcontrollers → STM32 32-bit ARM Cortex MCUS → STM32F1 Series → STM32F103*
- *Software → STM32Cube*
- *STM32CubeF1*
- *Get Software → Download*
- w rozpakowanym archiwum: *Drivers → CMSIS → Device → ST → STM32F1xx → Include*
- w pliku *stm32f1xx.h* sprawdzamy nazwę pliku nagłówkowego dla naszego mikrokontrolera (np. mikrokontroler *STM32F103VC* to plik *stm32f103xe.h*)
- i mamy nasz garniec złota na krańcu tęczy – plik nagłówkowy mikrokontrolera z definicjami rejestrów i bitów: *stm32f103xe.h*

⁴³ no chyba, że ktoś ma ochotę sam sobie pedefiniować wszystkie rejesty i bity :)

Resztę można wywalić (w pliku nagłówkowym należy zakomentować odwołanie do pliku `system_stm32f...`, nie jest on do niczego potrzebny).

Pozostało jeszcze zdobycie kawałka od ARMa. Najnowszą wersję oczywiście pobieramy z oficjalnej strony firmy (wymaga założenia darmowego konta):

- www.arm.com
- *Products → Cortex-M Series → CMSIS*
- zakładka *Download CMSIS i free download*
- trzeba się zalogować (lub zarejestrować jeśli nie mamy konta)
- po prawo klik na: *Download Now*

W paczce jest sporo fajnych rzeczy – np. gotowe funkcje do operacji na macierzach i wektorach zoptymalizowane pod Cortexa. Polecam pogrzebać i poczytać dokumentację. Nas na początku interesują następujące pliki (*CMSIS/Include/*):

- *core_cm_x.h*⁴⁴ – zawiera definicje nazw rejestrów i bitów konfiguracyjnych peryferiów rdzenia (SysTick, NVIC, etc..) oraz kilka bardzo niskopoziomowych funkcji związanych z obsługą np. kontrolera przerwań (NVIC)
- *cmsis_gcc.h*⁴⁵ - funkcje operujące na rejestrach specjalnych rdzenia (np. dostęp do wskaźnika stosu i konfiguracja przerwań) oraz funkcje *intrinsic*, czyli proste wstawki asm umożliwiające wywołanie, z poziomu języka C, konkretnych rozkazów asm (np. `__WFI()`, `__NOP()`, itd...)

Uwaga! W starszych wersjach CMSIS (chyba do wersji 4.3) funkcje operujące na rejestrach specjalnych rdzenia i funkcje *intrinsic*, były rozdzielone na kilka plików (m.in. *core_cmFunc.h*, *core_cmInstr.h*). W najnowszym CMSISie nastąpiła mała reorganizacja. Funkcje zostały umieszczone we wspólnym pliku. Plik ten występuje w kilku wersjach, dla różnych kompilatorów, np.:

- *cmsis_armcc.h* - kompilator *RealView*
- *cmsis_gcc.h* - kompilator *GNU GCC*

44 gdzie *x* to numer rdzenia

45 lub inną wersja, jeśli nie korzystasz z kompilatora gcc

- *cmsis_css.h* - kompilator *TI CSS*
- ...

Pliki *core_cmFunc.h* oraz *core_cmInstr.h* dalej są obecne w paczce. Zawierają one jedynie kilka warunków preprocesora. Ich celem jest m.in. „zainkludowanie” odpowiedniego pliku z funkcjami, zależnie od używanego kompilatora. Polecam sobie poprzednią przeglądać.

Osobiście nie lubię jak mi się łączą w projekcie pliki, o wątpliwej przydatności. Korzystam tylko z kompilatora *gcc*, więc interesuje mnie tylko plik przygotowany dla tegoż kompilatora - *cmsis_gcc.h*. Jest tylko małe „ale”. Plik z definicjami dla rdzenia (*core_cmx.h*) odwołuje się do plików *core_cmFunc.h* oraz *core_cmInstr.h* („inkluduje” je) - jeżeli nie będzie ich w projekcie to pojawi się błąd. Rozwiązań są dwa:

- można jednak skopiować te pliki do projektu
 - można zamienić odwołania do nich, na odwołanie bezpośrednio do *cmsis_gcc.h*
-

Tyle w kwestii bibliotek.

Co warto zapamiętać z tego rozdziału?

- korzystanie z bibliotek firmowych nie jest konieczne – **wszystko** da się zrobić „na rejestrach”
- w chwilach kryzysu można podejrzeć jak dany peryferial jest konfigurowany w funkcji bibliotecznej
- w tym poradniku nie będziemy korzystać z bibliotek ST
- na start należy zaopatrzyć się w pliki:
 - *core_cmx.h*
 - *cmsis_gcc.h*
 - oraz plik nagłówkowy mikrokontrolera *stm32fxx.h*

2.7. Wskazówki przed startowe („*Ave, Caesar, morituri te salutant*”⁴⁶)

46 „*Witaj, Cezarze, idący na śmierć cię pozdrawiają.*”

Uff. To już ostatni rozdział wstępu. Ale ważny niebywale! Kilka rzeczy które warto wiedzieć/zrobić, zanim pójdziemy do ogródka witać się z gąską, a które wcześniej nigdzie mi nie pasowały.

Primum! Wspominałem o CMSIS i ujednoliconym sposobie dostępu do rejestrów konfiguracyjnych. Jak to wygląda w praktyce? W AVR stosowana była taka konwencja (przypominam, że określenie „AVR” w Poradniku, odnosi się do popularnych układów z rodzin ATmega i ATtiny):

```
REJESTR (operator) 1<<BIT;
```

na przykład jeśli chcielibyśmy włączyć przetwornik ADC to będzie to mniej więcej coś takiego⁴⁷:

```
ADCSRA |= 1<<ADEN;
```

W STM konwencja jest inna (standard CMSIS). Przede wszystkim jest o wiele więcej peryferiów, które mają jeszcze więcej rejestrów konfiguracyjnych, a rejesty więcej bitów. Zaprowadzono więc porządek i rejesty konfiguracyjne pogrupowano w struktury. Dodatkowo darowano sobie to przesuwanie bitowe i definicje bitów konfiguracyjnych zawierają od razu „jedynkę przesuniętą na odpowiednią pozycję” (czyli maskę bitową a nie numer bitu jak w nagłówkach AVR). W praktyce wygląda to np. tak: ustawienie bitu *CEN* w rejestrze konfiguracyjnym *CR1* licznika *TIM3*:

```
TIM3->CR1 |= TIM_CR1_CEN;  
BL0K_PERYFERYJNY -> REJESTR (operator) BL0K_REJESTR_BIT;
```

Proste prawda? To magiczne „->” to zwyczajny operator języka C (dostęp do składnika struktury poprzez wskaźnik). W Poradniku, chcąc odnieść się do rejestru *CR1* licznika *TIM3*, będę stosował zapis:

```
TIM3_CR1
```

lub, jeśli będę miał na myśli rejestr *CR1* jakiegokolwiek licznika (nie koniecznie trzeciego, jakiegoś licznika iks):

```
TIMx_CR1
```

Tzn. takie ambitne założenia ongiś poczyniłem. W praktyce pewnie będę skracał zapis do *TIM_CR1* lub samego *CR1* jeśli z kontekstu będzie jasno wynikało o jaki peryferial chodzi.

⁴⁷ pomijam tu kwestię czy powinno to być przypisanie (=) czy suma bitowa (|=) bo nie o to się rozchodzi

Nazwa bitu składa się z: nazwy bloku peryferyjnego, nazwy rejestrów i właściwej nazwy bitu. Przykładowo *TIM_CRI_CEN*:

- chodzi o bit licznika (*TIM*)
- bit jest w rejestrze *CRI*
- bit nazywa się *CEN*

Proszę zwrócić uwagę, że w nazwie bitu nie podaje się numeru układu peryferyjnego (*TIM1*, *TIM2*, ...). To dlatego, że najczęściej definicje bitów wyglądają identycznie dla wszystkich układów danego rodzaju (np. liczników). Czyli bez sensu byłoby tworzyć osobne definicje:

- *TIM1_CRI_CEN*;
- *TIM2_CRI_CEN*;
- ...

gdyż wszystkie liczniki mają identyczne rejesty konfiguracyjne *CRI* (prawie, ale do wszystkiego dojdziemy w swoim czasie).

Przy nazwie rejestrów (*CRI*) numer jest zawsze. Każdy licznik ma dwa rejesty konfiguracyjne *CR_* (*CRI* i *CR2*) i są to zupełnie inne rejesty, zawierające różne bity konfiguracyjne. Proszę sobie to dobrze przemyśleć i zrozumieć :)

Jeśli w rejestrze konfiguracyjnym znajduje się pole składające się kilku bitów, to do nazwy bitu dodawany jest sufiks z jego numerem. Przykład: w rejestrze *TIMx_CCMR2* (blok licznika, rejestr *CCMR2*) znajduje się pole *OCIM*, które składa się z trzech bitów. Ich nazwy to:

- ***TIM_CCMR2_OC1M_0***
- ***TIM_CCMR2_OC1M_1***
- ***TIM_CCMR2_OC1M_2***

Odrobinę inaczej sprawa wygląda w przypadku definicji nazw bitów rejestrów rdzenia. ARM w plikach nagłówkowych umieścił osobno definicje masek (sufiks *_Msk*) i numerów (pozycji) bitów w słowie (sufiks *_Pos*).

Od przedstawionych reguł zdarzają się sporadycznie wyjątki. Omówimy je sobie jak się nadziejemy na któryś z nich. Zdaję sobie sprawę, że to wygląda makabrycznie, ale to tylko pozory. Po paru godzinach zabawy z STMami nazewnictwo bitów i rejestrów umiejscawia się w małym

palcu. To po prostu wygląda skomplikowanie, gdy próbuje się to opisać słownie :) Jak jazda na rowerze.

Żeby nie było: żadnej magii w tym nie ma od strony technicznej. To tylko język C. W pliku nagłówkowym, dla każdego peryferiala, stworzona jest struktura której pola odpowiadają kolejnym rejestrów konfiguracyjnym związanym z tym blokiem. Potem wystarczy wziąć wskaźnik na taką strukturę i ustawić go tak, aby wskazywał obszar w pamięci mikrokontrolera, gdzie znajdują się rejesty konfiguracyjne tego układu. Właśnie to zawiera plik nagłówkowy mikrokontrolera. W programie zapisujemy coś do pól takiej struktury, a to co zapiszemy ląduje pod adresem układu peryferyjnego. Ale to tak OT.

Secundo! W STM32 wszystko jest po resecie „wyłączone”. O co mi chodzi? W AVR jeśli chcieliśmy np. mrugać diodą to wystarczyło skonfigurować port i cyklicznie zmieniać jego stan. W STMach trzeba taki port **najpierw** „włączyć”, bo po resecie wszystkie peryferia mikrokontrolera są „wyłączone”. Dokładniej rzecz ujmując, wyłączone jest taktowanie bloków peryferyjnych aby ograniczyć pobór prądu. Psze się nie bać – chodzi o ustawienie raptem jednego bitu... ale gwarantuję, że na początku będziesz o tym nagminnie zapominać. A co wtedy? A wtedy:

- układ peryferyjny ogólnie nie będzie działał
- próba zapisania czegokolwiek do rejestrów tego peryferiala się nie powiedzie – np. ustawimy jakieś bity w rejestrze, a po odczytaniu⁴⁸ będą tam same zera (to jest dosyć wyraźna wskazówka, że blok nie jest taktowany)

Tertio! Znaczna część rejestrów konfiguracyjnych STMa ma pewną... *trap for young players*. Pułapka wynika z tego, że po resecie nie mają one wartości zero. Trzeba więc uważać na początkowy stan rejestrów i w razie potrzeby wyzerować sobie niepotrzebne bity, a nie stosować „bezpieczne przypisanie”⁴⁹ (sumę bitową) na pałę. Notabene w AVR też chyba część rejestrów miała domyślną wartość różną od zera – nie chce mi się szukać...

Należy również zwrócić uwagę na bity oznaczone jako *Reserved* (zastrzeżony). Zapisując coś do rejestrów zawierającego pola zastrzeżone należy zadbać o to, aby wpisywana wartość zawierała domyślne (początkowe) stany zastrzeżonych bitów. Np. jeśli pole *Reserved* ma domyślną

⁴⁸ np. debuggerem

⁴⁹ idiotyczny termin

wartość zero, to w nowej wartości wpisywanej do rejestru też musi być wyzerowane. Cytując RM: „*must be kept at reset value*”.

Quarto! Mikrokontrolery STM32 nie mają *fuse bitów* którymi można je „zablokować”. W ogóle ciężko je popsuć programowo. Owszem o systemie zegarowym tych mikrokontrolerów krążą w Internecie „legendy”. Ale prawda jest taka, że zegarów prawie wcale nie trzeba ruszać na początku. A jak już ktoś chce się pobawić to może zrobić co mu się żywnie podoba a procka nie zablokuje. STMy są żywotne do tego stopnia, że przestawione na zewnętrzny kwarc same mogą zmienić źródło taktowania na wewnętrzny oscylator jeśli kwarc odmówi posługi. Jedyny znany mi sposób zablokowania STMa na amen, to włączenie drugiego stopnia ochrony pamięci przed odczytem w STM32F334 lub STM32F429. Nie da się tego zrobić „przypadkiem” w programie. Mikrokontrolera STM32F103 nie da się w ogóle zablokować. Innych układów nie znam, więc się nie wymądrzam :)

Quinto! Proszę odrzucić wszelkie bariery psychologiczne. STMy są proste. Trudne to jest pisanie na AVR jak się pozna STM. Bo ciągle czegoś brakuje, kupę czasu zabiera zabawa w „jak tu coś podzielić żeby nie dzielić” albo trzeba się kopać z koniem żeby zapisać stałe w pamięci Flash i je potem odczytać... no i te *fuse bity* :)

Sexto! W Poradniku koncentruję się na mikrokontrolerach, które są w posiadanych przeze mnie zestawach rozwojowych. Dla skrócenia zapisu, w Poradniku będę używał nazw:

- **F103** w odniesieniu do mikrokontrolera *STM32F103VCT6* z zestawu *HY-mini* (z odłączonym wyświetlaczem)
- **F334** w odniesieniu do mikrokontrolera *STM32F334R8T6* z zestawu *Nucleo-F334R8*
- **F429** w odniesieniu do mikrokontrolera *STM32F429ZIT6* z zestawu *STM32F429i-Disco*

Wszelkie przykładowe kody będą przystosowane do uruchomienia na wspomnianych płytach.

Septimo! Na Boga! Nie próbuj uczyć się na pamięć rejestrów konfiguracyjnych, szczegółów działania różnych trybów czy cokolwiek w tym guście. I tak nie zapamiętasz wszystkiego. Najważniejsze to pi razy drzwi wiedzieć jakie tryby pracy układów peryferyjnych są dostępne i jak z grubsza działają. Potem mając konkretny problem do rozwiązania dopasujesz sobie szczegóły i doczytasz konkretы w dokumentacji.

Octava! Poszczególne rozdziały Poradnika, z założenia, nie są autonomiczne. Szczególnie początkowe zawierają znaczną ilość informacji nadprogramowych, nie do końca wynikających z głównego tematu rozdziału. Ponadto, pomimo podziału na rozdziały dotyczące poszczególnych mikrokontrolerów (F103, F334, F429), informacje dotyczące różnych rodzin czasem się przeplatają. Zdecydowanie należy przeczytać wszystkie, nawet jeśli nie zamierzasz korzystać np. z F429. Jeżeli przy tytule rozdziału nie ma wyszczególnionej konkretnej rodziny mikrokontrolerów, to dotyczy on wszystkich omawianych rodzin. Od razu uprzedzam też, że na początku nie wszystko da się zrozumieć przy pierwszym czytaniu. Układy mikrokontrolera często są od siebie zależne. Dlatego też w kilku miejscach nie udało się uniknąć sytuacji, gdzie do zrozumienia aktualnego zagadnienia przydatna jest wiedza z późniejszych rozdziałów. Szczególnie, że starałem się, aby opisy peryferiów były możliwie kompletne. To znaczy żeby w każdym rozdziale było wszystko, co uważam za warte uwzględnienia, bez względu na to czy jest to wiedza „podstawowa” czy jakiś „smaczek”. Generalnie zmierzam do tego, że na początku nie wszystko wyda się proste i warte uwagi, ale proszę się nie zniechęcać :) Dla ułatwienia po większości rozdziałów, jak już pewnie zauważyłeś, umieszczam skróconą listę najważniejszych zagadnień, które powinieneś opanować zanim pojedziesz dalej.

Co warto zapamiętać z tego rozdziału?

- z tego to akurat wszystko

3. PORTY I/O („ARDUA PRIMA VIA EST”⁵⁰)

3.1. Ogarnąć nóżki w F103 (F103)

Każdy szanujący się poradnik zawiera mikrokontrolerowe *Hello World*, czyli migającego leda. Wszyscy mają... mam i ja. W AVRach z portami wejścia/wyjścia związane były trzy rejestrów (DDR - kierunek portu, PORT - stan wyjściowy portu, PIN – odczyt stanu wejściowego portu). Za pomocą których można było skonfigurować pin w jednym z czterech trybów:

Tabela 3.1 Konfiguracja portu AVR

DDR	PORT	tryb pracy
0	0	wejście, płyniące
0	1	wejście, podciągnięcie do VCC
1	0	wyjście, stan niski
1	1	wyjście, stan wysoki

W STIMach trybów i rejestrów jest więcej, ale nie ma co się przerażać – od przybytku głowa nie boli – nawet jeśli jakiś tryb wydaje się naciągany to przecież obowiązku korzystania z niego nie ma. Może kiedyś się przyda.

Porty wejścia/wyjścia (GPIO⁵¹) są układem peryferyjnym mikrokontrolera, więc ich opis znajduje się w *Reference Manualu*. Dane elektryczne są w *datasheetie* mikrokontrolera (*Electrical characteristics*, podrozdziały: *Absolute maximum ratings* oraz *Operating conditions → I/O port characteristics*). Weźmy sobie dane z konkretnego RMa. Tak jak wspominałem, punktem wyjścia będzie *STM32F103*, więc korzystam z *RM0008*. Z GPIO związane są następujące rejestrze:

- CRL – *Configuration Register Low*
- CRH – *Configuration Register High*
- IDR – *Input Data Register*
- ODR – *Output Data Register*
- BSRR – *Bit Set/Reset Register*
- BRR – *Bit Reset Register*
- LCKR – *Lock Configuration Register*

50 „Początki są trudne.”

51 General Purpose Inputs / Outputs - czyli nóżki

Pierwsza dwa rejesty służą do konfiguracji trybu pracy nóżki mikrokontrolera. Każda nóżka konfigurowana jest za pomocą czterech bitów (dwa bity *GPIO_CRL/H_MODE* i dwa bity *GPIO_CRL/H_CNF*). Port STMa to 16 nóżek – jak łatwo policzyć konfiguracja całego portu to w sumie 64 bity konfiguracyjne (16 nóżek * 4 bity), czyli nie zmieści się toto w jednym rejestrze 32bitowym. Dlatego też są dwa rejesty CR (dolny i górny). Rejestr dolny (CRL) obejmuje konfigurację pinów od 0 do 7 (*PA0, PA1...PA7; PB0, PB1...PB7, PC0, PC1... itd.*). Nóżki o numerach 8 do 15 konfigurowane są w rejestrze górnym (CRH). W RM jest to zaznaczone w następujący sposób⁵²: w rozdziale *GPIO Registers*, przy opisie bitów *CNF* rejestrzu *CRL* jest taki zapis:

CNFy[1:0]: Port x configuration bits (y = 0 .. 7)

co oznacza mniej więcej to że: dwa bity CNFy (bit 0 i 1) to bity konfiguracyjne portu „x”⁵³, a „y” jest z zakresu 0...7. Przy rejestrze *CRH* „y” ma zakres 8..15.

Rejestr wejściowy (IDR) to rejestr tylko do odczytu (zwróć uwagę na oznaczenie 'r' w tabelce przy opisie rejestrów⁵⁴) służący do odczytu stanów wejść – taki PINx w AVRach. Z kolei analogią do AVRowego rejestrów (PORTx) jest rejestr ODR – rejestr „wyjściowy”. Pozostałe trzy rejesty na razie zostawmy w spokoju. Przejedźmy do trybów pracy GPIO:

Tabela 3.2 Konfiguracja portu F103

Konfiguracja	CNF	MODE	ODR
wyjście	<i>push-pull</i>	00	0 lub 1
	<i>open-drain</i>	01	
funkcja alternatywna	<i>push-pull</i>	10	01 - 2MHz 01 - 10MHz 11 - 50MHz
	<i>open-drain</i>	11	
wejście	<i>pływające</i>	01	bez znacz.
	<i>podciągnięte w dół</i>	10	
	<i>podciągnięte w góre</i>	10	
konfiguracja analogowa	00	00	bez znacz.

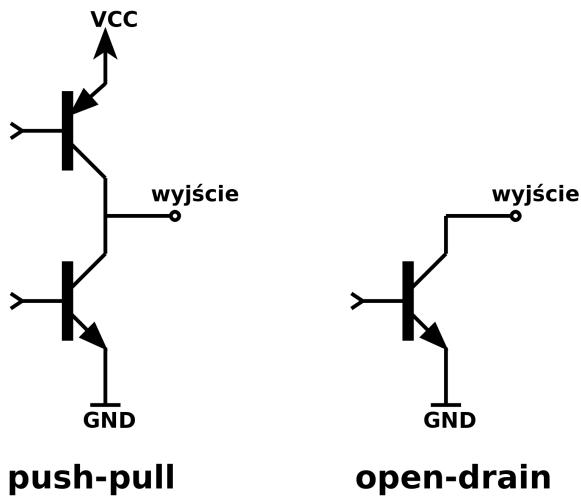
Wyjście – nowości w stosunku do AVRów są dwie. Po pierwsze wyjście może pracować w dwóch trybach: *push-pull* lub *open-drain*. Wyjście *push-pull* to dwa tranzystory (patrz rys. 3.1).

52 warto zwracać uwagę na różne takie niuanse żeby potem nie błędzić za długo :)

53 x to „oznaczenie” portu – np. dla portu A chodzi o to „A”, dla portu B o „B” itd...

54 spis wszystkich oznaczeń stosowanych w RM znajduje się na początku dokumentu, w rozdziale *Documentation conventions*

Górny wymusza stan wysoki wyjścia (napięcie bliskie V_{CC}), dolny odpowiada za stan niski (praktycznie GND). W ten sposób działały wyjścia w AVR. W przypadku wyjścia *open-drain* nie ma górnego tranzystora. Wyjście w stanie wysokim „wisi”, zaś w stanie niskim jest ściagnięte do masy przez dolny tranzystor. Po co tak? Np. do sterowania jakimś układem o innym napięciu niż mikrokontroler⁵⁵ lub gdy trzeba połączyć kilka wyjść różnych układów naraz... Zastosowania wyjść open-drain to nie temat tego Poradnika. Za stan wyjścia (niski / wysoki) odpowiada wartość rejestru ODR (AVRowy PORTx).



Rys. 3.1 Wyjścia push-pull i open-drain⁵⁶ (*paintCAD*)

Druga nowość to te nieszczęsne „prędkości” ustawiane za pomocą bitów *MODE* (patrz tabela 3.2). Wbrew temu co można wyczytać tu i tam w Internecie, te ustawienia nie odnoszą się bezpośrednio do częstotliwości z jaką będzie przełączał się pin czy cokolwiek w tym guście! Ustawienie „prędkości”⁵⁷ wpływa na **stromość zboczy sygnału wyjściowego** z pinu. Dokładniej zmienia „siłę” z jaką tranzystory wyjściowe ciągną wyjście w górę lub w dół. Im mniejsza ta siła, tym łagodniejsze zbocza sygnału bo wolniej ładują się pojemności obciążające wyjście. Pośrednio wpływa to na maksymalną częstotliwość sygnału prostokątnego wychodzącego z pinu. Ale nie chodzi o to, że po ustawieniu małej „prędkości” zmniejszy się częstotliwość pracy portu czy też generowanego sygnału wyjściowego. Nie! Chodzi o to, że bufory wyjściowe będą pracowały z mniejszą „siłą”, przez co zbocza prostokąta nie będą „wystarczająco” strome i wyjdzie z tego taki zaokrąglony prostokąt. Te wartości przy opisie bitów *MODE* (2MHz, 10MHz, 50MHz) to wartości granicznej częstotliwości sygnału prostokątnego, przy której (w określonych w *datasheet* warunkach obciążenia) sygnał spełnia (opisane w *datasheet*) wymagania stawiane sygnałowi prostokątnemu. Szczegóły elektryczne, czasowe, wykresy... do doczytania w *datasheet*.

⁵⁵ oczywiście w pewnych granicach – patrz *datasheet*

⁵⁶ w rzeczywistości są to tranzystory polowe, na rysunku użyto symboli bipolarnych dla zwiększenia czytelności

⁵⁷ strasznie mylące to określenie

kontrolera⁵⁸. Po co to? Zmniejszanie stromości zboczy redukuje zakłócenia indukowane generowane... elektromagn.... dobra, jeden pies – jakieś zakłócenia redukuje. I zmniejsza zużycie energii.

Funkcja alternatywna – tu jest prosta sprawa. Tryb „alternatywny” jest wykorzystywany jeśli danym **cyfrowym wyjściem** ma sterować jakiś peryferial mikrokontrolera. Np. jeśli ma to być wyjście PWM generowanego przez TIMER albo UARTowe Tx/D czy jakiekolwiek inne **wyjście cyfrowe** związane z peryferialem. Do wyboru są dwa tryby pracy (*push-pull* oraz *open-drain*) i „prędkość” tak samo jak w przypadku normalnego wyjścia. Szczegółowe informacje, jak należy skonfigurować nóżki współpracujące z różnymi peryferiami znajdują się w RM, w rozdziale *GPIO configurations for device peripherals*. W trybie alternatywnym, wartości wpisywane do rejestru wyjściowego ODR nie mają wpływu na zachowanie pinu.

Wejście – bardzo podobnie jak w AVR. Wejście może być „pływające” (jak w AVR, stan wysokiej impedancji) i to jest domyślny stan po resetie⁵⁹. Możemy je również podciągnąć wewnętrzny rezystorem w górę lub w dół (to jest nowość w porównaniu z AVR). Stan wejścia cyfrowego możemy odczytać z rejestru wejściowego IDR.

Konfiguracja analogowa – zupełnie nowość w stosunku do AVR. Tryb ten służy do współpracy z przetwornikami ADC i DAC. Nie możemy odczytywać „cyfrowego” stanu wejścia z rejestru IDR (będzie zwracać zawsze 0) ani sterować nóżką za pomocą rejestru ODR.

W RM tryb analogowy jest zaliczony do „wejść”. Uważam, że jest to mylące bo ten tryb służy również do obsługi „wyjść” analogowych (przetwornik DAC), więc w tabelce umieściłem ten tryb jako osobną konfigurację, a nie jako jedną z opcji wejściowych.

Uwaga pułapka! Założmy, że chcemy ustawić tryb alternatywny *push-pull*. Zgodnie z tabelką musimy ustawić bity CNF na 0b10. No więc, aby ustawić ten pierwszy bit, piszemy coś w stylu (paskudny pseudo-kod, ale myślę że wiadomo o co chodzi):

```
CNF |= 0b10;
```

I... ani be, ani me, ani tere fere - klapa, nie działa :) Wiesz dlaczego? Pisałem już o tym ooo tutaj: Tertio!. Rejestry GPIOx_CRL/H (tam siedzą bity CNF) domyślnie po resetie są ustawione tak, aby

58 rozdział *Input/Output AC characteristics*

59 wyjątkiem są m.in. piny JTAGa: PA15 (pull-up), PA14 (pull-down), PA13 (pull-up), PB4 (pull-up)

wybrany był tryb wejściowy pływający. Czyli zgodnie z tabelką 3.2 bity CNF mają wartość 0b01. Po naszej operacji ($\text{CNF} \mid= 0b10$) wyszło więc: $0b01 \mid 0b10 = 0b11$. Nie o to nam chodziło.

Zdaję sobie sprawę, że powyższe wydaje się strasznie skomplikowane, ale nie ma co się martwić na zapas. Nikt przecież nie będzie się uczył tych bitów konfiguracyjnych na pamięć ;)

Przy konfigurowaniu pinów warto zwrócić uwagę na to, aby nie zmienić konfiguracji (domyślnej) wyprowadzeń związanych z wykorzystywanym interfejsem programowania/debugowania (np. JTAG albo SWD). Wgranie programu, który zmienia konfigurację tych wyprowadzeń może uniemożliwić połączenie się z mikrokontrolerem. Bez paniki! Nic się w ten sposób nie zablokuje. W razie czego wystarczy przytrzymać *reset* podczas łączenia z mikrokontrolerem albo uruchomić go w trybie bootloadera (więcej na ten temat w rozdziale 16.2). Uprzedzam, żeby potem nie było niespodzianki :)

Co warto zapamiętać z tego rozdziału?

- nóżka może być:
 - wejściem (pływającym, podciagniętym do góry lub do dołu)
 - wyjściem (push-pull lub open-drain)
 - wejściem/wyjściem analogowym
- trzeba uważać na początkowe wartości rejestrów!
- „prędkość” wyjścia wpływa na siłę działania buforów wyjściowych (im mniejsza tym łagodniejsze zbocza i mniej zakłóceń)

3.2. Nieśmiertelnie Blinkający Hell of World (F103)

Koniec teoretyzowania. Zeit für die Praxis⁶⁰. No to lecimy na głęboką wodę. Tzn. Ty lecisz :P Czytelnika proszę o zasięcie, stworzenie nowego *maina* w Twoim ulubionym środowisku i napisanie kodu do migania diodą :) Pomocy szukamy w *Reference Manualu* (dokumentacji peryferiów mikrokontrolera) w rozdziale dotyczącym GPIO i RCC⁶¹ (przypominam: Secundo!). Proszę się nie oburzać, nie uśmiechać pod nosem, nie zniechęcać tylko sumiennie poczytać dokumentację (tylko te dwa rozdziały) i spróbować coś napisać – proste miganie diody. Nie obiecuję, że się uda... Wrócić do poradnika można jak dioda będzie migać lub miną przynajmniej trzy dni i migania się nie uświadeczy. Mówię całkiem serio! Mnie rozpracowanie podstaw środowiska i napisanie pierwszego działającego *blink led*a zajęło coś koło 5-dni mimo że wcześniej

60 z niem. *Czas na praktykę...* a przynajmniej taką mam nadzieję :)

61 *Reset and Clock Control* – czyt. resety i zegary

czytałem o STMach przez kilka tygodni. Możesz oczywiście skopiować mój kod (albo jakiś gotowiec z neta), odpalić, przeanalizować i stwierdzić że banał i sam dałbyś radę... tylko w takim razie czemu sam go nie napisał? Pięć dni poznawałem dokumentację, środowisko i uczyłem się jak szukać pomocy w Internecie – nie odbieraj sobie tego, bo to jest bezcenna wiedza której nie da się przekazać żadnym poradnikiem.

Zadanie domowe 3.1: Napisz program migający diodą. Czas start!

Przykładowe rozwiązanie (F103, dioda LED2 podłączona do PB1):

```
1. #include "stm32f10x.h"
2.
3. int main(void){
4.
5.     RCC->APB2ENR = RCC_APB2ENR_IOPBEN;
6.
7.     GPIOB->CRL |= GPIO_CRL_MODE1_1;
8.     GPIOB->CRL &= ~GPIO_CRL_CNF1_0;
9.
10.    volatile uint32_t delay;
11.
12.    while(1){
13.
14.        GPIOB->ODR |= GPIO_ODR_ODR1;
15.        for(delay = 1000000; delay; delay--) {};
16.
17.        GPIOB->ODR &= ~GPIO_ODR_ODR1;
18.        for(delay = 1000000; delay; delay--) {};
19.
20.    } /* while(1) */
21.
22. } /* main */
```

To jest CAŁY kod. Przed tym *mainem* jest tylko typowa „rozbiegówka”, taka sama jak w AVR (przede wszystkim inicjalizacja zmiennych) mikrokontrolera. W wielu miejscach w Internecie można wyczytać, że aby w ogóle STM działał należy konfigurować zegary dla całego mikrokontrolera, wywoływać jakieś magiczne *SystemInit()*, konfigurować pętle PLL i inne cuda wianki na kiju po wodzie. Bzdura totalna. Nie ma takiej potrzeby – można nic nie ruszać i wszystko będzie działać na wbudowanym oscylatorze (HSI⁶²) z częstotliwością 8MHz. Na zabawę zegarami przyjdzie czas.

Prześledźmy to arcydzieło. Na początku załączany jest plik nagłówkowy z definicjami nazw rejestrów i bitów (takie AVRowe „io.h”). W kolejnych kodach będę ucinał tą linijkę żeby nie wydłużać niepotrzebnie listingów. Tak jak wspominałem (Secundo!), każdy z układów peryferyjnych mikrokontrolera STM32 trzeba włączyć przed użyciem. W tym przykładzie korzystam z portu B (GPIOB). Należy sprawdzić do jakiej szyny jest podłączony ten port i go włączyć w odpowiednim rejestrze. Do wyboru mamy trzy szyny APB1, APB2 lub AHB (na razie

62 High Speed Internal, wszystko się wyjaśni w rozdziale 17

przyjmuj na wiare). Miejsc gdzie można sprawdzić, do której szyny jest podłączony port, jest kilka. IMHO najwygodniejsze to:

- rozdział *Memory map* w RM – odnaleźć port w tabeli *Register Boundary Addresses* i sprawdzić do jakiej szyny jest podpięty (kolumna *Bus*)
- wyszukać (w RM) bit włączający układ w opisie rejestrów włączających sygnał zegarowy (blok RCC, rejesty RCC_AHBENR, RCC_APB1ENR, RCC_APB2ENR), np. bit włączający GPIOB nazywa się IOPBEN i leży w rejestrze RCC_APB2ENR

Każda szyna ma swój rejestr włączający sygnał zegarowy peryferiów do niej podłączonych:

- szyna AHB - rejestr RCC_AHBENR
- szyna APB1 - rejestr RCC_APB1ENR
- szyna APB2 - rejestr RCC_APB2ENR

Port B podpięty jest do szyny APB2, włączenie zegara następuje w linijce 5 kodu. Zapis z tej linijki to dokładnie: przypisanie wartości ukrytej pod definicją bitu *RCC_APB2ENR_IOPBEN* do rejestru *RCC_APB2ENR*. Pod definicją bitu kryje się maska bitu włączającego zegar dla portu B (bit IOPBEN). Polecam pooglądać sobie co się kryje pod RCC_APB2ENR i RCC_APB2ENR_IOPBEN i skonfrontować z dokumentacją.

Linie 7 i 8 to konfiguracja wyjścia – ustawiany jest tryb wyjściowy, *push-pull*, *2MHz*. Zwracam uwagę na zerowanie jednego z bitów CNF, który początkowo (po resetie procka) był ustawiony (Tertio!).

W pętli głównej, z wykorzystaniem rejestrów GPIO_ODR, cyklicznie zerowany i ustawiany jest jeden z bitów rejestrów (linie 14 i 17 kodu) - ODR1 (czyli nóżka PB1). Reszta kodu nie ma nic wspólnego z STM i nie powinna budzić wątpliwości. No i jak? Jest w tym coś trudnego?

Proponuję ten prosty przykład wykorzystać dydaktycznie do granic możliwości – np. odpalić na nim debugger i sprawdzić jak działają komendy typu *step / pause / resume*, podglądanie wartości rejestrów konfiguracyjnych itd. Nie wnikam w to, bo to nie poradnik używania środowiska. I jeszcze taka uwaga (będę truł!) – mnie dojście do tego etapu (ze zrozumieniem) zajęło kilka dni. Kilka dni oswajania się ze środowiskiem, wertowania dokumentacji itd. itd. - uczenia się. Jeśli, Szanowny Czytelniku, pójdziesz na łatwiznę, skopiujesz ten mój kod i tyle, to do kitu taka zabawa. Zatrzymaj się tutaj, pobaw środowiskiem, debuggerem, pozmień coś w kodzie, obejrzyj wygenerowany kod assemblera, jeśli masz jakiekolwiek wątpliwości, czegoś nie rozumiesz - RTFM i STFW.

W każdym projekcie mikrokontrolerowym wykorzystuje się porty GPIO. Ustawianie za każdym razem, każdego z pinów, z wykorzystaniem rejestrów może okazać się odrobinę upierdliwe. Warto napisać sobie prostą funkcję ułatwiającą to zadanie albo znaleźć jakąś, która nam pasuje⁶³. Ze swojej strony polecam funkcje udostępnione przez *Freddiego Chopina* w przykładach na jego stronie⁶⁴. Swego czasu próbowałem napisać coś po swojemu, jednak z każdą poprawką mojej funkcji coraz bardziej zbliżałem się do wersji *Freddiego*⁶⁵ :) Tak czy siak w przykładach będę korzystał z funkcji *Freddiego* albo bardzo zbliżonej. Nie ma sensu jej teraz omawiać, kod źródłowy pokazany jest w dodatku 1. Przykład użycia funkcji w celu ustawienia PB0 jako wyjścia *push-pull*, *2MHz*:

```
gpio_pin_cfg(GPIOB, PB0, gpio_mode_output_PP_2MHz);
```

Prawda, że logiczne i wygodne?

Wróćmy do rejestrów związanych z portem – zostały jeszcze 3 do omówienia:

- BSRR – Bit Set/Reset Register
- BRR – Bit Reset Register
- LCKR – Lock Configuration Register

Zacznijmy od LCKR. Rejestr ten umożliwia zablokowanie możliwości zmian konfiguracji wybranych pinów portu, aż do następnego resetu mikrokontrolera. Celem jest uniemożliwienie przypadkowej zmiany konfiguracji pinu np. na skutek błędu w programie. Aby zablokować możliwość zmian należy kilkukrotnie wykonać zapis do rejestru GPIO_LCKR określonej sekwencji wartości – odsyłam do RM.

Zadanie domowe 3.2: sprawdzić empirycznie, czy tak zablokowaną konfigurację pinu da się zmienić korzystając z debuggera.

Pozostałe dwa rejesty (BSRR i BRR) umożliwiają wykonywanie atomowych operacji na portach. I zasługują na osobny rozdział :)

Co warto zapamiętać z tego rozdziału?

- przed użyciem jakiegokolwiek układu peryferyjnego należy włączyć jego taktowanie w odpowiednim rejestrze!

63 błagam, tylko nie potwora z biblioteki SPL

64 www.freddiechopin.info

65 wszystkie drogi prowadzą do *Freddiego*

- do konfiguracji pinów będziemy wykorzystywać funkcję z dodatku 1
- konfigurację pinu można zablokować

3.3. Atomowo macham nogą i nie tylko

Na początek krótki wstęp o atomowości. Określenie „operacja atomowa” oznacza operacje niepodzielne. Założmy, że chcemy ustawić trzeci bit w rejestrze *foobar* nie zmieniając stanu pozostałych bitów (pseudo-kod):

```
foobar |= (1<<3);
```

powyższe oczywiście działa, ale jest mały haczyk. Jeśli podejrzymy wygenerowany kod asm to dostaniemy mniej więcej coś takiego (poniższe jest pisane z głowy, więc proszę się nie czepiać uproszczeń, chodzi o ideę) :

ldr r3, &foobar	załadow do r3 to co jest pod adresem foobar
orr r3, 0b1000	suma logiczna wartości z rejestr r3 i stałej 0b1000
str &foobar, r3	wrzucenie zawartości r3 pod adres rejestru foobar

Ta operacja **nie jest** atomowa - składa się z trzech operacji składowych. Powyższa kombinacja określana jest jako **RMW** (*Read - Modify - Write*), gdyż składa się z :

- odczytania zawartości rejestr peryferiala (lub pamięci) do „zmiennej pomocniczej” (rejestru ogólnego procesora - np. r3)
- operacji arytmetyczno-logicznej na tej zmiennej pomocniczej
- zapisania wartości zmiennej pomocniczej nazad do rejestr peryferiala (lub pamięci)

Dlaczego tak dziwnie? Dlatego, że operacje arytmetyczno-logiczne mogą być wykonywane tylko na rejestrach ogólnych procesora⁶⁶. Problem z nieatomowymi operacjami jest taki, że coś może się w nie „wcisnąć”.

Proponuję się skupić, żeby załapać przykład :) Założmy, że w funkcji *main* mamy takie właśnie nieatomowe ustawienie pierwszego bitu rejestr:

```
GPIOB->ODR |= GPIO_ODR_ODR1;
```

ponadto w jakimś przerwaniu jest kod który również modyfikuje ten rejestr, np. tak:

⁶⁶ architektura RISC tak ma

```
GPIOB->ODR = 0;
```

Problem pojawi się, jeśli przerwanie wystąpi w trakcie wykonywania (nieatomowej) operacji RMW w funkcji *main*. Założmy, że na początku rejestr ODR ma wartość 1 (ustawiony tylko zerowy bit). W programie dochodzimy do nieatomowego ustawiania pierwszego bitu. Wartość rejestru (czyli 1) jest odczytywana do „zmiennej pomocniczej”. Potem jest sumowana z 0b10. Po tej operacji wartość zmiennej pomocniczej wynosi 0b11 i... tu występuje przerwanie, które zeruje rejestr (ODR = 0). Przerwanie się kończy, wracamy do *main* i kończymy naszą nieatomową sekwencję RMW. Wartość zmiennej pomocniczej (0b11) jest wpisywana do rejestrów ODR (skasowanego przed chwilą w przerwaniu). W tym momencie następuje tragedia! Zerowy bit jest znowu ustawiony!

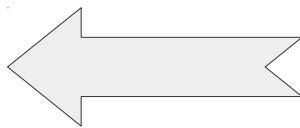
Zatrzymaj się tu i dobrze przemyśl sprawę: w *main* była tylko operacja ustawiająca pierwszy bit, w przerwaniu skasowaliśmy wszystkie bity. A na końcu bit zerowy znowu jest ustawiony! Zmiana dokonana przez przerwanie została „nadpisana”, bo w zmiennej pomocniczej (r3) była stara wartość rejestrów. Katastrofa!

Jak się przed nią uchronić? Można, brutalnie, wyłączyć przerwania na czas operacji RMW. W AVR był *atomic block*, czyli takie trochę bardziej eleganckie wyłączenie przerwań. STMy oferują kilka mechanizmów umożliwiających uzyskanie atomowości. Wyłączenie przerwań jest niezbyt finezyjną ostatecznością. W przypadku portów GPIO z dodatkową pomocą przychodzą nam rejesty GPIO_BSRR i GPIO_BRR. Zajmijmy się pierwszym z nich.

GPIOx_BSRR to 32 bitowy rejestr. Młodsze 16 bitów (Set Bits) odpowiada za ustawianie bitów w rejestrze GPIOx_ODR. Działa to tak, że wpisanie jedynki do któregoś z bitów dolnej połowy BSRR, powoduje ustawienie odpowiadającego mu bitu w rejestrze ODR. Przykładowo jeśli wpiszę jedynkę na pozycję bitu BS9 w rejestrze GPIO_BSRR to spowoduje to ustawienie⁶⁷ 9-go bitu rejestrów GPIO_ODR. Wpisanie zera nic nie powoduje, jest ignorowane.

Uwaga! Rejestr GPIO_BSRR (i GPIO_BRR) jest **tylko do zapisu** (write only). Ponadto **wpisanie zera do BSRR (lub BRR) nie ma żadnego skutku**. To jest ważne! Na rejestrze BSRR (i BRR) nie wykonuje się operacji odczytu ani tym bardziej RMW:

```
coś = BSRR;  
if (BSRR & coś) ...  
BSRR |= coś;  
BRR &= coś;
```



To jest be!

Powyższe operacje są błędne! To są rejesty **tylko do zapisu**, jedyne dopuszczalne działanie to:

⁶⁷ za pomocą jakiegoś sprzętowego *hokus-pokus*

```
BSRR = coś;  
BRR = coś;
```

Powyższe operacje są atomowe. Nie ma tu odczytu rejestru, potem operacji *OR/AND*, i na końcu zapisu. Jest tylko sam (atomowy) zapis wartości do rejestru. Połączenie między rejestrami BSRR (i BRR) a ODR jest zrealizowane sprzętowo i o to się nie musimy martwić :) Wpisujemy jedynkę (lub kilka jedynek) i nic nas więcej nie obchodzi. Niektórzy próbują potem taką jedynkę z BSRR lub BRR kasować albo odczytywać. Nein! To nie jest zwyczajny rejestr który „zachowuje” to co do niego zapiszemy. To taka studnia bez dna - wrzucamy jedynkę to coś się zadzieje (np. ustawi się jakiś bit rejestru ODR), ale wrzucona zawartość przepada, nie możemy jej odczytywać czy kasować.

Jeszcze jeden przykład. Ustawienie trzeciego bitu rejestru GPIO_ODR (bez zmiany pozostałych bitów) klasycznie i za pomocą rejestru BSRR:

```
GPIOx->ODR |= GPIO_ODR_ODR3;
```

```
GPIOx->BSRR = GPIO_BSRR_BS3;
```

Efekt jest ten sam. Obie linijki powodują ustawienie trzeciego bitu rejestru GPIO_ODR bez zmiany stanu pozostałych bitów. Różnica polega na tym, że pierwsza wersja nie jest atomowa a druga jak najbardziej. Tak dla pewności jeszcze dopiszę, że operacja:

```
GPIOx->ODR = GPIO_ODR_ODR3;
```

odpada bo takie przypisanie wykasuje wszystkie bity poza trzecim. A założenie było takie, że ustawiamy trzeci bit, ale reszty nie ruszamy.

Jak dotąd mówiłem o młodszych 16-bitach rejestru GPIO_BSRR – wrzucenie tam „1” powoduje ustawienie odpowiadającego bitu w GPIO_ODR. Wrzucenie „0” nie ma żadnego efektu. Drugie 16-bitów rejestru BSRR (bity od 16 do 31) działa tak samo, tylko że odwrotnie. Wpisanie jedynki powoduje kasowanie bitów rejestru GPIO_ODR (zamiast ustawiania). Wpisanie zera nic nie powoduje.

Został jeszcze jeden. Rejestr GPIO_BRR to 16-bitowy rejestr, który działa tak samo jak górna połowa rejestru BSRR. Czyli umożliwia atomowe kasowanie bitów rejestru ODR. Kasowanie za pomocą rejestrów BSRR i BRR różni się tylko położeniem bitów „kasujących” w rejestrze.

Rdzeń Cortex-M oferuje jeszcze jeden mechanizm umożliwiający wykonywanie atomowych operacji na bitach rejestrów (i nie tylko rejestrów): *bit-banding*. Ale to temat na osobny rozdział.

Zadanie domowe 3.3: sprawdzić empirycznie co się zadzieje jeśli spróbujemy jednocześnie ustawić i skasować ten sam bit za pomocą rejestrów GPIO_BSRR. Np. tak:

```
GPIOB->BSRR = GPIO_BSRR_BS7 | GPIO_BSRR_BR7;
```

Po sprawdzeniu empirycznym proszę, dla sportu, znaleźć potwierdzenie uzyskanej odpowiedzi w RMie. Miłych poszukiwań :)

Zadanie domowe 3.4: napisać program zapalający jedną z dwóch diod w zależności od stanu przycisku. Jeśli przycisk nie jest wciśnięty pali się pierwsza dioda. Jeśli jest wciśnięty pali się druga dioda. Jedeną diodę proszę gasić/zapalać nie-atomowo, drugą – atomowo. Oczywiście najpierw, minimum 3 dni, bawisz się sam. Dopiero potem można podglądać moje rozwiązanie!

Przykładowe rozwiązanie (F103, diody na PB0 i PB1, przycisk PB2):

```
1. int main(void){
2.     RCC->APB2ENR = RCC_APB2ENR_IOPBEN;
3.
4.     gpio_pin_cfg(GPIOB, PB0, gpio_mode_output_PP_2MHz);
5.     gpio_pin_cfg(GPIOB, PB1, gpio_mode_output_PP_2MHz);
6.     gpio_pin_cfg(GPIOB, PB2, gpio_mode_input_floating);
7.
8.     while(1){
9.
10.        if ( GPIOB->IDR & PB2 ){
11.            GPIOB->ODR |= GPIO_ODR_ODR0;
12.            GPIOB->BRR = GPIO_BRR_BR1;
13.        } else {
14.            GPIOB->ODR &= ~GPIO_ODR_ODR0;
15.            GPIOB->BSRR = GPIO_BSRR_BS1;
16.        }
17.    }
18. }
19. /* while(1) */
20.
21. } /* main */
```

Uwaga! Użyta w przykładzie definicja „PB2” nie jest standardową definicją z pliku nagłówkowego. To moje własne dzieło upraszczające zapis - patrz dodatek 1.

Co warto zapamiętać z tego rozdziału?

- operacje atomowe to operacje niepodzielne
- wszelkiej maści sumy i iloczyny bitowe to operacje nieatomowe - składają się z sekwencji trzech operacji RMW (odczytaj, modyfikuj, zapisz)
- nieatomowość rodzi problemy jeśli dane modyfikowane są asynchronicznie (w przerwaniach lub równoległych wątkach programu)

- rejesty GPIO_BSRR i GPIO_BRR umożliwiają atomowe machanie nóżkami

3.4. Cortex-M4 wybieram Cię! (F334 i F429)

Przyjrzymy się teraz jak to wygląda w STM32F334 i STM32F429. Najpierw omówimy F429, a F334 zostawimy sobie na deser. Są to o wiele bardziej rozbudowane mikrokontrolery, więc i portom dostało się więcej opcji. Zakładam, że opis GPIO w STM32F103 masz przeczytany, przetrawiony, przećwiczony, przespany i znalazłeś odpowiedzi na wszystkie wątpliwości jakie się po drodze pojawiły :) GPIO z F103 będzie dla Nas punktem odniesienia.

To co z miejsca rzuca się w oczy, po otwarciu RMa mikrokontrolera STM32F429 (mikrokontrolerem F334 zajmiemy się za chwilę), to inne rejesty konfiguracyjne portów:

- *MODER* – tryb pracy (wejście, wyjście, funkcja alternatywna, pin analogowy)
- *OTYPER* – typ wyjścia (*push-pull*, *open-drain*)
- *OSPEEDR* - nieszczęsna „prędkość” wyjścia
- *PUPDR* – włączanie *pull-upu* lub *pull-downu*
- *IDR* – rejestr wejściowy
- *ODR* – rejestr wyjściowy
- *BSRR* – atomowe kasowanie, ustawianie bitów rejestrów *GPIO_ODR*
- *LCKR* – blokowanie konfiguracji
- *AFRL* – wybór funkcji alternatywnej pinu
- *AFRH* – wybór funkcji alternatywnej pinu

i więcej trybów pracy (tabela 3.3). Osobiście uważam że pomimo, pozornie ciut większego skomplikowania, organizacja rejestrów konfiguracyjnych w F429 jest lepiej przemyślana i prostsza niż w F103. Tutaj każdy rejestr odpowiada za konkretną rzecz. Na co warto zwrócić uwagę:

- tabela nie uwzględnia wszystkich opcji konfiguracji, np. nie ma opcji z PUPDR = 0b11 bo taka konfiguracja (włączenie naraz podciągania pinu w górę i w dół) nie ma sensu! W RM oznaczone jest to jako *Reserved* bez dodatkowych komentarzy.
- ST zrezygnowało z (wprowadzającego w błąd) podawania „prędkości” w MHz przy konfiguracji wyjścia

- nie wszystkie konfiguracje, choć formalnie poprawne, mają sens w praktyce – np. jaki jest sens konfiguracji wyjścia *push-pull* z podciąganiem w górę lub w dół? Dosyć dyskusyjny.
- różne rejesty, dla różnych portów, mają różne wartości początkowe! Tzn. np. MODER dla portu A (GPIOA_MODER) ma inną wartość po resecie niż dla portu B (GPIOB_MODER)
- wcięło rejestr BRR :)

Tabela 3.3 Konfiguracja portów F429

MODER	OTYPER	OSPEEDR ⁶⁸	PUPDR	Konfiguracja ⁶⁹	
01	0	00 – Low Speed 01 – Medium Speed	00	wyjście PP	
			01	wyjście PP + PU	
			10	wyjście PP + PD	
			00	wyjście OD	
			01	wyjście OD + PU	
			10	wyjście OD + PD	
	10		00	f. alternat. PP	
			01	f. alternat. PP + PU	
			10	f. alternat. PP + PD	
			00	f. alternat. OD	
			01	f. alternat. OD + PU	
			10	f. alternat. OD + PD	
00	bez znaczenia		00	wejście pływające	
	bez znaczenia		01	wejście PU	
	bez znaczenia		10	wejście PD	
11	bez znaczenia		00	wejście / wyjście analogowe	

Podobnie jak poprzednio, warto sobie przygotować funkcję ułatwiającą konfigurację portów. Przykład przedstawiam w dodatku 2.

Zadanie domowe 3.5: treść taka sama jak w zadaniu 3.4 tylko mikrokontroler inny. Konfigurację portów proszę przeprowadzić ręcznie na rejestrach.

68 w nawiasach podano nazwy „obowiązujące” do (chyba) 10 wersji *Reference Manuala*, potem nazwy zostały zmienione. I dobrze, zawsze my się myliło czy *High Speed* jest szybsze od *Fast Speed* czy odwrotnie :)

69 *PP – push-pull, PU – pull up, PD – pull down, OD – open drain*

Przykładowe rozwiązanie (F429, diody PG13 i PG14, przycisk PA0):

```
1. #include "stm32f429xx.h"
2.
3. int main(void){
4.
5.     RCC->AHB1ENR = RCC_AHB1ENR_GPIOAEN | RCC_AHB1ENR_GPIOGEN;
6.     __DSB();
7.
8.     GPIOG->MODER = GPIO_MODER_MODER13_0 | GPIO_MODER_MODER14_0;
9.
10.    while(1){
11.
12.        if ( GPIOA->IDR & GPIO_IDR_IDR_0 ){
13.            GPIOG->ODR |= GPIO_ODR_ODR_13;
14.            GPIOG->BSRR = GPIO_BSRR_BR_14;
15.        } else {
16.            GPIOG->ODR &= ~GPIO_ODR_ODR_13;
17.            GPIOG->BSRR = GPIO_BSRR_BS_14;
18.        }
19.
20.    } /* while(1) */
21.
22. } /* main */
```

Proste? Dobra. Przyznać się, kto zauważył w kodzie coś dziwnego?

W STM32F429 występuje mała niedoróbka. Po włączeniu sygnału zegarowego dla jakiegoś bloku, należy chwilę odczekać. W przeciwnym wypadku istnieje ryzyko, że blok nie zdąży się uruchomić i to co wpiszemy do jego rejestrów konfiguracyjnych nie zadziała. Jest to opisane w *erracie* do tego mikrokontrolera (uprzedzałem, że warto do niej zaglądać). Jednym z podanych rozwiązań jest użycie rozkazu *DSB* (*Data Synchronization Barrier*) po włączeniu zegara. DSB, w uproszczeniu, wstrzymuje wykonywanie programu do czasu zakończenia operacji na pamięci.

Przy czym nie ma konieczności stosowania akurat instrukcji barierowej DSB. To nie o nią tu chodzi. Ważne jest krótkie opóźnienie (2 cykle zegara dla układów szyny AHB lub 1 + wartość preskalera AHB/APB cykli dla układów szyny APB - wyjaśni się w rozdziale 17) pomiędzy włączeniem zegara dla układu peryferyjnego, a pierwszym dostępem do jego rejestrów (np. zapisem do rejestrów konfiguracyjnych). Opóźnienie można uzyskać dowolną metodą, byleby była skuteczna. Zamiast instrukcji barierowej, można tak zaplanować program, aby konfiguracja peryferiali nie występowała natychmiast po włączeniu ich zegarów. Każde rozwiązanie jest dobre! W większości przykładowych programów (dla F4) będę stosował instrukcję barierową nawet jeśli opóźnienie będzie wynikało z innych działań w programie. DSB będzie kluło w oczy i przypominało o erracie :)

Zajmijmy się teraz mikrokontrolerem F334. Sprawa jest banalnie prosta. Porty wejścia/wyjścia w tym układzie działają praktycznie tak samo jak w F429. Są dwie różnice (właściwie nawet trzy), którymi chyba warto się zainteresować. Jedną z nich omówimy niedługo w rozdziale 3.5. Druga natomiast dotyczy naszych ulubionych „prędkości wyjść”. Otóż w F334 jest mniej prędkości niż w F429. Ubyło nam jedno *high speed*. Szczegóły zebrane w poniższej tabelce:

Tabela 3.4 Prędkości wyjść w F429 i F334

Wartość pola OSPEEDR	F429	F334
00	<i>Low Speed</i>	<i>Low Speed</i>
01	<i>Medium Speed</i>	<i>Medium Speed</i>
10	<i>High Speed</i>	<i>Low Speed</i>
11	<i>Very High Speed</i>	<i>High Speed</i>

Ot i cała różnica. Nie pozostaje nic innego, jak przećwiczyć to w praktyce.

Zadanie domowe 3.6: napisać program zapalający diodę (szkoda, że Nucleo ma tylko jedną diodę...), gdy wciśnięty jest przycisk. Do dzieła! Btw. historyczny moment - to będzie mój pierwszy program napisany dla F334 :)

Przykładowe rozwiązanie (F334, dioda PA5, przycisk PC13):

```
1. int main(void){  
2.     RCC->AHBENR = RCC_AHBENR_GPIOAEN | RCC_AHBENR_GPIOCEN;  
3.     GPIOA->MODER |= GPIO_MODER_MODE5_0;  
4.  
5.     while(1){  
6.         if(GPIOC->IDR & GPIO_IDR_13) GPIOA->ODR &= ~GPIO_ODR_5;  
7.         else GPIOA->ODR |= GPIO_ODR_5;  
8.     }  
9. }  
10. }
```

Nic się nie zmieniło, więc nie ma co omawiać. A! Errata do F334 nie wspomina o konieczności stosowania opóźnienia między włączeniem zegara a dostępem do rejestrów świeżo włączonego peryferiala.

Wspomniałem o trzech różnicach między F429 a F334 - jedna będzie omówiona w rozdziale 3.5, druga to prędkości wyjść. Została nam jeszcze trzecia. W F334 jest bug⁷⁰ (opisany w erracie), związany z portami GPIO. Dotyczy on funkcji blokowania konfiguracji portu, a dokładniej blokowania rejestru GPIOx_OTYPER, z wykorzystaniem rejestru GPIOx_LCKR. Nieprawidłowość działania objawia się tym, że w przypadku blokowania pinów o numerach od 0 do 7 w rejestrze OTYPER blokowane są pola dotyczące tych pinów oraz pinów o numerach wyższych o 8. Na przykład:

- zablokowanie konfiguracji pinu 0 blokuje rejestr OTYPER dla pinów 0 i 8
- zablokowanie konfiguracji pinu 1 blokuje rejestr OTYPER dla pinów 1 i 9

⁷⁰ „to nie błąd, to dodatkowy, nieplanowany ficzer!” :>

- ...
- zablokowanie konfiguracji pinu 7 blokuje rejestr OTYPER dla pinów 7 i 15

Górna połowa rejestrów LCKR, odpowiedzialna za blokowanie konfiguracji pinów 8-15, w ogóle nie blokuje rejestrów OTYPER! Tzn. że jedynym sposobem na zablokowanie pól z górnej połowy rejestrów OTYPER, jest zablokowanie ich „równolegle” z dolną połową. Na szczęście nie będziemy zbyt często wykorzystywać tej funkcji, więc i błąd nie będzie nam się naprzykrzał.

Co warto zapamiętać z tego rozdziału?

- nie każda możliwa konfiguracja pinu ma sens w praktyce
- po włączeniu zegara (w F429) należy odczekać chwilę przed wykonywaniem operacji na włączanym bloku
- porty w F334 i F429 działają z grubsza tak samo

3.5. Wybór funkcji alternatywnej (F334 i F429)

W F429 i F334 inaczej rozwijano konfigurację pinu do współpracy z układami peryferyjnymi (funkcje alternatywne pinu). Przypomnijmy sobie jak to było w F103:

- jeśli pin miał realizować wyjściową (cyfrową) funkcję alternatywną (np. wyjście PWM albo TxD) to należało ustawić go w trybie *alternate*
- jeśli pin miał realizować wejściową (cyfrową) funkcję alternatywną (np. wejście RxD albo zewnętrzne źródło taktowania licznika) to należało ustawić ten pin tak jak zwykłe wejście
- jeśli pin miał być związany z jakąś funkcją analogową (wejście ADC, wyjście DAC) to należało ustawić go w trybie analogowym

W mikrokontrolerach F334 i F429 sytuacja przedstawia się następująco:

- jeśli pin ma realizować cyfrową funkcję alternatywną to należy go ustawić w tryb *funkcji alternatywnej bez względu na to czy ma być wejściem czy wyjściem*
- jeśli pin ma być związany z jakąś funkcją analogową (wejście ADC, wyjście DAC) to należy ustawić go w trybie analogowym

To czy pin w konfiguracji alternatywnej, będzie działał jako wejście czy wyjście będzie zależało od konfiguracji układu peryferyjnego. Powiem szczerze, że mnie się to rozwiązanie nie podoba i zaraz powiem dokładniej dlaczego.

Z powyższego wynika jeszcze jedna ciekawa rzecz. Pamiętasz jak krytykowaliśmy sens konfiguracji w stylu: wyjście *push-pull* i podciąganie, bo nie miały one sensu w praktyce? W przypadku F334 i F429 konfiguracje: *alternate push-pull + podciąganie*, mają sens. Tzn.: jeśli pin alternatywny będzie pracował jako wyjście to będzie alternatywnym *push-pull* - i tu podciąganie jest dalej bez sensu. Ale! Jeśli tak samo ustawiony pin będzie pracował jako alternatywne wejście (co wynika z konfiguracji peryferiala) to wtedy ten *push-pull* zostanie pominięty (bo wejście nie może być *push-pull*) i zostanie samo *alternate + podciąganie*. Czyli wejście alternatywne podciągnięte wewnętrznym rezystorem. A to już ma sens jak najbardziej.

Następną nowością w F334 i F429 są rejestrzy GPIO_AFRH i GPIO_AFRH, które służą do wyboru funkcji alternatywnej pinu. Praktycznie każdy pin mikrokontrolera może współpracować z kilkoma różnymi układami peryferyjnymi. Czyli nie jest już tak jak w AVR, że przykładowo: PWM z licznika TIM1 można generować (sprzętowo) tylko na nóżce PB7. Mamy pewne (ograniczone ale jednak) pole wyboru na której nóżce co ma być. Służą do tego właśnie rejestrzy GPIO_AFRx. Umożliwiają one przyporządkowanie konkretnej nóżce, jednej z kilku możliwych funkcji alternatywnych, np. aby uprościć projekt płytki. Po szczegółów odsyłam do datasheeta⁷¹ układu F429 - rozdział *Pinouts and pin description* → tabela *Alternate function mapping*. Przykładowo nóżka PF8 może być (sprawdź sam czy potrafisz znaleźć te informacje):

- linią MOSI interfejsu SPI5 – funkcja alternatywna *AF5*
- linią SCK_B interfejsu SAI1⁷² - funkcja alternatywna *AF6*
- kanałem pierwszym licznika TIM13 – funkcja alternatywna *AF9*
- linią NIOWR⁷² interfejsu FMC – funkcja alternatywna *AF12*
- wyjściem EVENTOUT – funkcja alternatywna *AF15*

Sposób konfiguracji funkcji alternatywnych portu zostanie omówiony szczegółowo jak poznamy jakieś peryferia :) Na razie tylko sygnalizuję zagadnienie. Przy konfiguracji nóżki w trybie alternatywnym należy (i to jest trzecia z różnic między portami GPIO w F334 i F429):

- w przypadku mikrokontrolera F334 skonfigurować kolejno:

⁷¹ tak datasheeta nie *reference manuala*!

⁷² cokolwiek to jest

- numer wybranej funkcji alternatywnej (rejestry GPIO_AFRL/H) zgodnie z tabelą z datasheeta
- konfigurację wyprowadzenia (rejestry GPIO_OTYPER, GPIO_OSPEEDR, GPIO_PUPDR)
- tryb alternatywny pinu (w rejestrze GPIO_MODER),
- w przypadku mikrokontrolera F429 RM zaleca inną kolejność konfiguracji:
 - tryb alternatywny pinu (w rejestrze GPIO_MODER)
 - konfiguracja wyprowadzenia (rejestry GPIO_OTYPER, GPIO_OSPEEDR, GPIO_PUPDR)
 - numer wybranej funkcji alternatywnej (rejestry GPIO_AFRL/H) zgodnie z tabelą z datasheeta

Przypuszczam, że powyższe różnice nie mają większego znaczenia, ale dla świętego spokoju można zmienić funkcję konfigurującą porty (z dodatku 2). Wersja dla F334 wygląda praktycznie tak samo, zmienia się kolejność konfiguracji rejestrów.

Rejestry AFR są dwa, bo w jednym nie zmieściłyby się bity konfiguracyjne wszystkich nóżek. Rejestr dolny (AFRL) to konfiguracja pinów 0-7, rejestr górny (AFRH) pinów (8-15).

Co mi się konkretnie nie podoba w tym rozwiążaniu? Uwaga! Trochę będę straszył... ale tylko trochę. Ano nie podoba mi się to, że przy konfiguracji pinu „alternatywnego” nie możemy wybrać od razu jego kierunku. Kierunek będzie wynikał z konfiguracji i „widzi mi się” układu peryferyjnego. Np. jeśli ustawimy licznik tak, aby liczył impulsy z zewnątrz, to pin będzie się zachowywał jak wejście. Jeśli ustawimy licznik tak aby generował PWM, to pin będzie wyjściem. A co jeśli pomylimy się przy konfiguracji licznika (zawsze może się zdarzyć) i pin, który miał działać jako wejście będzie wyjściem? Możemy uszkodzić coś w układzie. Dlatego woałbym już na etapie konfiguracji pinu wybierać kierunek.

Druga sprawa - jak będzie zachowywał się pin alternatywny jeśli związany z nim układ peryferyjny będzie w ogóle wyłączony? Nie znalazłem na to jednoznacznej odpowiedzi w dokumentacji. Z prób i testów na kilku pinach wychodzi, że pin jest wejściem... ale czy to reguła?

I wreszcie największy zarzut jaki mam do tego rozwiązania: nie każda funkcja alternatywna jest dostępna dla każdego pinu. To znaczy np.: nóżka PA15, w F429, może być powiązana z funkcjami alternatywnymi: AF0, AF1, AF5, AF6, AF15. No i pytanie: a co jeśli ustawimy (przez pomyłkę) jedną z nieobsługiwanych funkcji alternatywnych? Np. AF10? Jak się zachowa pin? Dokumentacja milczy. Z moich testów wynika że, o zgrozo, pin jest wtedy wyjściem! Jak dla mnie

to straszna wtopa. Dla zainteresowanych i rządnych dalszej lektury, temat na forum ST: *Alternate Function Input on STM32F4*.

Żeby już tak bardzo nie straszyć powiem na koniec, że przy pisaniu Poradnika uruchomiłem układ w którym miałem połączone dwa piny mikrokontrolera (przewodzikiem). Jeden był wyjściem sygnału PWM. Drugi miał być wejściem, ale przez pomylony numer funkcji alternatywnej działał jak wyjście w stanie niskim... Czyli przez chwilę ten PWM był zustyty do masy przez inną nóżkę. STMy na szczęście nie są jakoś wybitnie delikatne. Pracowało to kilkanaście sekund i nic się nie stało :)

Zadanie domowe 3.7: na podstawie dokumentacji mikrokontrolerów F334 i F429 zidentyfikować nóżki odpowiadające kanałom 1, 2, 3 licznika TIM1 i obczaić sposób konfiguracji (numery funkcji alternatywnych).

Przykładowe rozwiązanie (F429):

Nóżek oczywiście szukamy w *datasheetcie*. Najwygodniej odpalić sobie tabelę: *STM32F427xx and STM32F429xx alternate function mapping*. Lokalizujemy kolumnę z potrzebnym peryferialem (*TIM1*⁷³) i zapamiętujemy numer funkcji alternatywnej odpowiadającej temu peryferialowi (AF1). Teraz lustrujemy kolumnę w poszukiwaniu upragnionych funkcji i szukamy odpowiadających im wprowadzeń:

- kanał 1: TIM1_CH1 - PA8 i PE9
- kanał 2: TIM1_CH2 - PA9 i PE11
- kanał 3: TIM1_CH3 - PA10 i PE13

Jeśli potrzebujemy numerów fizycznych wprowadzań układu scalonego to przeglądamy tabelę *STM32F427xx and STM32F429xx pin and ball definitions*. Dla ułatwienia można sobie w wyszukiwarce w pdfie wpisać np. *TIM1_CH1* lub *PA8*. W tabeli mamy też informacje o tym czy pin toleruje 5V (FT, zaraz się wyjaśni) i ewentualne uwagi. Jak widać, dla każdego kanału mamy dwie nóżki. Z licznikiem będzie współpracowała ta, którą skonfigurujemy w trybie alternatywnym i zapodamy jej funkcję alternatywną AF1. Proste prawda? No ok, wiem. Abstrakcja totalna. Ale jak pojawią się przykłady praktyczne to się wszystko wykłaruje :)

73 kolumna jest akurat wspólna dla *TIM1* i *TIM2*

Przykładowe rozwiązanie (F334):

Postępujemy dokładnie tak samo. Otwieramy tabelkę z datasheeta, tym razem nazywa się ona *Alternate functions*. Lokalizujemy kolumnę z potrzebnym peryferialem (*TIM1*)... o i fajnie się złożyło, bo *TIM1* występuje w kilku kolumnach (AF2, AF4, AF6, AF9, AF11, AF12). Lustrujemy kolumny i notujemy to co nam potrzebne. Dla pewności możemy jeszcze przeszukać datasheet (*Ctrl+F*) pod kątem „*TIM1_CHx*”. Tak czy siak, wynik przedstawia się następująco:

- kanał 1: *TIM1_CH1* - PA8 (AF6) oraz PC0 (AF2)
- kanał 2: *TIM1_CH2* - PA9 (AF6) oraz PC1 (wtf!?)
- kanał 3: *TIM1_CH3* - PA10 (AF6) oraz PC2 (AF2)

I... *Houston, we have a problem!* Mamy zagwozdkę z PC1. W tabeli *STM32F334x4/6/8 pin definitions*, w kolumnie *Alternate functions*, jest informacja, że jedną z funkcji alternatywnych tego wyprowadzenia jest *TIM_CH2*. Co więcej, to ładnie pasuje do tego co już znaleźliśmy - kolejne kanały licznika na kolejnych wyprowadzeniach portu (PC0, PC1, PC2). Obraz tej sielanki psuje nieco tabela z funkcjami alternatywnymi (*Alternate functions*), w której nie ma takiej funkcji alternatywnej dla PC1. Ba! W tej tabeli w ogóle nie ma PC1 (o_0). I weź bądź tu mądry. Śmiem przypuszczać, że... nie wiem - trzeba będzie sprawdzić doświadczalnie⁷⁴ :)

Co warto zapamiętać z tego rozdziału?

- w F334 i F429 to układ peryferyjny decyduje o kierunku nóżki działającej w trybie alternatywnym
- należy uważnie konfigurować funkcje alternatywne, niewłaściwa konfiguracja może spowodować, że pin przeznaczony do pracy jako wejście stanie się wyjściem
- numery funkcji alternatywnych:
 - F334: datasheet → tabela *Alternate functions*
 - F429: datasheet → tabela *STM32F427xx and STM32F429xx alternate function mapping*
- numery wyprowadzeń:
 - F334: datasheet → tabela *STM32F334x4/6/8 pin definitions*
 - F429: datasheet → tabela *STM32F427xx and STM32F429xx pin and ball definitions*

⁷⁴ Empirycznie sprawdziłem i ta funkcja alternatywna działa na PC1... Także ten...

3.6. Remapping funkcji alternatywnych (F103)

Przy pierwszym czytaniu można sobie ten rozdział odpuścić - nie jest trudny, ale za dużo nowości na początku nie pomaga.

W F334 i F429 wybór powiązania między funkcją alternatywną a konkretnym pinem mikrokontrolera dokonywany jest przez rejesty GPIO_AFRx. W F103 też możemy wpływać na to, z którym pinem będzie współpracował dany układ peryferyjny. Służy do tego mechanizm remappingu. Sprawa jest bardzo prosta jak wszystko w STMach. W datasheetie, w tabeli z opisem wyprowadzeń, są dwie kolumny z funkcjami alternatywnymi pinów: *Default* i *Remap*. Domyślnie pin powiązany jest z funkcją alternatywną z kolumny *Default*. Remapping pozwala aktywować na pinie funkcję alternatywną z kolumny *Remap*.

Jak to działa? Otwieramy RM, rozdział *Alternate function I/O and debug configuration* (AFIO). W tym rozdziale znajdują się tabelki pokazujące piny współpracujące z układami peryferyjnymi przy różnym stopniu remapu. Patrzmy np. na tabelkę: *USART3 remapping*. Dotyczy ona trzeciego interfejsu USART. Jak to należy odczytać? Np.:

- bez zastosowania remapu: USART3_TX (linia nadawcza) znajduje się na nóżce PB10
- po wyłączeniu częściowego remapu: USART3_TX (linia nadawcza) znajduje się na nóżce PC10
- po wyłączeniu całkowitego remapu: USART3_TX (linia nadawcza) znajduje się na nóżce PD8

Dodatkowo w tabeli mamy podane wartości bitów USART3_REMAP odpowiadające poszczególnym „stopniom” remapowania. Bity te znajdują się w rejestrze AFIO_MAPR. Czyli, aby ustawić całkowity remap USARTu3 należy wykonać dwie operacje:

- włączyć zegar bloku AFIO jeśli wcześniej nie był włączony
- ustawić pole bitowe USART3_REMAP na wartość 0b11

Oto cała filozofia. Dla innych układów peryferyjnych działa to identycznie. Nie jest to niestety rozwiązanie tak elastyczne jak w F334/F429, ale zawsze coś. W razie potrzeby korzystania z remappingu polecam wcześniej zerknąć do erraty bo z tym mechanizmem powiązanych jest kilka bugów.

Co warto zapamiętać z tego rozdziału?

- remapping daje mocno ograniczoną możliwość zmiany wyprowadzenia mikrokontrolera, związanego z daną funkcją alternatywną
- do wyboru są tylko dwa „stopnie” remappingu (częściowy i całkowity)

- ten mechanizm nie jest tak elastyczny jak rozwiązanie z F334/F429
- korzystając z remappingu należy włączyć taktowanie bloku AFIO

3.7. Elektryczna strona medalu

W ramach odskoczni, popatrzmy na podstawowe parametry elektryczne portów. Poniżej, w tabeli 3.5, podaję zestawienie najważniejszych wartości na podstawie datasheetów STM32F103VC, STM32F334R8 oraz STM32F429ZI. Uwaga! Nie bierz tych danych za pewnik. Podaję je tylko w celach orientacyjnych.

Dwie rzeczy na pewno wymagają komentarza:

- prąd wstrzykiwany (*injected current*)
- piny tolerujące 5V (oznaczone *FT*)

Piny mikrokontrolera najczęściej są zabezpieczone przed przekroczeniem maksymalnego zakresu napięć wejściowych poprzez dwie, wbudowane w mikrokontroler, diody. Jedna dioda łączy daną nóżkę z masą (VSS), druga z zasilaniem (VDD). Jeżeli potencjał na nóżce spadnie poniżej VSS to pierwsza dioda zaczyna przewodzić. Jeżeli potencjał nóżki wzrośnie powyżej napięcia zasilania, to przewodzi druga dioda - prąd odpływa do zasilania. Takie proste zabezpieczenie. AVRy też tak miały. I teraz ten prąd płynący przez diody zabezpieczające, wynikający z przekroczenia zakresu napięć zasilających mikrokontroler, to jest właśnie *injected current*.

W tabeli podane są maksymalne wartości prądu *injected* dla zwykłych pinów i pinów FT, oraz sumaryczna dopuszczalna wartość dla całego układu⁷⁵. Przykładowo zapis: -5mA oznacza, że dopuszczalny prąd wypływający z nóżki, gdy jej potencjał spadnie poniżej potencjału masy wynosi 5mA. Wartości z plusem odnoszą się do prądu wpływającego, gdy potencjał nóżki jest wyższy od V_{DD}. Na koniec jeszcze tylko dorzucę, że ujemny prąd wstrzykiwany zakłóca w jakimś tam stopniu pracę przetwornika ADC.

⁷⁵ pytanie tylko czy to ma być suma geometryczna czy arytmetyczna?

Tabela 3.5 Podstawowe parametry elektryczne portów GPIO

mikrokontroler	STM32F103VC	STM32F429ZI	STM32F334R8
maksymalny prąd upływu wejścia	$\pm 1\mu A$ dla zwykłych pinów lub $3\mu A$ przy napięciu 5V dla pinów FT ⁷⁶		
	$10\mu A$ przy napięciu 5V dla pinów FT ⁷⁷		
rezystory podciągające (pull-up, pull-down)	$40k\Omega$	$10k\Omega$ dla PA10 i PB12 $40k\Omega$ dla pozostałych	$40k\Omega$
obciążalność wyjść	$\pm 8mA$ (lub $\pm 20mA$ jeśli nie zależy nam na utrzymaniu katalogowych poziomów napięć wyjściowych) $\pm 3mA$ dla PC13, PC14, PC15, PI8		
	$\pm 25mA$		
absolutnie maksymalny prąd wyjścia	$150mA$	$270mA$	$140mA$
minimalne napięcie wejściowe pinu	$V_{ss} - 0,3V$		
maksymalne napięcie wejściowe pinu	$4V$ $V_{dd} + 4V$ dla pinów FT		
maksymalny prąd wstrzykiwany pinu FT	$+0 / -5mA$		
maksymalny prąd wstrzykiwany pozostałych pinów	± 0 dla OSC_IN32, PA4, OSC_OUT32, PA5, PC13 $\pm 5mA$ dla pozostałych	± 0 dla PA0...PA3, PA6, PA7, PB0, PC0...PC5, PH1...PH5 $+ 5 / - 0mA$ dla PA4 i PA5 $\pm 5mA$ dla pozostałych	$+5 / -0mA$ dla PC0...PC3, PF1 $+0 / -5mA$ dla PB10 i PB11 $\pm 5mA$ dla pozostałych
pojemność pinu	$5pF$		

FT oznacza *Five (Volt) Tolerant*. Pomimo że zasilanie mikrokontrolera to 3,3V, większość pinów to piny FT (tolerujące 5V). Aby się upewnić czy dany pin jest FT należy odszukać go w datasheetcie w tabelce *Pin Definitions* w rozdziale *Pinouts and pin descriptions*. Piny tolerujące 5V mają oznaczenie FT w kolumnie *I/O Level*. Na piny FT może być podane napięcie wyższe niż napięcie zasilania mikrokontrolera, do 4V więcej niż V_{dd} . Nie posiadają one diody zabezpieczającej włączonej między nóżkę a dodatnią szynę zasilania mikrokontrolera - stąd nie jest możliwe wystąpienie dodatniego prądu *injected* (patrz tabela 3.5).

W dokumentacji nie ma zbyt wielu szczegółów na temat FT. Uprzedzam, że poniższe informacje pochodzą z Internetu (głównie z forum ST: temat [STM32 5V tolerant I/O ?](#) i wypowiedź użytkownika *waclawek.jan*):

- piny FT są zabezpieczone czymś w rodzaju 4V zenerki do V_{dd}

76 *Five (Volt) Tolerant* – oznaczenie pinów tolerujących napięcie 5V w trybie wejściowym

77 *Five (Volt) Tolerant* – oznaczenie pinów tolerujących napięcie 5V w trybie wejściowym

- zasada *maksymalnie* $V_{dd}+4V$ obowiązuje również jeśli procesor nie jest zasilany – tzn. nie można podać więcej niż 4V na pin FT jeśli procesor nie jest zasilany ($V_{DD}=0$)
- tolerancja wyższych (niż V_{dd}) napięć dotyczy tylko pinów FT pracujących jako wejście lub wyjście typu open-drain⁷⁸
- włączenie pull-upu tudzież pull-downu jeśli na pinie jest napięcie wyższe niż V_{dd} wydaje się być kiepskim pomysłem

Co warto zapamiętać z tego rozdziału?

- większość pinów STMa toleruje napięcie 5V na wejściu (nie ma możliwości aby uzyskać 5V na wyjściu!)
- obciążalność wyjść wynosi $\pm 20mA$ (jeśli nie zależy nam na poziomach napięć)
- wewnętrzne rezystory podciągające mają około $40k\Omega$

3.8. Skompensuj mi celę (F429)

W F429 występuje coś takiego jak *I/O Compensation Cell*. Z tego co udało mi się znaleźć w Internecie wynika, że funkcja ta ogranicza wpływ portów GPIO na stabilność zasilania mikrokontrolera. Tzn. prądy pobierane przez bufory wyjściowe portów nie szarpią tak zasilaniem całego układu :)

Datasheet zaleca włączenie kompensacji przy napięciu zasilania powyżej 2,4V i korzystaniu z wyjść o „prędkości” ustawionej na 50MHz lub więcej. Gdzieś się też dogrzebałem, że włączenie kompensacji powoduje wzrost poboru prądu przez mikrokontroler o około 0,22mA.

Z komórką kompensacyjną związany jest rejestr SYSCFG_CMPCR. Zawiera on:

- bit włączający kompensację: CMP_PD
- flagę wskazującą czy kompensacja działa: READY

Co warto zapamiętać z tego rozdziału?

- wychodzi na to, że można włączyć i zapomnieć. Tyle w temacie.

78 oczywiście musimy pilnować natężenia prądu w stanie niskim i zadbać o jego ograniczenie w razie potrzeby

4. BIT BANDING („*GEMMA GEMMARUM*”⁷⁹)

4.1. Ale o co chodzi?

*Bit banding*⁸⁰ (*bb*) jest wspaniałym (i prostym) mechanizmem oferowanym przez rdzeń Cortex-M. Rdzeń, nie mikrokontroler! Czyli w każdym Cortexie działa tak samo... tzn. nie każdym - już wspomniałem że CM0 nie ma wielu fajnych rzeczy... BB umożliwia dokonywanie atomowych (z punktu widzenia programu, patrz rozdział 4.2) operacji bitowych na pamięci SRAM i rejestrach peryferiali. O operacjach atomowych wspomniałem już przy opisie portów GPIO, w rozdziale 3.3.

Problem wynikający z nieatomowości pewnych operacji występował również w AVRach. Miał tylko mniejsze znaczenie ze względu na mało rozwinięty system przerwań i marginalne stosowanie systemów wielowątkowych. Dodatkowo architektura AVR umożliwia atomowe kasowanie i ustawianie bitów rejestrów układów peryferyjnych za pomocą specjalnych rozkazów: *sbi*, *cbi*. ARM tego bajeru nie oferuje. Mamy za to *bit banding*, który daje większe możliwości - np. możliwość odczytywania bitów rejestru.

OT: kompilator AVR-GCC kiedyś nie wspierał tego mechanizmu architektury AVR (rozkazów *sbi*, *cbi*). Żeby jawnie wymusić operacje atomowe wprowadzono specjalne makra/wstawki asm: SBI i CBI. W Internecie bez problemu znajdziesz programy na AVR z ich użyciem. Na szczęście to już przeszłość. Od której tam wersji GCC korzysta z *sbi*, *cbi* bez kombinowania.

Zdaję sobie sprawę, że jest to kompletna nowość. I nawet planowałem dać ten rozdział trochę później, żeby nie przesadzać na początku z nowinkami... ale mechanizm jest na tyle wygodny, że nie chce mi się potem pisać przykładowych kodów bez bb. Lenistwo wygrało i omówię bb już na starcie żeby móc z niego korzystać :)

Już w 1966r Nancy Sinatra śpiewała (czy jakoś podobnie... w filmie *Kill Bit* była ta piosenka):

„*Bit band, I've set to one
Bit band, you're set to one
Bit band, an atomic fun
Bit band, I used to set to one*”

79 „Klejnot nad klejnotami.”

80 ktoś ma pomysł na sensowne tłumaczenie tego pojęcia? „dowiązania bitowe”?

Co warto zapamiętać z tego rozdziału?

- *bit banding* jest mechanizmem umożliwiającym przeprowadzanie atomowych (z punktu widzenia programu) operacji na bitach
- bb obejmuje rejesty konfiguracyjne peryferiali mikrokontrolera i pamięci SRAM
- bb jest rewelacyjny, prosty i praktyczny
- bb jest Twoim przyjacielem!

4.2. Jak to działa?

BB pozwala na atomowe ustawianie, kasowanie oraz odczytywanie (!) bitów z pamięci. Działanie bb opiera się na jakimś magicznym sprzętowym *hokus-pokus*, które łączy dwa obszary pamięci. Pierwszy obszar - ***bit-band region*** - zawiera dane, w których chcemy atomowo zmieniać bity (lub je odczytywać). Drugi obszar - ***bit-band alias*** - zawiera słowa (32 bitowe) „połączone” z bitami *bb regionu*. Każdemu kolejnemu bitowi z regionu przyporządkowane jest jedno słowo z aliasu. I teraz uwaga! Zapisanie do danego słowa aliasu wartości „1”⁸¹ powoduje automatyczne, sprzętowe i atomowe ustawienie odpowiadającego mu bitu w *bb regionie*. Analogicznie wpisanie wartości zero⁸¹ do słowa aliasu, powoduje skasowanie bitu regionu. Proszę sobie to dobrze przemyśleć – mechanizm jest bardzo prosty :)

W tym miejscu musimy się na chwilę zatrzymać i doprecyzować jedną rzecz. Jeżeli nie interesują Cię mechanizmy działania bit bandingu, a jedynie jego „użytkowa strona” to możesz ten kawałek opuścić. Ale, coś za coś, wtedy nie zrozumiesz haczyka opisywanego w rozdziale 4.6 i będziesz musiał przyjąć go na wiare.

Przy okazji omawiania bit bandingu, wiele razy pojawią się określenia: „atomowo”, „bez sekwencji *read-modify-write*”. Warto sobie zdawać sprawę, że ten brak sekwencji RMW występuje tylko z punktu widzenia programu. Bit banding z punktu widzenia pamięci, której zawartość modyfikujemy z wykorzystaniem bb, niczym nie różni się od „zwyczajnego” dostępu. Na pamięci wykonywana jest zwyczajna operacja RMW.

Powyżej napisałem, że wpisanie 0 lub 1 do słowa aliasu uruchamia jakieś magiczne *hokus-pokus*, które ustawia bit w regionie bb. To „*hokus-pokus*” to sprzętowy mechanizm, który wykonuje na pamięci (regionie bb) sekwencję *read-modify-write*. Cała magia bb polega na tym, że zamiast robić w programie nie-atomową operację RMW, robi ją za nas sprzęt (jakiś *System Bus*

⁸¹ słowo ma 32 bity, ale w przypadku bb liczy się tylko najmłodszy bit, reszta jest bez znaczenia

Controller, czy coś takiego). Czyli: w programie wykonujemy (atomowy) zapis (zera lub jedynki) do pamięci aliasu, a to wyzwala sprzętową sekwencję *read-modify-write*. Ponadto sprzęt (*System Bus Controller*) pilnuje, aby w czasie wykonywania tego sprzętowego RMW, procesor nie modyfikował pamięci. Styknie tego, wracamy do głównego wątku :]

W poprzednim akapicie (przed fragmentem wydzielonym poziomymi odcinkami) wspomniałem, że bb daje większe możliwości niż AVRowe *sbi* i *cbi*. Te większe możliwości dotyczą tego, że za pomocą bb można modyfikować bity związane z peryferiami mikrokontrolera (tak jak w AVR) oraz - i tu nowość - zawartość pamięci SRAM mikrokontrolera. No i bb pozwala odczytywać pojedyncze bity. Trzy ważne uwagi:

- *bit banding* nie ma dostępu do rejestrów peryferiów rdzenia (np. do SysTicka)
- DMA nie ma dostępu do aliasu bb (trochę pieśń przyszłości, ale wolę wspomnieć)
- *bit band region* nie musi obejmować wszystkich peryferiów mikrokontrolera

Ta ostatnia uwaga chyba wymaga małego komentarza. Za pomocą bb można modyfikować⁸² pamięć, która leży w zakresie adresów objętych regionem bb. Region bb dla układów peryferyjnych, zarówno w Cortexie M3 jak i M4, zaczyna się od adresu 0x4000 0000 kończy zaś pod adresem 0x400F FFFF (szczegóły nadchodzą w rozdziale 4.3). BB ma dostęp tylko do peryferiów leżących w tym obszarze! No to teraz, posługując się mapką pamięci (*Memory Map*) z RMa posiadanej mikrokontrolera sprawdź, które układy nie załapały się na bb. Nie czytaj dalej póki sam nie sprawdzisz, ćwiczeń nigdy za dużo!

- F103: USB OTG FS i FSMC
- F334: GPIOA, GPIOB, GPIOC, GPIOD, GPIOF, ADC1, ADC2
- F429: USB OTG FS, DCMI, CRYP, HASH, RNG, FSMC, FMC

Układami USB, DCMI i kontrolerami pamięci F(S)MC i tak nie będziemy się na razie zajmować. Ale o portach I/O i przetwornikach ADC w F334 radzę pamiętać :)

Co warto zapamiętać z tego rozdziału?

- bb nie ma dostępu do rejestrów rdzenia i jego peryferiów

⁸² z rozpoczętu piszę o modyfikowaniu, ale proszę nie zapominaj, że bb pozwala też odczytywać pamięć :)

- bb nie musi obejmować wszystkich rejestrów mikrokontrolera (porty I/O w F334!)
- DMA nie ma dostępu do bb
- bb *alias* zawiera słowa, sprzętowo połączone z bitami bb *regionu*
- zapis do aliasu bb wyzwala sekwencję RMW realizowaną sprzętowo

4.3. Jasne i proste jak cała matematyka

W tym rozdziale opiszę sposób obliczania adresu w bb *aliasie* odpowiadającego bitowi w bb *regionie*. Sprawa jest prosta, choć na początku wydaje się prawie wiedzą tajemną. Nie jest to wiedza niezbędna, bo i tak stworzymy sobie uniwersalne narzędzie liczące to za nas. Jedziemy.

Zacznijmy od informacji o tym jakie adresy obejmuje bb *region* i bb *alias*. Odpalamy dokumentację⁸³ i szukamy rozdziału o bb. Przykładowo PM0056:

Tabela 4.1 Adresy regionu i aliasu bb

obszar pamięci	adres początku	adres końca
SRAM bb region	0x2000 0000	0x200F FFFF
SRAM bb alias	0x2200 0000	0x23FF FFFF
Peripheral bb region	0x4000 0000	0x400F FFFF
Peripheral bb alias	0x4200 0000	0x43FF FFFF

Tak jak wspominałem bb umożliwia manipulację rejestrami peryferiów (*peripheral*) oraz pamięcią SRAM, stąd dwa zestawy obszarów pamięci. W tym miejscu polecam zerknięcie na mapę pamięci mikrokontrolera i sprawdzenie co obejmują regiony bb - tak w ramach ćwiczeń. W szczególności warto wiedzieć czego za pomocą *bit bandingu* nie zmienimy, żeby się potem nie zapędzić :)

W dokumentacji jest ładna formułka jak obliczyć adres słowa aliasu odpowiadający danemu bitowi regionu bb. Wygląda to strasznie skomplikowanie. Spróbujmy sami do tego dojść. Rozpatrzmy region związany z peryferiami. Znamy adres początkowy regionu i odpowiadającego mu aliasu (tabela 4.1). Ponadto wiemy, że kolejne słowa aliasu odpowiadają kolejnym bitom regionu, czyli licząc od początku regionu:

- pierwszemu bitowi regionu będzie odpowiadało pierwsze słowo aliasu, o adresie 0x4200 0000

⁸³ opis bb jest w *Programming Manualu* bo to funkcjonalność rdzenia, w RM też coś jest ale bez szczegółów

- drugiemu bitowi regionu będzie odpowiadało drugie słowo aliasu, o adresie 0x4200 0004⁸⁴
- trzeciemu bitowi regionu będzie odpowiadało trzecie słowo aliasu, o adresie 0x4200 0008
- itd...

My oczywiście nie będziemy chcieli ustawić np. 325 bitu regionu, tylko np. 5-ty bit rejestru GPIOB_ODR, czy ogólnej n-ty bit rejestru X. Musimy więc policzyć, który to jest bit od początku regionu. W tym celu wykonujemy takie oto wielce skomplikowane działania:

- obliczamy przesunięcie (w bajtach) od początku bb *regionu* do naszego rejestru *X*:

$$\text{offset_rejestru} = \text{adres_rejestru_X} - \text{adres_początku_regionu}$$
- obliczamy ile bitów było od początku regionu do naszego rejestru *X* (czyli w obliczonym powyżej *offsecie*):

$$\text{ilość_bitów_od_początku_regionu_do_X} = \text{offset_rejestru} * 8$$
- dodajemy do powyższego, numer bitu który chcemy zmienić w rejestrze *X*:

$$\text{ilość_bitów_od_początku_regionu_do_bitu_n} = \text{ilość_bitów_od_początku_regionu_do_X} + n$$

Ta dam! Wiemy już którym bitem, od początku regionu bb, jest n-ty bit rejestru *X*. Jak więc policzyć adres słowa, z którym będzie powiązany nasz bit? Wiedząc że każdemu kolejnemu bitowi odpowiada słowo (4B) *aliasu* i znając adres początkowy *aliasu* wykonujemy ostateczną operację:

$$\text{adres_w_aliasie} = \text{ilość_bitów_od_początku_regionu_do_bitu_n} * 4 + \text{adres_początkowy_aliasu}$$

Suma summarum, po uporządkowaniu całości otrzymujemy:

$$\text{adres_w_aliasie} = \text{adres_pocz_aliasu} + (\text{adres_rejestru_X} - \text{adres_pocz_regionu}) * 32 + \text{nr_bitu} * 4$$

Domyślamsię, że wygląda strasznie... ale spokojnie, jak wspominałem, zrobimy sobie narzędzie obliczające to za nas i zapomnimy o matematyce. Narzędzie opisane zostało szczegółowo w dodatku 3 i trochę w rozdziale 4.4.

Co warto zapamiętać z tego rozdziału?

- że w razie potrzeby można tu znaleźć opis jak policzyć adres w *aliasie* bb

⁸⁴ kolejnym bitom odpowiadają kolejne słowa, słowo ma 4B stąd adres rośnie o 4

4.4. Makro do *bit bandingu*

Przykład: chcemy ustawić piąty bit rejestru GPIO_ODR dla portu B. Podejście klasyczne:

```
GPIOB->ODR |= GPIO_ODR_ODR5;
```

za pomocą makra bb (patrz dodatek 3) będzie to wyglądało tak:

```
BB(GPIOB->ODR, GPIO_ODR_ODR5) = 1;
```

lub (definicje wyprowadzeń typu PA0, PA1, PB5, ... nie pochodzą ze standardowego pliku nagłówkowego, to mój twór poprawiający przejrzystość i skracający zapis):

```
BB(GPIOB->ODR, PB5) = 1;
```

za pomocą makra można też kasować, negować i odczytywać bity:

```
BB(GPIOB->ODR, GPIO_ODR_ODR5) = 0;
```

```
BB(GPIOB->ODR, GPIO_ODR_ODR5) ^= 1;
```

```
if ( BB(GPIOB->ODR, GPIO_ODR_ODR5) == 1 ) ...
```

Prawda że wygodne w użyciu?

Żeby było bardziej edukacyjnie porównajmy kod klasyczny i powstały po użyciu bb. Poniżej wycinek pliku *.iss dla wersji klasycznej (umiejętność analizy kodów asm jest szalenie przydatna!):

```
1. GPIOB->ODR |= GPIO_ODR_ODR5;
2. 8000188: 4a05    ldr    r2, [pc, #20]      ; (80001a0 <main+0x1c>)
3. 800018a: 4b05    ldr    r3, [pc, #20]      ; (80001a0 <main+0x1c>)
4. 800018c: 68db    ldr    r3, [r3, #12]
5. 800018e: f043 0320 orr.w r3, r3, #32
6. 8000192: 60d3    str   r3, [r2, #12]
7. ...
8. 80001a0: 40010c00 .word 0x40010c00
```

Przy analizie będziemy opierać się na opisie rozkazów rdzenia Cortex-M3, który znajduje się w *Programming Manualu*. Pierwsza linia to kod w C, poniżej jest „odpowiadający mu” kod *asm*. Trzeba pamiętać, że to nie jest ścisły związek, szczególnie przy wyższych poziomach optymalizacji

związek jest... nikły. No to czytamy nasz listing, spróbujemy wycisnąć z niego jak najwięcej - edukacyjnie... jak na lekcjach języka polskiego „*co kompilator miał na myśli?*”.

Pierwsza instrukcja to rozkaz *ldr*. Znajduje się on w pamięci programu pod adresem *0x0800 0188*. Pamiętasz opis wspólnej przestrzeni adresowej ze wstępem (2.2)? Ten adres to obszar jakiej pamięci? *LDR* powoduje załadowanie do rejestru ogólnego *r2* wartości spod adresu *PC+20*. Jest to adresowanie pośrednie, oparte o licznik programu *PC* (*Program Counter*). PC wskazuje adres aktualnie wykonywanej instrukcji. W nawiasie mamy podpowiedź, że chodzi o wartość spod adresu *0x0800 01A0*. Pod wspomnianym adresem jest zapisana wartość *0x4001 0C00*. Jaki obszar pamięci wskazuje ten adres? Na mapie pamięci sprawdzamy, że jest to adres bazowy portu B. Czyli od tego adresu zaczynają się rejesty związanego z portem B.

W trzeciej linijce, ta sama wartość jest ładowana do rejestru ogólnego *r3*. Trzeci rozkaz *ldr* powoduje załadowanie do rejestru ogólnego *r3* tego co znajduje się pod adresem *r3+12*. Wiemy, że w *r3* znajdował się adres początkowy portu B (*0x4001 0C00*), po dodaniu przesunięcia 12 otrzymujemy adres rejestru ODR portu B. Sprawdzić to można w kilku miejscach w RM:

- przy opisie każdego rejestru jest podany *Address offset* czyli przesunięcie względem adresu bazowego, np. dla rejestru ODR wynosi *0x0C* (czyli 12 w systemie dziesiętnym... kto by się spodziewał)
- za opisem rejestrów, jest coś takiego jak *Register map* – taka tabela z podsumowaniem wszystkich rejestrów bloku – *offset* podany jest w pierwszej kolumnie tabeli

Wracamy do analizy. Wiemy już, że do *r3* ładowana jest wartość rejestru *GPIOB_ODR*.

Rozkaz, z linii 5., *orr* to suma bitowa. Sufiks „*w*” oznacza operację na całym słowie czy coś w tym stylu. Zawartość rejestru *r3* zostaje więc zsumowana (bitowo) z liczbą 32 (*1<<5*) i zapisana nazad w *r3*. Ostatni rozkaz (*str*) powoduje wpisanie wartości z *r3* pod adres *r2+12* czyli z powrotem do rejestru *GPIOB_ODR*.

Uff. Prawda, że proste :) No to podsumujmy telegraficznie takim pseudo-kodem:

1. *r2 = &GPIOB* (adres portu B do *r2*)
2. *r3 = &GPIOB* (adres portu B do *r3*)
3. *r3 = *(r3 + 12)* (odczytanie rejestru ODR do *r3*)
4. *r3 = r3 | 32* (suma bitowa *r3* i *1<<5*)
5. **(r2+12) = r3* (zapisanie *r3* do rejestru ODR)

Udało się i jest to w pełni zgodne z tym czego oczekiwaliśmy. No to z jedziemy z *bit bandingiem*:

```

1. BB(GPIOB->ODR, GPIO_0DR_ODR5) = 1;
2. 8000188: 4b04      ldr    r3, [pc, #16] ; (800019c <main+0x18>)
3. 800018a: 2201      movs   r2, #1
4. 800018c: 601a      str    r2, [r3, #0]
5. ...
6. 800019c: 42218194 .word 0x42218194

```

Już widać, że krótsze :) W pierwszym rozkazie wartość `0x4221 8194` ładowana jest (poprzez adresowanie pośrednie) do `r3`. Do rejestru `r2` jest zapisywana stała o wartości 1. Ostatni rozkaz to wpisanie wartości rejestru `r2` pod adres z rejestru `r3`. Prościzna. Dla asemblero-fobów, w takim pseudo-kodzie:

- `r3 = 0x4221 8194`
- `r2 = 1`
- `*(r3) = r2`

Jeśli Czytelnik uważnie studiował poprzedni rozdział o mechanizmie działania bb to zapewne już snuje podejrzenia, że ta wartość z czwórką na początku, to wyliczony adres słowa w *aliasie* odpowiadającego bitowi 5 rejestru GPIOB_ODR w *BB regionie*. Policzymy i sprawdźmy:

- adres początku *bb regionu*: `0x4000 0000`
- adres rejestru GPIOB_ODR: `0x4001 0C0C`
- nr bitu: 5
- adres początku aliasu: `0x4200 0000`

formułkę mamy już wyprowadzoną, więc teraz tylko podstawiamy:

$$0x4200\ 0000 + (0x4001\ 0C0C - 0x4000\ 0000) * 32 + 5 * 4 = 0x4221\ 8194 \text{ (*fanfary*)}$$

Wyszło co miało wyjść... no i fajnie. W analogiczny sposób można korzystać z makra do operacji na bitach słowa w pamięci SRAM. Szczegóły opisano w dodatku 3.

Co warto zapamiętać z tego rozdziału?

- składnię i sposób korzystania z makra do bb:
 - ustawianie bitu: $BB(REGISTER, BIT) = 1;$
 - kasowanie bitu: $BB(REGISTER, BIT) = 0;$
 - negowanie bitu: $BB(REGISTER, BIT) \wedge= 1;$
 - odczytywanie bitu: $bit = BB(REGISTER, BIT);$

4.5. Kiedy stosować *bit banding*

Na koniec może zrodzić się pytanie: kiedy ustawiać rejesty klasycznie, kiedy *bit bandingiem*, a kiedy np. stosować atomowe *BSRR* i *BRR* przy portach. Prostej odpowiedzi pewnie na to nie ma. W trakcie mojej przygody z STMami wypracowało mi się takie podejście:

- przy konfiguracji i inicjalizacji peryferiów - kiedy jest dużo bitów do ustawienia - podejście klasyczne
- przy prostych operacjach na pojedynczym bicie (włącz, wyłącz, skasuj flagę) - *bit banding*
- rejestrów *GPIO_BSRR* i *GPIO_BRR* nie użyłem chyba nigdy, jedyna sytuacja kiedy mogą się okazać przydatne to potrzeba skasowania bądź ustawienie kilku pinów jednocześnie

Powyzsze reguły podyktowane są subiektywną wygodą i przejrzystością zapisu. Jasnym jest, że jeśli zależy nam na uzyskaniu atomowości to nie ma o czym gadać i podejście klasyczne odpada w przedbiegach. Ponadto trzeba zwrócić uwagę na mały haczyk związany z bit bandingiem, opisany w nadchodzącym rozdziale (rozdział 4.6).

Co warto zapamiętać z tego rozdziału?

- a trzeba tu coś specjalnie zapamiętywać?

4.6. A gdzie jest haczyk?

Kiedyś myślałem, że w sumie to haczyka nie ma. Całkiem niedawno, jakoś po publikacji Poradnika w wersji 1.5 :), zupełnie przypadkiem nadziałem się na coś co mnie zupełnie zaskoczyło i pokazało, że haczyk jest i to całkiem zacny. Nie wiem jak mogłem to wcześniej przegapić...

W rozdziale 4.2, we fragmencie wydzielonym za pomocą odcinków horyzontalnych, opisałem odrobinę sposób działania bit bandingu. Tzn. że modyfikacja bitu, z pomocą *bb*, to sekwencja *read-modify-write* realizowana na poziomie sprzętu. No i jest ona „atomowa”, bo *System Bus Controller* wstrzymuje procesor, jeśli ten próbuje dobrać się do pamięci w trakcie trwania operacji RMW.

Haczyki (z powyższym związanem) na jakie możemy się nadziać są dwa. I są bardzo podobne do siebie. Oba dotyczą dostępu do wszelkiej maści rejestrów statusowych układów peryferyjnych. Ten rozdział, niestety, troszkę wyprzedza nasz „stan wiedzy”. To czego na razie nie rozumiesz,

przyjmuj na wiarę :) Potem tu wróćisz, gdy dzięki dalszej lekturze Poradnika i wiedzy zeń płynącej, rozkwitniesz niczym... niczym coś co rozkwita... dajmy temu spokój.

Pierwszy haczyk dotyczy rejestrów zawierających bity kasowane poprzez wpisanie doń jedynki. W dokumentacji (RM) bity takie oznaczone są skrótem⁸⁵ *rc_w1*. Skrót oznacza, że bit może być czytany (*read*) oraz kasowany (*clear*) poprzez wpisanie jedynki (*write 1*). Przykładem takiego rejestru jest rejestr zawierający flagi przerwań zewnętrznych (EXTI_PR). Wyobraźmy sobie teraz, że chcemy skasować jedną z flag tego rejestru i, niezbyt szczęśliwie, postanowiliśmy wykorzystać do tego bit banding. Posiłkując się naszym makrem, piszemy więc na przykład coś takiego:

```
BB(EXTI->PR, EXTI_PR_PR5) = 1;
```

zgadza się? No zgadza - flaga kasowana wpisaniem jedynki, wpisujemy jedynkę na odpowiednią pozycję. Skasowaliśmy flagę, pozamiatane, w czym problem? Problem w tym, że jeżeli w rejestrze były ustawione jeszcze jakieś flagi *rc_w1*, to je też pozamiataliśmy przy okazji. Wiesz czemu?

BB to sekwencja RMW, tylko sprzętowa. Zawartość rejestru EXTI_PR została odczytana. „Coś sprzętowego”, co realizuje operacje bit bandingowe, ustawiło w tej wartości bit EXTI_PR_PR5 po czym wynik został nazad wpisany do rejestru. A co jeżeli kilka flag w rejestrze było ustawionych? Wtedy w odczytanej wartości było kilka jedynek i teraz, gdy została ona wpisana do EXTI_PR, wszystkie te flagi zostały skasowane! Olaboga! Mocium Panie, dramat! Przemyśl sobie dobrze ten przypadek, bo łatwiej zapamiętać coś co się rozumie :) Rada? Rada jest prosta: nie modyfikować rejestrów zawierających pola *rc_w1* za pomocą bb a jeśli już, to po dogłębnym przemyśleniu sprawy. I tyle. Na szczęście nie ma zbyt wielu takich rejestrów.

Czas na drugi haczyk. Wynika on z tego, że z punktu widzenia pamięci, operacje bb niczym nie różnią się od zwyczajnych (programowych) sekwencji RMW. BB też wywołuje RMW, jeno realizowane przez sprzęt. I tak jak wspominałem, w rozdziale 4.2, procesor jest blokowany jeśli w czasie sekwencji RMW będzie próbował uzyskać dostęp do tej pamięci. Ale! Nie tylko procesor może modyfikować „pamięć”. Jeżeli za pomocą bb będziemy modyfikowali rejesty układowego peryferyjnego, np. licznika, to ten licznik nie zostanie zablokowany na czas operacji RMW. Czyli jest ryzyko, że zawartość jego rejestrów (np. rejestr statusowego) się zmieni. W tym momencie to nie będzie operacja atomowa! A czym grozi zmiana zawartości rejestrów modyfikowanego nieatomową sekwencją RMW już wiemy z rozdziału 3.3.

Rada? Trzeba myśleć co się robi :) To nie jest problem wynikający z bit bandingu! W takim samym stopniu dotyczy on bezpośrednich operacji na pamięci, ale korzystając z bb trudniej dostrzec zagrożenie! Za każdym razem, gdy modyfikowane są rejesty zawierające pola których

⁸⁵ lista wszystkich oznaczeń znajduje się na początku RMa (*List of abbreviations for registers*)

stan zmienia się sprzętowo, należy przemyśleć, czy ewentualna operacja RMW czegoś nam nie nadpisze. Innej rady nie ma znam :)

Zdaję sobie sprawę, że w tym momencie nie wszystko rozumiesz. Luz. Wszystko się z czasem ułoży. Być może pokazałem bb w trochę złym świetle. Ale tak nie jest! To naprawdę użyteczny mechanizm i nie trzeba się go bać. Serio!

No w sumie to znam jeszcze trzy pułapki związane z bb... O jednej już wspomniałem w rozdziale 4.2. Chodzi o to, że w niektórych mikrokontrolerach bb nie obejmuje wszystkich układów peryferyjnych. Jak się tego nie sprawdzi i z rozpetto zacznie się stosować bb to wychodzą bzdury. Na szczęście to jest stosunkowo łatwy do wykrycia błąd.

Druga pułapka dotyczy pamięci CCM (*Core Coupled Memory*) występującej w mikrokontrolerach z rdzeniem Cortex-M4. Bit banding nie obejmuje pamięci CCM. Nieśmiało strzelić że to, że ktoś początkujący będzie zaraz kombinował z bb i pamięcią CCM na początku swojej przygody z STMami, jest równie prawdopodobne jak to, że komuś uda się pomalować amelinium.

Co do ostatniej pułapki... pułapka to złe słowo... tak czy siak - żadnych wiedzy i szukania dziury w całym odsyłam do dodatku 4.

Co warto zapamiętać z tego rozdziału?

- należy diabelnie uważać modyfikując za pomocą bb rejesty zawierające pola *rc_w1*
- w ogóle należy uważać z rejestrami, których pola są modyfikowane sprzętowo (bb nie jest dla nich atomowy!)
- nie wszystkie rejesty układów peryferyjnych muszą być objęte bb
- bb wbrew pozorom nie jest podstępny i jest naszym przyjacielem

5. WYJĄTKI („*MACTE ANIMO, IUVENIS!*”⁸⁶)

Proszę wyłączyć blokadę psychiczną i zanurzyć się w lekturze :) Będzie sporo nowości i trochę teorii o tym jak to *Cortex* ogarnia wyjątki. To nie gryzie. I nie trzeba wiedzieć 75% z tego co zaraz przeczytasz, aby pisać prościutkie programy w C. Choć na pewno wiedza nie zaszkodzi. Miłej lektury. Przy pierwszym czytaniu można ominąć pierwszy podrozdział.

Wszystko co przeczytasz w tym rozdziale, dotyczy zarówno F103 (*Cortex-M3*) jak i F334/F429 (*Cortex-M4*). Chyba, że wyraźnie napisano inaczej :)

5.1. Trochę zbędnej teorii o trybach pracy rdzenia

Rdzeń Cortex może pracować w dwóch trybach (*thread mode* i *handler mode*) i na dwóch poziomach uprzywilejowania (*privileged*, *unprivileged*). Dla uspokojenia powiem, że dla początkującego programisty większego znaczenia to nie ma. Potraktuj to jako ciekawostkę.

Procesor po resecie pracuje w trybie „użytkownika” (*thread mode*) i na poziomie uprzywilejowanym. Drugi tryb (*handler mode*) to tryb obsługi wyjątków. Podjęciem wyjątku mieści się wszystko, co zaburza naturalny bieg programu. Np. znane z AVR przerwanie, które powoduje że procesor „przystaje” wykonywać aktualny program i przenosi się w zupełnie inne miejsce kodu (do procedury obsługi przerwania). Wyjątek to pojęcie ogólne. Przerwanie jest jednym z trzech rodzajów wyjątków (pozostałe dwa rodzaje to błędy i pułapki). W ramach pocieszenia powiem, że do momentu publikacji pierwszej wersji tego Poradnika, nie wiedziałem jaka jest w sumie różnica między przerwaniami a pozostałymi rodzajami wyjątków. Ba! Nie wiedziałem, że istnieje coś takiego jak pułapka... i ani trochę nie przeszkadzało mi to w pisaniu prostych programów :)

Po wystąpieniu wyjątku (np. przerwania) rdzeń automatycznie przechodzi do trybu *handler*. Wszystkie funkcje obsługi wyjątków pracują w tym trybie. Kiedy rdzeń nie obsługuje wyjątku to wraca do trybu *thread*. Podsumowując sprawę:

- *handler mode* - obsługa wyjątków (np. przerwań)
- *thread mode* - wszystko co nie jest wyjątkiem (np. funkcja *main*)

Różnice między trybami *thread* i *handler* są dwie:

86 „*Bądź mężczyzną, młodzieńcze!*”

- w trybie obsługi wyjątków program zawsze ma poziom uprzywilejowany (*privileged*)
- w trybie obsługi wyjątków program zawsze korzysta z głównego stosu⁸⁷

W trybie użytkownika (*thread*) poziom uprzywilejowania i wykorzystywany stos można zmienić w rejestrze specjalnym rdzenia: *CONTROL*. Modyfikacja tego rejestru jest możliwa tylko przez kod działający w trybie uprzywilejowanym poprzez wykorzystanie specjalnych rozkazów.

Gdy rdzeń pracuje w trybie nieuprzywilejowanym to ma ograniczony dostęp do:

- instrukcji operujących na rejestrach specjalnych (*mrs*, *msr*) np. wskaźnikach stosu, rejestrze *CONTROL*, rejestrach związanych z obsługą wyjątków
- nie może używać instrukcji *cps* (operacje na rejestrach specjalnych *FAULT* i *PRIMASK*)
- nie ma dostępu do rejestrów konfiguracyjnych peryferii rdzenia (NVIC, SCB, SysTick)
- może mieć ograniczony dostęp do pamięci i peryferiów

W trybie uprzywilejowanym program ma generalnie rzecz ujmując dostęp do wszystkiego i może wszystko.

Rozdzielenie stosów (*main stack* i *process stack*) i całe te tryby uprzywilejowania to ukłon w stronę systemów operacyjnych. System operacyjny ma swój stos i większe uprawnienia niż zwykły wątek. Chodzi o to aby wadliwa „aplikacja” nie rozłożyła całego systemu – a przynajmniej miała trudniej i piaskiem po oczach.

Jeżeli nie korzystamy z systemu operacyjnego to proponuję nie ruszać poziomu uprzywilejowania (zostawić domyślny - uprzywilejowany) oraz nie korzystać z podziału na dwa stosy, tak aby wyjątki i główna funkcja korzystały z jednego stosu (domyślne zachowanie procesora).

Co warto zapamiętać z tego rozdziału?

- wyjątek to ogólne pojęcie określające „coś” co powoduje zmianę w przepływie programu (skok do procedury obsługi wyjątku), przerwanie jest jednym z rodzajów wyjątków
- jeśli nie korzystamy z systemów operacyjnych to rozdzielenie stosów i zmiana poziomu uprzywilejowania rdzenia nie są nam potrzebne
- domyślnie po *resetie* procesor jest na poziomie uprzywilejowanym a podział stosów jest wyłączony

⁸⁷ stosy są dwa: *main stack* i *process stack*

- nic nie musimy zmieniać (przy czym nie gwarantuję, że posiadane środowisko nie włącza np. podziału stosów w procedurze startowej - to już każdy musi sobie sam zweryfikować)
- *handler mode* służy do obsługi wyjątków, po powrocie do „zwykłego kodu” procesor przechodzi do *thread mode*

5.2. Poznajmy wyjątki w Cortexie

Pojęcie wyjątku obejmuje wszystko, co zaburza naturalny bieg programu. Przykładowo może to być przerwanie, które powoduje że procesor przerywa to co aktualnie robi i skacze w inne miejsce kodu (do procedury obsługi przerwania - ISR). Są trzy rodzaje wyjątków:

- przerwanie
- błąd
- pułapka

Z naszego punktu widzenia, najciekawsze są przerwania. Przerwania mogą pochodzić od:

- układów peryferyjnych mikrokontrolera jak w AVR (np. od interfejsów komunikacyjnych, przetworników ADC, nóżek mikrokontrolera, ...)
- układów rdzenia (licznik SysTick)
- mogą być wywoływanie z premedytacją programowo (przerwanie PendSV)

Przerwania różnią się od pozostałych rodzajów wyjątków (błędów i pułapek) w dwóch zasadniczych kwestiach:

- nie muszą być obsłużone od razu - przerwanie może być np. wyłączone (jak w AVR - cli) lub mieć zbyt niski priorytet (patrz rozdział 5.4), w efekcie nie jest obsługiwane od razu po zgłoszeniu tylko przechodzi w stan oczekiwania
- zgłoszenie przerwania nie wynika w sposób bezpośredni z tego co się dzieje aktualnie w programie (przerwania są asynchroniczne) - np. zgłoszenie przerwania zegarowego od licznika nie ma najczęściej nic wspólnego z rozkazami, które aktualnie wykonuje procesor

Pozostałe dwa rodzaje wyjątków są synchroniczne, czyli wynikają bezpośrednio z kodu programu (działania procesora). Np. próba wykonania dzielenia przez zero może wywołać

wyjątek - błąd - który będzie powiązany z konkretnym rozkazem procesora (dzieleniem). Ponadto błędy i pułapki powinny zostać obsłużone natychmiast. Jeżeli jest to niemożliwe (np. błąd jest wyłączony lub ma zbyt niski priorytet) to następuje zjawisko „eskalacji” błędu i wywoływany jest wyjątek HardFault. Błędami i pułapkami nie będziemy się zajmować szczegółowo, wyszlibyśmy zbyt daleko poza bezpieczny „playground” ogrodzony znakami „początkujący” :) Domyślnie wszystkie błędy (poza HardFault) są wyłączone, a przed wywołaniem sporej części z nich, skutecznie, chroni nas kompilator.

Dobra do rzeczy. W Cortexie mamy następujące wyjątki (w tabeli uwzględniono tylko te pochodzące od rdzenia, kompletna lista z wszystkimi przerwaniami od peryferiali mikrokontrolera znajduje się w RM):

Tabela 5.1 Wyjątki

Nazwa wyjątku	Priorytet	Opis (uproszczony, po szczegółów odsyłam do dokumentacji)
Reset	-3 (stały, najwyższy)	<i>Reset to reset (np. po włączeniu zasilania czy zadziałaniu watchdoga). Procesor pobiera adres stosu i początku kodu z tablicy wektorów, rejestrzy przyjmując domyślne wartości⁸⁸. Szczegóły w rozdziale 11.1.</i>
NMI	-2 (stały)	<i>Przerwania nie-maskowalne (Non Maskable Interrupts). Nie można ich wyłączyć! W STMach przerwanie NMI jest odpalone jeśli system kontroli sygnału zegarowego (Clock Security System) wykryje awarię zewnętrznego rezonatora (patrz zadanie 17.2) lub w przypadku wykrycia błędu bitu parzystości pamięci SRAM (dotyczy tylko F334).</i>
HardFault	-1 (stały)	<i>Błąd w obsłudze innego wyjątku lub jeśli inny błąd/pułapka nie może zostać obsłużony po zgłoszeniu (np. ma zbyt niski priorytet).</i>
MemManage	konfigurowalny	<i>Naruszenie zasad ochrony pamięci zdefiniowanych w bloku MPU lub próba wykonywania instrukcji spod adresów „niewykonywalnych” - np. z przestrzeni rejestrów peryferiów.</i>
BusFault	konfigurowalny	<i>Błąd magistrali, np. próba dostępu do wyłączonej pamięci zewnętrznej.</i>
UsageFault	konfigurowalny	<i>Nieznana instrukcja lub dzielenie przez 0⁸⁹.</i>
SVCall	konfigurowalny	<i>SuperVisor Call, wyjątek odpalany programowo (instrukcją svc)</i>
PendSV	konfigurowalny	<i>Jakieś czary mary przydatne przy korzystaniu z systemów operacyjnych</i>
SysTick	konfigurowalny	<i>Wyjątkuje jak zegarek systemowy (SysTick) zliczy do zera, takie przerwanie zegarowe.</i>
Interrupts	konfigurowalny	<i>Tu symbolicznie zawierają się wszystkie przerwania od peryferiów mikrokontrolera (liczników, interfejsów etc.) dołączonych do kontrolera NVIC. Pełna lista przerwań dostępna jest w Reference Manualu.</i>

Omówmy z grubsza o co chodzi. Tabelka 5.1 zawiera nazwy wyjątków, informacje o priorytecie i krótki opis. Jak widać spora część z nich to błędy (te z *Fault* w nazwie). Nie ma co

⁸⁸ z wyjątkiem rejestru PWR_CSR (który pozwala określić źródło resetu) i rejestrów podtrzymywanych baterijnie (*Backup Registers, RTC*)

⁸⁹ w zależności od stanu bitu SCB_CCR_DIV_0_TRP dzielenie przez 0 może zwracać 0 lub wywoływać wyjątek

panikować - większości z nich nie musimy się bać, bo piszemy w języku wysokiego poziomu (C) i pilnuje nas kompilator. Ponadto przypominam, że domyślnie wszystkie błędy poza HardFault są wyłączone (patrz rejestr SCB_SHCSR). Szansa na to, że w naszym programie pojawi się np. nieznany rozkaz jest... nikła. W 99% przypadków będziemy lądowali w *Faultcie* lub jakimś domyślnym *handlerze* wyjątku, ze względu na odpalenie przerwania dla którego nie napisaliśmy procedury obsługi, ewentualnie przesadzimy ze zmiennymi i wysypie się stos. Wyjątki *SVCall* i *PendSV* to znowu ukłon w stronę systemów operacyjnych i można o nich na razie zapomnieć.

To co nas będzie najbardziej interesowało to przerwania *zewnętrzne* (zewnętrzne z punktu widzenia rdzenia), czyli ostatni wiersz tabeli oraz przerwanie SysTick. Tutaj siedzą wszystkie przerwania od liczników, interfejsów komunikacyjnych i całej zgrai bloków peryferyjnych. Szczegółową listę przerwań mikrokontrolera można znaleźć w RM. Nie będę jej tu wstawiał całą bo jest... dłuższa. Za kontrolę nad przerwaniami od układów peryferyjnych odpowiada kontroler przerwań NVIC (*Nested Vectored Interrupt Controller*).

NVICa wyobrażam sobie jako takie wrota przez które przerwania od układów peryferyjnych mikrokontrolera docierają do rdzenia. Taki nadzorca ruchu... strażnik bramy. Za jego pomocą można, przede wszystkim, włączać przerwania i ustawać ich priorytety. W NVICu będziemy włączać tylko przerwania pochodzące od peryferiów mikrokontrolera! Przerwania od peryferiów rdzenia nie wymagają włączenia w NVICu (bo już są blisko rdzenia i nie przechodzą przez „*NVICowe wrota*”).

Uwaga! W przeciwieństwie do AVR, w Cortexie przerwania domyślnie są odblokowane (globalnie) po resecie! Nie ma potrzeby „włączania przerwań” tak jak miało to miejsce w AVR (instrukcją *sei*).

Zupełną nowością są priorytety wyjątków (druga kolumna tabeli). Szczegóły dotyczące priorytetów pojawią się w rozdziale 5.4. Na razie chciałbym zwrócić uwagę na to, że większość wyjątków ma *konfigurowalny priorytet*. Jedynie *reset*, *NMI* oraz *HardFault* mają stałą wartość priorytetu. Im ta wartość jest niższa, tym wyjątek ma wyższy priorytet (jest ważniejszy). Reset ma najwyższy priorytet -3.

Co warto zapamiętać z tego rozdziału?

- przerwania od peryferiów mikrokontrolera należy włączyć w kontrolerze NVIC
- przerwania od peryferiów rdzenia nie wymagają włączania w NVICu
- po resecie przerwania są globalnie „włączone” (i raczej nie ma potrzeby ich wyłączać)

- wyjątku od przerwań niemaskowalnych (NMI), jak sama nazwa wskazuje, nie można wyłączyć!
- *reset* ma najwyższy priorytet

5.3. Mechanika działania wyjątków

Każdy wyjątek ma określony stan:

- *active* - wyjątek jest aktualnie obsługiwany przez procesor (wykonuje się procedura obsługi wyjątku, np. ISR⁹⁰ przerwania). Uwaga na przyszłość: jeśli nastąpi wywłaszczenie wyjątku to oba (wywłaszczyony i wywłaszczający) będą w stanie *aktywnym*.
- *pending* (spodziewany/oczekujący) - wyjątek czeka na bycie obsłużonym przez procesor, np. jeśli procesor aktualnie obsługuje ważniejszy wyjątek lub jeśli pojawił się wyjątek ale nie jest on włączony (potem się wszystko wyjaśni) to oczekuje on w stanie *pending*
- *active and pending* - jeśli w trakcie trwania obsługi wyjątku pojawi się znowu ten sam wyjątek to będzie on naraz *aktywny i oczekujący*
- *inactive* - kiedy nie jest w żadnym innym stanie

Gdy pojawia się nowy wyjątek⁹¹ (np. przerwanie zewnętrzne) to przyjmuje stan *pending*. To co się dalej wydarzy zależy od priorytetu tego wyjątku, konfiguracji rdzenia i tego co rdzeń aktualnie obsługuje (priorytetu rdzenia). Założmy że rdzeń mieli coś w funkcji *main* a przerwania nie są w żaden sposób wyłączone. No więc pojawia się nasz nowy wyjątek w stanie *pending* - rozpoczyna się procedura wejścia w jego obsługę. Procesor odkłada (sprzętowo!) na stos strukturę składającą się z (nazywa się to *stacking*):

- rejestrów ogólnych: *r0, r1, r2, r3, r12*
 - rejestrów zawierającego adres powrotny z ISR: *LR*
 - licznika programu: *PC*
 - rejestrów statusowych: *PSR* (flagi *Zero, Carry, Overflow* itd...)
-

Uwaga - ciekawostka! To niepozorne **sprzętowe** odkładanie rejestrów oraz pewien szczwany myk z rejestrzem LR i wartością EXC_RETURN (do doczytania w dokumentacji rdzenia we własnym

90 *Interrupt Service Routine* - procedura obsługi przerwania

91 przypominam, że przerwania też są wyjątkami - jeśli dla kogoś jest już za dużo nowości, to proponuję aby tymczasowo myślał o każdym wyjątku jak o przerwaniu (takim jak znane z AVR)

zakresie) są szalenie zajebiste. Dlaczego? Bo dzięki temu procedura obsługi przerwania, która z natury jest wywoływaną asynchronicznie (znienacka), nie musi tego odkładać i zdejmować programowo. A to z kolei powoduje, że procedura obsługi przerwania przy rdzeniu Cortex-M niczym nie różni się od zwykłej procedury języka C! Nie potrzeba żadnych dodatkowych atrybutów, czy innych hocków-klocków⁹². Co więcej, procedurę obsługi przerwania można wywołać jak każdą inną zwyczajną funkcję z programu. Koniec OT.

Po zakończeniu *stackingu* (właściwie nawet równolegle z nim) procesor pobiera z tablicy wektorów adres procedury obsługi wyjątku. Gdy tylko *stacking* się zakończy, rozpoczyna się wykonywanie instrukcji z handlera wyjątku, np. ISR. Stan wyjątku zmienia się z *pending* na *active*. Opóźnienie od pojawienia się *tego co powoduje wyjątek* do rozpoczęcia wykonywania procedury jego obsługi wynosi maksymalnie 12 cykli jeśli nie ma dodatkowych opóźnień wynikających np. z zastosowania *Wait States* przy dostępie do pamięci (wyjaśni się w przyszłości).

Jeżeli w trakcie obsługi przerwania zostanie ono jeszcze raz wywołane (np. jeśli w trakcie wykonywania procedury obsługi przerwania zewnętrznego, pojawi się nowe przerwanie z tego źródła) to będzie miało jednocześnie stan *active* i *pending*. Gdy zakończy się obsługa przerwania (tego w stanie *active*) to dalej będzie utrzymany stan oczekiwania (*pending*) i procesor ponownie skoczy do tej samej ISR. Dzięki temu nie nastąpi „zgubienie” drugiego przerwania z tego samego źródła. W przypadku pozostałych rodzajów wyjątków (błędów i pułapek) ponowne zgłoszenie tego samego wyjątku w trakcie jego obsługi spowoduje eskalację do wyjątku HardFault (patrz rozdział 5.2).

Po zakończeniu obsługi wyjątków następuje powrót do „trybu użytkownika” (*thread mode*) - funkcji głównej jak ktoś woli. Przywracane są przy tym wartości rejestrów zapisanych podczas *stackingu*.

Procesor ma do dyspozycji kilka dodatkowych mechanizmów, zwykle niewidocznych dla programisty, przyspieszających obsługę wyjątków:

- *late-arriving* - późne przybycie - jeśli w czasie wchodzenia w obsługę jakiegoś wyjątku (np. podczas *stackingu*) pojawi się nowy wyjątek o wyższym priorytecie (ważniejszy), to procesor odrzuca ten stary wyjątek i nowy wyjątek „odziedzicza” procedurę wejściową (*stacking*) tak jakby była przygotowana z myślą o nim. Stary wyjątek zostaje w stanie oczekiwania i będzie obsłużony później.

92 jest tu mały haczyk, w ramach chuchania na zimne będziemy używać pewnego atrybutu → odsyłam do dodatku 5

- *tail-chaining* - ogon łańcuchowy (?) - jeśli po zakończeniu obsługi wyjątku, w kolejce są kolejne oczekujące wyjątki to procesor nie przeprowadza procedury „powrotowej” (zdjęcie rejestrów ze stosu) tylko od razu zaczyna obsługiwać kolejny wyjątek. Dzięki temu nie musi od nowa odkładać rejestrów na stosie, co zmniejsza opóźnienie obsługi przerwania.
- *preemption* - wywłaszczenie - to przerwanie obsługi wyjątku przez inny wyjątek, zostanie omówione w następnym rozdziale

Co warto zapamiętać z tego rozdziału?

- wyjątek zawsze jest w jednym ze stanów (*inactive, pending, active, active&pending*)
- Cortex-M ma kilka fajnych mechanizmów przyspieszających obsługę wyjątków (niewidocznych dla programisty)
- Cortex-M sprzętowo zachowuje stan niektórych rejestrów przy obsłudze wyjątków
- procedura obsługi przerwania/wyjątku nie różni się niczym od zwyczajnej funkcji

5.4. Priorytety i wywłaszczanie

Konfigurowalny priorytet wyjątku (przerwania) jest nowością w stosunku do AVRów. Każdy wyjątek (np. przerwanie) ma jakiś priorytet. Priorytet decyduje o tym czy:

- dany wyjątek może przerwać wykonywanie innego wyjątku (*wywłaszczyć*)
- w jakiej kolejności mają się wykonywać wyjątki jeśli pojawiły się jednocześnie
- czy wyjątek w ogóle zostanie obsłużony (czy ma wystarczający priorytet)

Uwaga! Można się zapłatać: **im niższa jest wartość priorytetu (liczba określająca priorytet) tym priorytet wyjątku jest wyższy (wyjątek jest „ważniejszy”)**. Reset ma najwyższy priorytet (-3)⁹³. Program może wyłączyć wszystkie wyjątki poza Resetem i NMI.

Wszystkie wyjątki poza Resetem, NMI i HardFaultem mają konfigurowalny (programowo) priorytet. Domyślnie, po resecie, priorytet jest równy 0.

⁹³ co jest dosyć logiczne - wyobraź sobie, że zwierasz nóżkę *reset* do masy a procesor to olewa bo właśnie obsługuje przerwanie o wyższym priorytecie :) makabra jakaś

Na początek ważna uwaga. Nie ma konieczności ruszania priorytetów i w większości prostych programów generuje to więcej kłopotów niż pozytku. Domyślnie po resecie wszystkie wyjątki (z konfigurowalnym priorytetem) mają taki sam priorytet (0) – wszystkie są równe, nie ma żadnych wywłaszczeń itp. spokój i nuda jak w AVR.

System priorytetów w Cortexie wygląda tak, że priorytet składa się z dwóch członów:

- priorytetu grupowego - *group priority* (czasem określany jako *preemption priority*)
- pod-priorytetu - *sub priority*

Priorytet grupowy decyduje o wywłaszczeniach. Tzn. jeśli w czasie trwania obsługi wyjątku (np. przerwania) pojawi się nowy o wyższym priorytecie grupowym to nastąpi wywłaszczenie. Wywłaszczenie jest to przerwanie wyjątku przez inny wyjątek (np. przerwanie przerwania przerwaniem). Czyli procesor przerywa wykonywanie procedury obsługi wyjątku o niższym priorytecie i skacze do tej od wyższego priorytetu. W AVRach wejście do procedury obsługi przerwania powodowało automatyczne „zablokowanie” możliwości wystąpienia innych przerwań (wszystkich). W przypadku Cortexa, wejście do procedury obsługi wyjątku blokuje jedynie wyjątki o priorytecie równym i niższym od aktualnie obsługiwanej. Dzięki temu możliwe jest wywłaszczenie przez inny, ważniejszy wyjątek.

Wejście do ISR w AVR też blokuje przerwania o równym/nizszym priorytecie, ale ponieważ wszystkie przerwania w AVR mają ten sam priorytet to efekt jest taki, że wszystkie zostają zablokowane.

Pod-priorytet decyduje tylko o kolejności obsłużenia wyjątków posiadających ten sam priorytet grupowy. Jeżeli kilka wyjątków o tym samym grupowym priorytecie oczekuje (*pendinguje*) to pod-priorytet decyduje o tym, w jakiej kolejności się wykonają. Nie ma on wpływu na wywłaszczanie.

Jeżeli wszystkie oczekujące wyjątki mają ustawiony identyczny priorytet, to o kolejności wykonania decyduje pozycja w tablicy wektorów. Odsyłam do tabeli *Vector Table* w RM. Najpierw zostają wykonane te wyjątki, które mają niższy numer (są wyżej). Podobnie działało to w AVR.

Proste, prawda? To jeszcze jedna informacja na koniec żeby nie było za prosto. Priorytet grupowy i pod-priorytet zakodowane są razem w jednej czterobitowej wartości. Sposób kodowania

określa wartość PRIGROUP w rejestrze SCB_AIRCR. Od niej zależy ile z tych czterech bitów będzie określać priorytet grupowy, a ile pod-priorytet.

Tabela 5.2 Podział wartości priorytetu

wartość PRIGROUP	sposób kodowania priorytetów ⁹⁴	liczba poziomów priorytetów grupowych	liczba poziomów pod-priorytetów
3	$0bGGGG$	16	brak
4	$0bGGGP$	8	2
5	$0bGGPP$	4	4
6	$0bGPPP$	2	8
7	$0bPPPP$	brak	16

Czyli przykładowo, jeśli ustawimy PRIGROUP na „5” to cztery bity kodujące priorytet zostają podzielone na pół: 2b na priorytet grupowy i 2b na pod-priorytet. W dwóch bitach można zapisać liczby od 0 do 3, czyli mamy cztery poziomy priorytetu grupowego i pod-priorytetu. Jeśli potrzebujemy więcej poziomów priorytetów grupowych to trzeba zmienić wartość PRIGROUP np. na „4”. Wtedy priorytet grupowy jest zapisany w trzech bitach. Trzy bity to zakres 0 - 7 czyli osiem poziomów priorytetów. Ale na pod-priorytet zostaje już tylko jeden bit!

Na koniec ważna uwaga! Dokumentacja IMHO nie preczyzuje jednoznacznie czy podział na priorytet grupowy i pod-priorytet dotyczy tylko przerwań od peryferiów mikrokontrolera czy też wyjątków pochodzących od rdzenia, które nie przechodzą przez kontroler NVIC (np. układ SysTick). Na Elektrodzie można znaleźć ciekawy wątek poświęcony temu zagadnieniu (*[Cortex] NVIC Priorytety przerwań*). Polecam lekturę. Osobiście pozwoliłem sobie porobić kilka testów, z których wynika że podział priorytetów dotyczy wszystkich wyjątków o konfigurowalnych priorytecie, bez różniczy czy pochodzą od peryferiów mikrokontrolera czy od rdzenia. I tego będę się trzymał.

Z priorytetami jest związany jeszcze jeden ciekawy mechanizm. Możliwe jest mianowicie zablokowanie wyjątków do pewnego priorytetu. Czyli np. kiedy procesor robi coś ważnego, to można zablokować mało ważne przerwania - będą one wtedy oczekiwane aż ban zostanie zniesiony. Służy do tego rejestr specjalny BASEPRI. Pozwala on ustawić minimalny priorytet jaki musi mieć wyjątek, aby mógł zostać obsłużony. Taka ciekawostka.

Przypominam, że jedną z cech błędów i pułapek jest to, że powinny zostać obsłużone od razu po zgłoszeniu. Należy to mieć na uwadze przy konfigurowaniu priorytetów wyjątków, jeśli zamierzamy korzystać z błędów lub pułapek. Jeżeli priorytet błędu/pułapki nie pozwoli na

94 G - bit kodujący priorytet grupowy; P - bit kodujący pod-priorytet

natychmiastowe obsłużenie (będzie równy lub niższy od aktualnego priorytetu procesora), to nastąpi eskalacja błędu i wygenerowanie błędu HardFault (o stałym priorytecie równym -1).

Co warto zapamiętać z tego rozdziału?

- w prostych programach nie ma potrzeby zmieniania priorytetów wyjątków
- na priorytet wyjątku składa się priorytet grupowy i pod-priorytet
- priorytet grupowy decyduje o wywłaszczeniu
- pod-priorytet decyduje o kolejności wykonania oczekujących wyjątków o tym samym priorytecie grupowym
- po wejściu do handlera wyjątku (np. ISR) blokowane są tylko wyjątki o równym i niższym priorytecie
- wyjątek o wyższym priorytecie może przerwać (wywłaszczyć) aktualnie obsługiwany handler (nie jest do tego konieczne jakieś dodatkowe „odblokowywanie przerwań” jak w AVR)

5.5. Funkcje pomocnicze

Do konfigurowania wyjątków ARM przygotował nam zabawki w postaci kilku funkcji dostępnych w CMSIS. Zebrałem je w tabeli 5.3. Sporo tego jest. Zawsze się gubię w tych funkcjach od priorytetów. Dobra wiadomość jest taka, że jeśli nie będziemy ruszać priorytetów ani specjalnie cudować, to do podstawowej obsługi przerwań, z całej tej listy przyda nam się tylko:

- włączanie/wyłączenie przerwania w NVICu:
 - `void NVIC_EnableIRQ(IRQn_t)`
 - `void NVIC_DisableIRQ(IRQn_t)`
- kasowanie oczekującego przerwania:
 - `void NVIC_ClearPendingIRQ(IRQn_t)`

Jeśli uprzemys się na zabawę z priorytetami to dojdą trzy funkcje:

- funkcja ustawiająca podział bitów priorytetu na grupę i pod-priorytet:
 - `void NVIC_SetPriorityGrouping(uint32_t)`
- funkcja kodująca wartości priorytetu grupowego i pod-priorytetu w jedną liczbę:
 - `uint32_t NVIC_EncodePriority (. . .)`
- funkcja przypisująca zakodowany priorytet konkretnemu wyjątkowi... wyjątku:

- o `void NVIC_SetPriority(IRQn_t IRQn, uint32_t priority)`

O reszcie funkcji wystarczy pamiętać, że coś takiego było i gdzie je znaleźć. Opis funkcji znajdziemy np. w *Programming Manualu*, źródła natomiast w plikach *core_cmx.h* i *core_cmFunc.h*.

Tabela 5.3 Funkcje pomocnicze do obsługi wyjątków i przerwań

funkcja	opis
<code>void __enable_irq(void)</code>	włączenie i wyłączenie wyjątków z konfigurowalnym priorytetem (nie łąpią się: Reset, NMI, HardFault); trochę jak <i>cli</i> i <i>sei</i> z AVR
<code>void __disable_irq(void)</code>	
<code>void __enable_fault_irq(void)</code>	jak wyżej ale obejmuje również wyjątek HardFault
<code>void __disable_fault_irq(void)</code>	
<code>uint32_t __get_BASEPRI (void)</code>	funkcje pobierające i ustawiające wartość rejestru określającego minimalny priorytet jaki musi mieć wyjątek aby został obsłużony (rejestr BASEPRI)
<code>void __set_BASEPRI(uint32_t val)</code>	
<code>void NVIC_EnableIRQ(IRQn_t⁹⁵)</code>	funkcje włączające i wyłączające konkretne przerwanie w kontrolerze NVIC; w kontrolerze
<code>void NVIC_DisableIRQ(IRQn_t⁹⁵)</code>	włącza się tylko przerwania od peryferiów mikrokontrolera
<code>uint32_t NVIC_GetPendingIRQ(IRQn_t⁹⁵)</code>	funkcja zwraca 1 jeśli przerwanie podane w argumencie jest w stanie oczekującym (pending) w kontrolerze NVIC
<code>void NVIC_SetPendingIRQ(IRQn_t⁹⁵)</code>	funkcja powoduje przejście przerwania podanego w argumencie do stanu oczekiwania (ustawienie flagi pending w kontrolerze NVIC)
<code>void NVIC_ClearPendingIRQ(IRQn_t⁹⁵)</code>	funkcja powoduje skasowanie flagi oczekiwania (pending) przerwania podanego w argumencie, w kontrolerze NVIC
<code>uint32_t NVIC_GetActive(IRQn_t⁹⁵)</code>	zwraca 1 jeśli przerwanie podane w argumencie jest w stanie aktywnym
<code>void NVIC_SetPriorityGrouping(uint32_t)</code>	funkcje ustawiające i pobierające wartość rejestru ustalającego sposób podziału priorytetu na priorytet grupowy i pod-priorytet (rejestr PRIGROUP)
<code>uint32_t NVIC_GetPriorityGrouping(void)</code>	
<code>uint32_t NVIC_EncodePriority (uint32_t PRIGROUP, uint32_t GroupPrio, uint32_t SubPrio)</code>	funkcja, uwzględniając podział wartości priorytetu za pomocą PRIGROUP, oblicza wynikową wartość priorytetu dla podanego priorytetu grupowego i pod-priorytetu i zwraca tą wartość; nic nie modyfikuje w rejestrach konfiguracyjnych!
<code>void NVIC_DecodePriority(uint32_t Priority, uint32_t PRIGROUP, uint32_t* GroupPrio, uint32_t* SubPrio)</code>	funkcja wyluskuje z podanego priorytetu wartość priorytetu grupowego i pod-priorytetu (uwzględniając podział priorytetu za pomocą PRIGROUP); nic nie modyfikuje w rejestrach!
<code>void NVIC_SetPriority(IRQn_t IRQn⁹⁵, uint32_t priority)</code>	funkcja ustawia priorytet (zakodowany np. za pomocą funkcji NVIC_EncodePriority) przerwania podanego w argumencie
<code>uint32_t NVIC_GetPriority(IRQn_Type IRQn⁹⁵)</code>	funkcja zwraca priorytet (który potem można odkodować funkcją NVIC_DecodePriority) przerwania podanego w argumencie

To co opisałem zdecydowanie nie wyczerpuje tematu przerwań! Ale na razie wystarczy. Cóż, standardowo... RTFM!

Co warto zapamiętać z tego rozdziału?

- CMSIS dostarcza nam zestaw zabawek do obsługi wyjątków/przerwań
- w szczególności przydadzą się funkcje:

⁹⁵ Funkcja przyjmuje w parametrze numer przerwania. W pliku nagłówkowym mikrokontrolera zdefiniowane są symboliczne nazwy przerwań (np. przerwanie od licznika TIM2 nazywa się: TIM2_IRQHandler)

- void NVIC_EnableIRQ(IRQn_t)
- void NVIC_DisableIRQ(IRQn_t)
- void NVIC_ClearPendingIRQ(IRQn_t)

5.6. Priorytety przykład praktyczny

Uwaga! Zaraz będzie przykładowy kod obrazujący zabawę priorytetami w praktyce. Niestety żeby móc bawić się priorytetami przerwań, trzeba wykorzystać różne przerwania. Żeby mieć różne przerwania, trzeba znać różne układy peryferyjne. Żeby móc omówić dogłębnie różne układy peryferyjne, trzeba co nieco wiedzieć o przerwaniach. Kółko się zamyka. W tym przykładzie zostanie wykorzystana „wiedza” z późniejszych rozdziałów. Także ten. Proszę się skupić tylko na priorytetach a resztę przyjąć na wiarę. Zdecydowanie polecam w ogóle na razie opuścić ten przykład i wrócić do niego po skończonej lekturze dalszych rozdziałów Poradnika :)

Zadanie domowe 5.1: niechaj w programie będą trzy przerwania: dwa zegarowe (SysTick i jakiś licznik) oraz zewnętrzne (wyzwalane przyciskiem). Przerwania zegarowe mają machać dwoma ledami. W przerwaniu zewnętrznym ma być pętla nieskończona, która nic nie robi. System priorytetów ma być skonfigurowany tak aby były cztery priorytety grupowe i cztery podpriorytety. Priorytety grupowe mają być ustalone tak aby przerwanie licznika mogło wywąszczyć przerwanie zewnętrzne, zaś przerwanie zewnętrzne mogło wywąszczyć przerwanie SysTicka. Podpriorytety wedle uznania. Czas start!

Przykładowe rozwiązanie (F103, diody na PB0 i PB1, przerwanie zewnętrzne od PC13):

```
1. #define PRIGROUP_16G_0S          ((const uint32_t) 0x03)
2. #define PRIGROUP_8G_2S          ((const uint32_t) 0x04)
3. #define PRIGROUP_4G_4S          ((const uint32_t) 0x05)
4. #define PRIGROUP_2G_8S          ((const uint32_t) 0x06)
5. #define PRIGROUP_0G_16S          ((const uint32_t) 0x07)
6.
7. int main(void) {
8.
9.     RCC->APB2ENR = RCC_APB2ENR_IOPBEN | RCC_APB2ENR_IOPCEN | RCC_APB2ENR_AFIOEN;
10.    RCC->APB1ENR = RCC_APB1ENR_TIM3EN;
11.    gpio_pin_cfg(GPIOB, PB0, gpio_mode_output_PP_2MHz);
12.    gpio_pin_cfg(GPIOB, PB1, gpio_mode_output_PP_2MHz);
13.    gpio_pin_cfg(GPIOC, PC13, gpio_mode_input_pull);
14.    GPIOC->ODR = PC13;
15.    exti_cfg(GPIOC, PC13, exti_mode_fallingEdge_interrupt);
16.
17.    TIM3->PSC = 8000-1;
18.    TIM3->ARR = 500-1;
19.    TIM3->DIER = TIM_DIER_UIE;
20.    TIM3->EGR = TIM_EGR_UG;
21.    TIM3->CR1 = TIM_CR1_CEN;
22.
23.    SysTick_Config(8000000/6);
24.
25.    NVIC_SetPriorityGrouping(PRIGROUP_4G_4S);
26.    uint32_t prio;
27.
28.    prio = NVIC_EncodePriority(PRIGROUP_4G_4S, 1, 0);
29.    NVIC_SetPriority(TIM3_IRQn, prio);
30.
31.    prio = NVIC_EncodePriority(PRIGROUP_4G_4S, 2, 0);
32.    NVIC_SetPriority(EXTI15_10_IRQn, prio);
33.
34.    prio = NVIC_EncodePriority(PRIGROUP_4G_4S, 3, 0);
35.    NVIC_SetPriority(SysTick_IRQn, prio);
36.
37.    NVIC_EnableIRQ(EXTI15_10_IRQn);
38.    NVIC_EnableIRQ(TIM3_IRQn);
39.
40.    while (1);
41.
42. } /* main */
43.
44. __attribute__((interrupt)) void TIM3_IRQHandler(void){
45.     if (BB(TIM3->SR,TIM_SR UIF)){
46.         TIM3->SR = ~TIM_SR UIF;
47.         BB(GPIOB->ODR, PB0) ^=1;
48.     }
49. }
50.
51. __attribute__((interrupt)) void EXTI15_10_IRQHandler(void){
52.     if(BB(EXTI->PR, EXTI_PR_PR13)){
53.         EXTI->PR = EXTI_PR_PR13;
54.         while(1);
55.     }
56. }
57.
58. __attribute__((interrupt)) void SysTick_Handler(void){
59.     BB(GPIOB->ODR, PB1) ^=1;
60. }
```

1 - 5) kilka definicji dla funkcji *NVIC_SetPriorityGrouping()*. Niestety w plikach nagłówkowych nie ma wygodnych (czytelnych dla człowieka) definicji wartości argumentu tej funkcji... więc napisałem sobie sam :) Przypominam, że ta funkcja konfiguruje sposób podziału priorytetu na grupowy i pod-priorytet. Definicje, mam nadzieję, są czytelne i intuicyjne. Przykładowo: *PRIGROUP_2G_8S* to dwa poziomy grupowe i osiem poziomów podpriorytetu.

9 - 15) włączenie wykorzystywanych bloków mikrokontrolera (więcej w rozdziale 17), konfiguracja portów i przerwania zewnętrznego (więcej w rozdziałach 3 oraz 7)

17 - 21) konfiguracja licznika tak aby generował przerwanie zegarowe co około 0,5s (więcej w rozdziale 8.2)

23) włączenie SysTicka (patrz rozdział 6)

25) to nie powinno budzić wątpliwości - konfiguruje sposób podziału priorytetu na priorytet grupowy i podpriorytet. Wykorzystuję przy tym definicje z początku listingu.

28) funkcja *NVIC_EncodePriority()* przyjmuje trzy argumenty:

- sposób podziału priorytetów na grupowy i podpriorytet (PRIGROUP_2G_8S)
- wartość priorytetu grupowego (1)
- wartość podpriorytetu (0)

na podstawie tych danych funkcja oblicza wartość priorytetu i ją zwraca. Wartość ta jest następnie wpisywana do odpowiedniego rejestru konfiguracyjnego (w następnej linijce). Przypominam, że im niższa wartość priorytetu, tym wyższy priorytet (ważniejszy). Wyliczenie i ustawienie priorytetów powtarzam dla każdego z wykorzystywanych przerwań (licznik, przerwanie zewnętrzne, SysTick).

37, 38) włączenie przerwań od układów peryferyjnych w kontrolerze NVIC

44 - 49) procedura obsługi przerwania licznika (miganie diodą na PB0)

51 - 56) procedura obsługi przerwania zewnętrznego (pętla nieskończona)

58 - 60) procedura obsługi przerwania licznika SysTick (miganie diodą na PB1)

Zastanówmy się czego oczekujemy od tego programu. Po włączeniu pracują dwa liczniki (TIM3 i SysTick). Oba generują przerwania zegarowe. W obu przerwaniach migana jest dioda. Działa? Działa. Intrygująco robi się po wyzwoleniu przerwania zewnętrznego. W przerwaniu jest pętla nieskończona. Procesor nigdy nie skończy tej procedury i nie wróci do mejna. Gdyby nie priorytety i wywłaszczenie przerwań to na tym sprawa by się zakończyła. Procesor utknąłby w tej pętli do końca... końca czegoś. W ramach testu proponuję uruchomić ten przykład z zakomentowaną konfiguracją priorytetów. Wtedy wszystkie przerwania będą miały równy priorytet grupowy i nie będzie wywłaszczeń (jak w AVR). Procesor utknie, diody przestaną migać i tyle.

Ale nas interesuje ciekawszy przypadek, czyli z wywłaszczeniem. Wracamy do naszego kodu. SysTick ma niższy priorytet (wyższa liczba określająca priorytet) niż przerwanie zewnętrzne. Czyli przerwanie SysTicka nie będzie mogło wywłaszczyć (przerwać) procedury przerwania zewnętrznego. Licznik TIM3, z kolei, ma wyższy priorytet. Czyli przerwanie licznika będzie mogło wywłaszczyć przerwanie zewnętrzne. W efekcie spodziewamy się, że dioda SysTickowa się

zatrzyma, zaś druga dioda będzie dalej migać. Żeby nie przedłużać niepewności spieszę z informacją, że tak też się dzieje :)

Jeszcze raz dla utrwalenia. Procesor siedzi w przerwaniu zewnętrznym (bo pętla nieskończona). Pojawia się przerwanie SysTicka. Nie może ono wywalczyć przerwania zewnętrznego ze względu na niższy priorytet grupowy, więc się nie wykonuje (dioda nie mig). Przerwanie od licznika ma wyższy priorytet, więc przerywa procedurę obsługi przerwania zewnętrznego (dioda mig). Po zakończeniu procedury obsługi przerwania licznika TIM3 procesor schodzi „poziom niżej” czyli wraca do przerwania zewnętrznego. Poziomów zagnieżdżeń może być oczywiście więcej. Trzeba się odzwyczaić od AVRowego „schematu startowego” gdzie podstawą programu była pętla główna a przerwania z rzadka ją przerywały i miały być jak najkrótsze... bo inaczej to czarna wołga przyjedzie. A odblokowanie przerwań w przerwaniu (w AVR) to już w ogóle mogła na miejscu. Przy okazji STMów po raz pierwszy spotkałem się z podejściem, w którym program w ogóle nie powinien mieć pętli głównej (!) Tylko przerwania! Na początku brzmiało jak herezje... ale z czasem mi się nawet spodobało. Koniec OT.

Co warto zapamiętać z tego rozdziału?

- należy pamiętać, żeby wrócić do tego rozdziału jak już się opanuje resztę Poradnika :)

6. LICZNIK SYSTEMOWY SYSTICK („*MAGNUM OPUS*”⁹⁶)

6.1. Blink me baby one more time

Po poprzednich, dosyć zawiłych rozdziałach, czas na coś przyjemniejszego i krótkiego dla odprężenia. Panie i Panowie a oto i *SysTick*. SysTick jest układem peryferyjnym rdzenia. Jego dokumentacji należy poszukiwać w *Programming Manualu* (lub innym dokumencie opisującym rdzeń). Jest to 24-bitowy, bardzo prosty, licznik (timer). To *bardzo prosty* oznacza między innymi że:

- może być taktowany tylko sygnałem zegarowym, nie może zliczać np. impulsów z nóżki mikrokontrolera
- może zliczać tylko od zadanej wartości początkowej w dół do zera... i tak w kółko
- przy przekręcaniu się przez zero może zgłaszać przerwanie
- nie ma żadnych bajerów typowych dla zwykłych liczników (PWM, bloki Capture/Compare, itd...)

I to właściwie cała charakterystyka SysTICKa. Wykorzystać go można przede wszystkim do generowania przerwań zegarowych. A co z tymi przerwaniami zrobimy to już nasza sprawa.

Czemu więc SysTick a nie zwykły timer? Mogę tylko pogdybać: w każdym programie przydaje się przerwanie odpalone co określony okres (np. 10ms). Szkoda by było marnować cały timer na coś tak trywialnego. Cortex w wielu miejscach „sprzyja” systemom operacyjnym, SysTick wydaje się być jednym z takich miejsc. Zresztą chyba stąd wziął swoją nazwę SysTick (*System Tick*). Można na nim oprzeć mechanizm przełączania wątków systemu operacyjnego. SysTick występuje w każdym Cortexie, bez względu na to jaki producent wsadził go do swojego mikrokontrolera, co ułatwia przenoszenie kodu systemu. Pewnie jeszcze coś ważnego wynika też z tego, że SysTick jest układem rdzenia... Ale na naszym etapie SysTick to prosty licznik do generowania przerwania zegarowego i tyle.

Do uruchamiania SysTICKa ARM przygotował nam bardzo przyjemną funkcję (źródło w pliku core_cmx.h):

```
uint32_t SysTick_Config(uint32_t ticks)
```

96 „Wielkie dzieło.”

Funkcja jako argument przyjmuje ilość ticków (cykli zegara) do przepełnienia licznika. Jej działanie jest następujące (zachęcam oczywiście do własnej analizy źródła i dokumentacji przed dalszą lekturą):

- sprawdza czy podana ilość ticków nie przekracza rozdzielcości licznika (24bit) – jeśli tak to kończy działanie i zwraca 1
- ładuje zadaną ilość ticków do rejestru przeładowania licznika (to jest ta wartość od której licznik będzie zliczał do zera)
- ustawia priorytet przerwania SysTicka na najniższy z możliwych
- zeruje wartość rejestru licznika
- włącza generowanie przerwań przez SysTick
- wyłącza preskaler SysTicka (SysTick w STMie może być taktowany z częstotliwością AHB lub AHB/8⁹⁷)
- włącza wreszcie sam licznik i zwraca 0

Oczywiście w razie potrzeby można sobie zmienić konfigurację czy napisać własną funkcję jeśli coś nam nie odpowiada. Przy czym sugerowałbym aby oryginalnej funkcji z biblioteki nie ruszać, niech biblioteka pozostanie niezmieniona. Dosyć gadania, jedziem z kodem:

Zadanie domowe 6.1: migający led oparty o przerwanie systemowe od SysTicka.

Przykładowe rozwiązanie (F103, dioda na PB0):

```
1. int main(void){  
2.     RCC->APB2ENR = RCC_APB2ENR_IOPBEN;  
3.     gpio_pin_cfg(GPIOB, PB0, gpio_mode_output_PP_2MHz);  
4.     SysTick_Config(8000000 * 0.5);  
5.     while(1);  
6. } /* main */  
7.  
8.  
9.  
10. _attribute_((interrupt)) void SysTick_Handler(void){  
11.     BB(GPIOB->ODR, PB0) ^= 1;  
12. }  
13.  
14.  
15. }
```

Mówiłem, że będzie proste! Na PB0 jest śliczny prostokąt 1Hz, 50% wypełnienia (zrzut z analizatora dla wzrokowców):

⁹⁷ wszystko się wyjaśni później - w rozdziale o systemie zegarowym RCC → o tu: 17



Rys. 6.1. Przerwanie zegarowe

Prześledźmy kod:

3, 4) włączenie zegara dla portu B i konfiguracja pinu PB0. Zegara SysTicka włączać nie trzeba, bo to układ rdzenia a nie peryferial mikrokontrolera.

6) wywoływana jest funkcja konfigurująca licznik SysTick. Jej argumentem jest liczba taktów zegara, które mają być zliczone między wystąpieniami przerwania. Założyłem, że dioda ma migać z częstotliwością 1Hz, stan wysoki ma trwać 0,5s, niski tyleż samo. Dioda jest przełączana w przerwaniu, czyli przerwanie powinno występować co 0,5s. Mikrokontroler F103 pracuje z domyślną częstotliwością 8MHz⁹⁸. Czyli jeden tick zegara trwa $1/8\text{MHz} = 125\text{ns}$. Nasze 0,5s to będzie w takim razie $500\text{ms}/125\text{ns} = 4\ 000\ 000$ ticków. Voila.

13) tu zaczyna się procedura obsługi przerwania. Interesujące są dwie sprawy:

- niedawno pisałem, że Cortex-M jest fajny bo ISR to zwykła funkcja i nie potrzeba żadnych atrybutów a tu jednak... odsyłam do dodatku 5
- nazwa funkcji **musi być identyczna** jak w tablicy wektorów (to, gdzie należy szukać tablicy zależy od używanego środowiska i jego konfiguracji)

Poza tym, procedura przerwania jest banalna. Zawiera tylko zmianę stanu pinu PB0. Do zmiany stanu pinu wykorzystałem bit banding, ale oczywiście nie jest to konieczne i równie dobrze możliwe klasycznie:

```
GPIOB->ODR ^= PB0;
```

Przypominam, że definicje PB0, PB1, itd nie występują domyślnie w plikach nagłówkowych. To moje dzieło, patrz dodatek 1. W plikach nagłówkowych są definicje z pełną nazwą rejestru, np. GPIO_ODR_ODR0... ale to za dużo pisania jak dla mnie :)

⁹⁸ Mówiłem o tym wcześniej? Coś mi się wydaje że chyba nie... ale na pewno mówiłem, że trzeba ćwiczyć samodzielne poszukiwanie informacji!

Zadanie domowe 6.2: doczytać o rejestrze SysTick_CALIB i wartości TENMS

Zadanie domowe 6.3: napisać program migający diodą, oparty o przerwanie SysTicka, dla F429

Przykładowe rozwiązanie (F429, dioda na PG13):

```
1. int main(void){  
2.     RCC->AHB1ENR |= RCC_AHB1ENR_GPIOGEN;  
3.     __DSB();  
4.  
5.     gpio_pin_cfg(GPIOG, PG13, gpio_mode_out_PP_LS);  
6.     SysTick_Config(16000000/2);  
7.     while(1);  
8.  
9.  
10. }  
11.  
12. void SysTick_Handler(void){  
13.     BB(GPIOG->ODR, PG13) ^= 1;  
14. }
```

3) włączenie zegara. Zwróć uwagę na to, że zastosowałem sumę bitową a nie przypisanie. To dlatego, że rejestr RCC_AHB1ENR domyślnie nie jest równy zero. Coś tam jest włączone (chyba pamięć CCM), a ja nie chciałem tego wyłączać :)

4) o tym już mówiłem, o tu: [DSB w F4](#)

7) wywołanie funkcji konfigurującej SysTick, STM32F429 domyślnie działa z prędkością 16MHz

12) tu już nie ma żadnego atrybutu dla przerwania (zabawa z atrybutem dotyczyła tylko F103)

Zadanie domowe 6.4: wyjaśnić jak działa poniższy program migający diodą:

Migająca dioda do analizy:

```
1. int main(void) {  
2.  
3.     RCC->APB2ENR = RCC_APB2ENR_IOPBEN;  
4.     gpio_pin_cfg(GPIOB, PB0, gpio_mode_output_PP_2MHz);  
5.  
6.     SysTick->LOAD = 4000000-1;  
7.     SysTick->CTRL = SysTick_CTRL_CLKSOURCE_Msk | SysTick_CTRL_ENABLE_Msk;  
8.  
9.     while (1){  
10.         while( ! (SysTick->CTRL & SysTick_CTRL_COUNTFLAG) );  
11.         GPIOB->ODR ^= PB0;  
12.     }  
13.  
14. } /* main */
```

Zadanie domowe 6.5: napisać program migający diodą, oparty o przerwanie SysTicka, dla... kto zgadnie? F334 :]

Przykładowe rozwiązanie (F334, dioda na PA5):

```
1. int main(void){  
2.     RCC->AHBENR = RCC_AHBENR_GPIOAEN;  
3.     gpio_pin_cfg(GPIOA, PA5, gpio_mode_output_PP_LS);  
4.     SysTick_Config(8000000/4);  
5.     while(1);  
6.  
7.  
8.  
9. }  
10.  
11. void SysTick_Handler(void){  
12.     BB(GPIOA->ODR, PA5) ^= 1;  
13. }
```

Programik prościutki, prawda? Włączenie zegara dla portu A, konfiguracja wyprowadzenia, uruchomienie SysTicka (F334 domyślnie pracuje z częstotliwością 8MHz) i nieskończona pętla. W przerwaniu nóżka jest machana z wykorzystaniem bit bandingu. Zapuszczamy program i... tak zwana lipa. Program nie działa. Procesor ląduje w *default handlerze*. Dokładniej rzecz ujmując próbuje wylądować w błędzie *Bus Fault*, ale nie włączliśmy obsługi tego błędu (domyślnie błędy są wyłączone), więc przechodzi „oczko wyżej”, do błędu *Hard Fault*. Ale tu znowu odbija się od ściany, bo w naszym programie nie ma procedury obsługi tego błędu, więc zabawa kończy się w domyślnym handlerze wyjątków. Ale to tak na marginesie, jak ktoś nie pamięta to kłania się rozdział 5.2 :>

Oczywiście nas to nie zaskakuje. Doskonale wiemy, czemu tak się stało, nie? No powinniśmy wiedzieć, bo nie dawno o tym pisałem (a Ty o tym czytałeś :] Stop! Nie czytaj dalej, zastanów się. Popatrz na ten kod linijka po linijce i zastanów się, dlaczego pojawił się błąd. Co jest źle? Gwarantuję, że o tym pisałem. I to nie gdzieś na marginesie, w przypisie dolnym, czcionką 8. To był cały akapit i do tego jeszcze wypunktowanie :] Odpowiedź jest poniżej (napisana czcionką w kolorze tła, jeżeli chcesz odczytać to zaznacz i skopiuj do swojego ulubionego edytora tekstu; ale najpierw kombinuj sam!) [].

Co warto zapamiętać z tego rozdziału?

- SysTick to prosty licznik wykorzystywany do generowania przerwań zegarowych, realizacji opóźnień itp.
- SysTick jest elementem rdzenia ARM Cortex-M, w każdym mikrokontrolerze z takim rdzeniem będzie działał identycznie
- SysTick jest prosty :)

7. BLOK EXTI I PRZERWANIA ZEWNĘTRZNE („*FACTA SUNT VERBIS DIFFICILORA*”⁹⁹)

Ostatni temat był miły, prosty i sympatyczny, prawda? No to... kontynuujmy i wykorzystajmy zdobytą wiedzę w praktyce. Mamy opanowane GPIO i przerwania – jak to połączyć? EXTI (Extended Interrupt / Event Controller) czyli układ obejmujący m.in. przerwania zewnętrzne. Przyjemny, nieco zakręcony temat, na którym przećwiczymy obsługę przerwań od peryferiali mikrokontrolera (SysTick był elementem rdzenia).

Z góry proszę o nie palenie mnie na stosie za pomysł z obsługą przycisków za pomocą przerwań. Wiem, że to paskudne rozwiązanie, ale wygodne do zabawy. A to nie jest poradnik dobrego programowania.

7.1. EXTI (F103)

Przerwania zewnętrzne związane są z portami GPIO, czyli peryferium mikrokontrolera. Z tego względu odpalamy *Reference Manual*. Interesują nas dwa rozdziały: *Interrupts and events* oraz *GPIO*. Na początek trochę ogólnej teorii i ciekawostek.

Praktycznie każdy pin mikrokontrolera może być źródłem przerwania. Wyjątek stanowią nóżki „specjalne” typu BOOT, NRST, V_{BAT} itp. oraz współdzielone z oscylatorem zewnętrznym HSE¹⁰⁰. Jednocześnie można włączyć przerwanie **tylko od jednego pinu o tym samym numerze** - tzn. nie można naraz włączyć przerwania od PC5 i PE5 (ten sam numer pinu - 5). Każde przerwanie można osobno skonfigurować – czy ma reagować na zbocze opadające, narastające lub też oba zbocza. Pamiętasz moje porównanie NVICa do bramy wejściowej rdzenia? Przerwania zewnętrzne połączone są z NVICiem poprzez linie EXTI (*External Interrupt*). W sumie jest dwadzieścia linii przerwań zewnętrznych:

- *EXTI0* → wywoływanie przez piny o numerze 0 (PA0, PB0, PC0, ...)
- *EXTI1* → wywoływanie przez piny o numerze 1 (PA1, PB1, PC1, ...)
- ...
- *EXTI15* → wywoływanie przez piny o numerze 15 (PA15, PB15, PC15, ...)
- *EXTI16* – przerwanie związane na sztywno z blokiem *PVD* (*Power Voltage Detector*)
- *EXTI17* – przerwanie związane na sztywno z alarmem zegara *RTC*
- *EXTI18* – przerwanie związane na sztywno z *USBWake up Event*

99 „Czyny są trudniejsze niż słowa.”

100 dotyczy tylko mikrokontrolerów w małych (<100pin) obudowach, patrz RM rozdział *Using OSC_IN/OSC_OUT pins as GPIO ports PD0/PDI*

- *EXTI19* – przerwanie związane na sztywno z *Ethernet Wakeup Event* (tylko w mikrokontrolerach z linii *connectivity*)

Czyli np. piny PA7, PB7, PC7, PD7... mogą generować przerwanie na linii EXTI7, przy czym jednocześnie możnałączyć tylko przerwanie od jednego z nich.

Jak widać ostatnie linie EXTI nie są stricte przerwaniami zewnętrznymi. Dotyczą jakiś układów peryferyjnych... no ale tak to ktoś wymyślił i tak jest. Cóż począć.

Nie każda linia EXTI ma osobny wektor przerwania! Proszę popatrzyć do tablicy wektorów: linie 5..9 mają wspólny wektor, tak samo linie 10..15. Oznacza to, że jeśli będziemy mieli przerwanie na pinach PB5 (czyli linia EXTI5), PD7 (linia EXTI7) i PD8 (linia EXTI8) to wszystkie te przerwania mają wspólny wektor (wspólną procedurę obsługi). Będzie trzeba „ręcznie” w programie sprawdzić, co konkretnie wywołało przerwanie. Wiem że to się robi trochę pokręcone... luz, ja dalej się w tym kociokwiku czasem zapłczę :) Starczy tej teorii:

Zadanie domowe 7.1: na podstawie dotychczasowych informacji i *Reference Manuala*, napisać prosty program z dwoma przerwaniami zewnętrznymi (np. od przycisków). Niech jedno zapala a drugie gasi diodę. Czas start :) Tik, tik, tik... *Yes you can!*

Przykładowe rozwiązanie (F103, dioda na PB0, przyciski PB2 i PC13):

```

1. int main(void) {
2.
3.     RCC->APB2ENR = RCC_APB2ENR_IOPBEN | RCC_APB2ENR_IOPCEN | RCC_APB2ENR_AFIOEN;
4.
5.     gpio_pin_cfg(GPIOB, PB0, gpio_mode_output_PP_2MHz);
6.     gpio_pin_cfg(GPIOB, PB2, gpio_mode_input_floating);
7.     gpio_pin_cfg(GPIOC, PC13, gpio_mode_input_floating);
8.
9.     AFIO->EXTICR[0] = AFIO_EXTICR1_EXTI2_PB;
10.    AFIO->EXTICR[3] = AFIO_EXTICR4_EXTI13_PC;
11.
12.    EXTI->IMR = EXTI_IMR_MR2 | EXTI_IMR_MR13;
13.    EXTI->RTSR = EXTI_RTSR_TR2;
14.    EXTI->FTSR = EXTI_FTSR_TR13;
15.
16.    NVIC_EnableIRQ(EXTI2 IRQn);
17.    NVIC_EnableIRQ(EXTI15_10 IRQn);
18.
19.    while (1);
20.
21. } /* main */
22.
23. __attribute__((interrupt)) void EXTI2_IRQHandler(void) {
24.     if (EXTI->PR & EXTI_PR_PR2) {
25.         EXTI->PR = EXTI_PR_PR2;
26.         BB(GPIOB->ODR, PB0) = 1;
27.     }
28. }
29.
30. __attribute__((interrupt)) void EXTI15_10_IRQHandler(void) {
31.     if (EXTI->PR & EXTI_PR_PR13) {
32.         EXTI->PR = EXTI_PR_PR13;
33.         BB(GPIOB->ODR, PB0) = 0;
34.     }
35. }
```

Bolało? Eee tam. Grunt, że działa i jak na dłoni widać pięknie to o czym wcześniej zanudzałem w rozdziale 5:

3) włączenie zegarów portów B i C oraz **włączenie zegara dla funkcji alternatywnych portów** (blok AFIO trzeba włączyć przy korzystaniu z EXTI i remappingu; idę o zakład, że kilka razy o tym zapomnisz)

5, 6, 7) konfiguracja GPIO, przyciski mają zewnętrzne podciąganie stąd ustawiam jako wejścia pływające

Czas na magię:

9) AFIO_EXTICR odpowiada za wybór portu (A, B, C, ...), który będzie wyzwałał przerwanie na danej linii. Np. linia trzecia (EXTI3) może być wyzwalała przez pin portu A (PA3), portu B (PB3), itd. W tym rejestrze wybieramy właśnie „literkę” portu¹⁰¹. Przy czym nie jest to takie proste jakby się mogło wydawać :) Linii związań z portami GPIO jest w sumie 16 i konfiguracja nie mieści się w jednym rejestrze EXTICR. Takich rejestrów są w sumie 4szt. Każdy obejmuje konfigurację 4 linii:

- rejestr AFIO_EXTICR1 – linie 0..3
- rejestr AFIO_EXTICR2 - linie 4..7
- itd...

I tu jest haczyk¹⁰², proszę się skupić. W RM rejesty EXTICR numerowane są od 1 do 4. W pliku nagłówkowym natomiast są zebrane w tablicy. Tablica w języku C jest numerowana od 0! Stąd chcąc odwołać się do EXTICR1 trzeba wziąć pierwszy element tablicy, czyli element o indeksie 0. Myślę, że tabela trochę rozjaśni:

Tabela 7.1 Oznaczenia rejestrów EXTICR

oznaczenie rejestru w RM	definicja z pliku nagłówkowego	konfigurowane linie
EXTICR1	EXTICR[0]	0 - 3
EXTICR2	EXTICR[1]	4 - 7
EXTICR3	EXTICR[2]	8 - 11
EXTICR4	EXTICR[3]	12 - 15

101 piny portów dołączone są do multipleksera, z którego „wychodzi” sygnał EXTI; rejestr EXTICR steruje tym multiplekserem

102 „Trap for young players” jak powiedziałby D. Jones

No to wracamy do linii 9 listingu. Chcę skonfigurować przerwanie od PB2, czyli to będzie druga linia. Zgodnie z tabelką konfiguracja drugiej linii (EXTI2) siedzi w pierwszym rejestrze (EXTICR1). W programie odwołuję się do niego poprzez tablicę. Pierwszy element ma indeks zero, stąd zapis ...EXTICR[0]... wiem pokrecone. Dalej już jest prosto. Ustawiam w rejestrze bit: AFIO_EXTICR1_EXTI2_PB. Rozszyfrujmy jego nazwę:

- *AFIO_EXTICR1* - bo chodzi o bit rejestru EXTICR1 w bloku AFIO
- *EXTI2* - bo druga linia (PB2)
- *PB* - bo port B (PB2)

10) działania analogiczne jak wyżej. PC13 → linia 13 → rejestr EXTICR4 → indeks tablicy 3

W tym momencie mam ustawione, które porty będą generowały przerwania na liniach EXTI. Następny krok to konfiguracja sposobu wyzwalania poszczególnych linii (jakie zbocze) iłączenie wybranych linii.

12) rejestr IMR (*Interrupt Mask Register*) określa, które linie EXTI będą aktywne. Włączamy linie 2 i 13. Proste.

13, 14) teraz konfigurujemy czy przerwanie na danej linii ma być wywoływanie przez zbocze rosnące (jedynka w rejestrze RTSR¹⁰³), opadające (jedynka w rejestr FTSR¹⁰³) lub oba zbocza (dwie jedynki).

W tym momencie mamy już skonfigurowaną część „peryferyjną”, tzn. port i blok przerwań zewnętrznych. Tak jak wspominałem ([tu](#)) przerwania od peryferiali docierają do rdzenia poprzez wrota NVICowe. Czas więc skonfigurować kontroler NVIC tak, aby przepuścił nasze przerwanie do rdzenia:

16, 17) te funkcje już znamy, służą do włączenia przerwania w kontrolerze NVIC. Banał.

23) tu rozpoczyna się procedura obsługi przerwania dla linii EXTI2. Interesujące są dwie sprawy:

- atrybut: o tym już pisałem przy okazji zadania 6.1 i w dodatku 5
- to, że nazwa funkcji **musi być identyczna** jak w tablicy wektorów, ale o tym też już pisałem przy okazji zadania 6.1 (wszystko już było :])

24) rejestr EXTI_PR zawiera flagi przerwań układu peryferyjnego. Sprawdzamy w nim co wywołało przerwanie. W tak prostym przypadku jak ten przykład nie ma to może specjalnego

103 **RTSR** - *Rising Trigger Selection Register*; **FTSR** - *Falling Trigger Selection Register*

sensu, ale pamiętajmy że niektóre wektory przerwań są wspólne dla wielu linii. Wtedy musimy programowo sprawdzić, która linia wywołała przerwanie.

25) testując rejestr EXTI_PR upewniliśmy się, że przerwanie wywołała linia nr 2. I teraz bardzo ważna sprawa! Kasujemy flagę przerwania w rejestrze układu peryferyjnego. Proszę nie mylić ustawionej flagi w rejestrze EXTI_PR ze stanem pending w NVICu, bo to zupełnie różne rzeczy. **To jest ważne!** Rejestr EXTI_PR to rejestr peryferiala (każdy układ peryferyjny generujący przerwania ma jakiś swój rejestr z flagami przerwań). Rejestr układu peryferyjnego wykorzystujemy do zidentyfikowania konkretnej przyczyny przerwania, po czym **trzeba go skasować ręcznie**. W omawianym przykładzie sprawdzamy, która linia EXTI wywołała przerwanie i kasujemy flagę. Z kolei stan pending w NVICu (i związana z nim flaga gdzieś w rejestrach rdzenia) oznacza, że przerwanie zostało zgłoszone przez układ peryferyjny, ale czeka w kontrolerze NVIC bo:

- aktualnie obsługiwany wyjątek ma wyższy priorytet
- dlatego że to przerwanie jest wyłączone w NVICu

W procedurze obsługi przerwania nie musimy kasować pendingu w NVICu. NVIC sam sobie zmieni stan wyjątku z pending na active.

Z tym kasowaniem flagi jest jeszcze jeden wałek. Kasowanie bitu (linia 25 kodu) następuje po wpisaniu do niego jedynki! Zera nie mają znaczenia. W RMie jest to oznaczone przy opisie rejestrów skrótem *rc_w1*: bit można czytać (*read*) i kasować (*clear*) poprzez wpisanie (*write*) jedynki **(1)**. Trzeba na to uważać, bo co peryferial to inaczej się kasuje flagę przerwania. Reszta kodu nie zawiera już nic nowego. Uwaga! Koniecznie przeczytaj opis rozwiązania analogicznego zadania, z rozdziału poświęconego mikrokontrolerowi F334 (zadanie 7.4; 26 i 27 linia kodu).

Zadanie domowe 7.2: sprawdzić co się stanie jeśli w ISR nie będzie kasowania flagi w rejestrze EXTI_PR. Tylko żeby nie było – chodzi mi o głębszą analizę niż „dioda się nie zapala” - proszę przemyśleć co się dzieje. Można oczywiście korzystać z debuggerów.

Odpowiedź: (czcionka w kolorze tła - aby odczytać skopij do innego edytora, ale najpierw sprawdź sam! Wszystko wydaje się proste jak się zna odpowiedź.) [
].

Na koniec bardzo ważna uwaga! Flagi przerwania **nie należy** czyścić na końcu obsługi przerwania. Najlepiej robić to na początku po sprawdzeniu źródła przerwania. Wy tłumaczenie

trochę mnie przerasta merytorycznie... generalnie chodzi o to, że jeśli flaga jest czyszczona na końcu, to istnieje ryzyko że procesor zdąży wyjść z przerwania zanim przetrawi się rozkaz kasowania flagi (dokładniej to chyba peryferium potrzebuje chwili na zdjęcie sygnału zgłoszenia przerwania). Tak czy siak zaowocuje to tym, że ponownie odpali się przerwanie – tak jakby flaga w ogóle nie była kasowana (patrz zadanie 7.2). Dlatego trzeba kasować odpowiednio wcześniej. Dodatkowo wczesne kasowanie flagi pozwala uniknąć gubienia przerwań, które pojawią się w czasie wykonywania ISR. Jeśli w czasie wykonywania ISR pojawi się nowe przerwanie z tego samego źródła to ustawi (skasowaną na początku ISR) flagę w rejestrze układu peryferyjnego. Gdybyśmy kasowali flagę na końcu to nie zauważymy tego drugiego przerwania.

W kontekście EXTI pojawiają się jeszcze dwa rejesty: SWIER (Software Interrupt Register) i EMR (Event Mask Register). SWIER pozwala programowo wymusić przerwanie zewnętrzne – prosta sprawa.

Sprawa prosta... ale: za pomocą rejestrów SWIER symulujemy przerwanie w układzie peryferyjnym. Ale mamy też drugą opcję żeby wymusić przerwanie - funkcja *NVIC_SetPendingIRQ()*, która wymusza przejście przerwania w NVICu w stan oczekujący (czyli jeśli jest włączone i priorytet pozwoli to zostanie obsłużone). No i oczywiście rodzi się pytanie: kiedy korzystać z której opcji? Powiem szczerze - nie mam bladego pojęcia. Tzn. na pewno jeśli wymusimy przerwanie w NVICu to nie będzie ustawiona flaga przerwania w rejestrze układu peryferyjnego. ISR powinno sprawdzać flagę aby określić źródło przerwania a tu niespodzianka bo nie będzie żadnej flagi... Pod tym względem lepiej skorzystać z SWIER, bo pozwala zasymulować konkretne przerwanie z ustawieniem flagi. Notabene właśnie to robi ten rejestr - ustawia flagę w rejestrze PR. Tyle wymyśliłem :) Koniec OT!

EMR natomiast działa analogicznie do IMR tylko zamiast przerwania od danej linii EXTI, włącza generowanie zdarzeń. W tym miejscu pojawia się pytanie: *WTF is Event?* Zdarzenie to takie... szturchnięcie rdzenia paluchem. Nie powoduje zmian w przepływie programu (jak przerwanie które powoduje skok do funkcji ISR), ale może wybudzić procek z uśpienia. Na razie tyle w temacie. Więcej przy oszczędzaniu energii w rozdziale 11.

Co warto zapamiętać z tego rozdziału?

- prawie każdy pin mikrokontrolera może generować przerwanie zewnętrzne

- naraz można włączyć przerwanie tylko od jednego pinu o danym numerze
- nie każda linia przerwania ma osobny wektor (funkcję obsługi przerwania)
- przy przerwaniach zewnętrznych musimy:
 - skonfigurować pin - jako wejście z ewentualnym podciąganiem
 - ustawić, który port ma generować przerwanie na danej linii
 - od-maskować linię i ustawić na jakie zbocza ma reagować
 - włączyć przerwanie w kontrolerze NVIC
- w ISR musimy zidentyfikować źródło przerwania i (najlepiej od razu) skasować flagę przerwania w peryferialu

7.2. EXTI (F429)

Niestety albo i stety, w STM32F429 jest troszkę inaczej. Tzn. dalej mamy rdzeń Cortex-M więc w kontrolerze NVIC, który jest elementem rdzenia, nic się nie zmienia. Ale w peryferiach mikrokontrolera a i owszem.

Wstyd się przyznać, ale z tym mikrokontrolerem nie miałem dotąd za wiele wspólnego. To jest chyba mój pierwszy raz jeśli chodzi o przerwania zewnętrzne w tej kostce... także będziemy się uczyć razem. Odpaliłem RM0090, rozdział *General-purpose I/Os* i podrozdział *External interrupt/wakeup lines*. To chyba dosyć oczywiste miejsce do rozpoczęcia poszukiwań. Coś mi chodzi po głowie¹⁰⁴, że przyda się też rozdział *System configuration controller*. Swoją drogą w RM0008 jest, na samym początku, taka tabelka¹⁰⁵, która pokazuje jakie rozdziały należy przeczytać jeśli chce się coś uruchomić. Niby nigdy nie korzystałem, ale wydaje się przydatne dla początkujących. Ciekawe czemu porzucili ten pomysł. Dosyć gadania, zanurzam się w lekturze. Czytelnikowi też to radzę :)

O! I już na początku czytania okazało się, że jeszcze jeden rozdział się przyda. W rozdziale o GPIO jest odwołanie do rozdziału *Interrupts and events*. Wcześniej go nie zauważylem :) A tak poza tym to już wszystko wiem i... jest po staremu z wyjątkiem tego, że:

- rejesty konfigurujące połączenia między liniami przerwań a portami (EXTICR_x) nie są teraz w bloku AFIO tylko w SYSCFG
- jest więcej linii przerwań przyporządkowanych na sztywno do jakichś peryferiów (bo i peryferiów jest więcej), po szczegóły odsyłam do dokumentacji

¹⁰⁴ to są plusy wielu dupogodzin spędzonych na przeglądaniu for internetowych :)

¹⁰⁵ Sections related to each STM32F10xxx product

Zadanie domowe 7.3: to samo co w zadaniu 7.1 tylko procek inny. Dwa przerwania, jedno zapala, drugie gasi diodę.

Przykładowe rozwiązanie (F429, dioda PG13, przerwania od PA0, PG2):

```
1. int main(void) {
2.
3.     RCC->AHB1ENR |= RCC_AHB1ENR_GPIOAEN | RCC_AHB1ENR_GPIOGEN;
4.     RCC->APB2ENR = RCC_APB2ENR_SYSCFGEN;
5.     __DSB();
6.
7.     gpio_pin_cfg(GPIOG, PG13, gpio_mode_output_PP_LS);
8.     gpio_pin_cfg(GPIOA, PA0, gpio_mode_input_floating);
9.     gpio_pin_cfg(GPIOG, PG2, gpio_mode_input_PU);
10.
11.    SYSCFG->EXTICR[0] = SYSCFG_EXTICR1 EXTI0_PA | SYSCFG_EXTICR1 EXTI2_PG;
12.
13.    EXTI->FTSR = EXTI_FTSR_TR2;
14.    EXTI->RTSR = EXTI_RTSR_TR0;
15.    EXTI->IMR = EXTI_IMR_MR0 | EXTI_IMR_MR2;
16.
17.    NVIC_EnableIRQ(EXTI0_IRQn);
18.    NVIC_EnableIRQ(EXTI2_IRQn);
19.
20.    while (1);
21.
22. } /* main */
23.
24. void EXTI0_IRQHandler(void) {
25.     if (EXTI->PR & EXTI_PR_PR0) {
26.         EXTI->PR = EXTI_PR_PR0;
27.         BB(GPIOG->ODR, PG13) = 1;
28.     }
29. }
30.
31. void EXTI2_IRQHandler(void) {
32.     if (EXTI->PR & EXTI_PR_PR2) {
33.         EXTI->PR = EXTI_PR_PR2;
34.         BB(GPIOG->ODR, PG13) = 0;
35.     }
36. }
```

Specjalnie się to nie różni od poprzedniej wersji:

3, 4) włączenie zegarów dla używanych portów i bloku SYSCFG. Przypominam o niezerowej wartości rejestru i sumie bitowej :)

5) nasz rodzynek z erraty: *dsb (Data Synchronization Barrier)*. Dla zainteresowanych: inne instrukcje „barierowe” to: *isb (Instruction Synchronization Barrier)* oraz *dmb (Data Memory Barier)*

11) tu jest właściwie jedyna różnica w stosunku do wersji na STM32F1. Rejestry EXTICR są w innym bloku. I tyleż w temacie :)

Co warto zapamiętać z tego rozdziału?

- praktycznie jedną różnicą między EXTI w F103 i F429 jest inne położenie rejestrów

7.3. EXTI (F334)

Moduł EXTI w mikrokontrolerze F334 jest podobny do wersji z F429. Istnieje jednak kilka kosmetycznych różnic, o których warto wspomnieć. Przede wszystkim jest więcej linii EXTI. ST zafundowało nam 36 linii, przy czym nie wszystkie są wykorzystane w omawianym mikrokontrolerze. Linie podzielono na dwie kategorie:

- linie określone jako *zewnętrzne* (28 linii)
- linie określone mianem *wewnętrznych* (8 linii)

Wśród linii zewnętrznych znajdziemy, znane i lubiane, linie związane z przerwaniami zewnętrznymi od pinów mikrokontrolera (EXTI0...EXTI15) oraz linie połączone z konkretnymi układami peryferyjnymi mikrokontrolera (wykropczyłem tylko te wykorzystane w F334):

- EXTI16 - programowalny układ kontroli napięcia (PVD, więcej w rozdziale 11.4)
- EXTI17 - alarm zegara czasu rzeczywistego (RTC Alarm, więcej w rozdziale 9.7)
- EXTI19 - funkcje *tamper* i *timestamp* zegara czasu rzeczywistego (RTC Tamper i RTC Timestamp, więcej w rozdziale 9.7)
- EXTI20 - funkcja wybudzania mikrokontrolera przez zegar czasu rzeczywistego (RTC Wakeup, więcej w rozdziale 9.7)
- EXTI22 - drugi komparator analogowy (COMP2)
- EXTI30 - czwarty komparator analogowy (COMP4)
- EXTI32 - szósty komparator analogowy (COMP6)

I wreszcie novum, czyli linie EXTI określane jako „wewnętrzne”, a wśród nich:

- EXTI23 - funkcja wybudzania mikrokontrolera przez układ I2C1 (I2C1 Wakeup)
- EXTI25 - funkcja wybudzania mikrokontrolera przez układ USART1 (USART1 Wakeup, więcej w rozdziale 15.3)
- EXTI26 - funkcja wybudzania mikrokontrolera przez układ USART2 (USART2 Wakeup)
- EXTI28 - funkcja wybudzania mikrokontrolera przez układ USART3 (USART3 Wakeup)

Uwaga! W RMie (wersja *Rev2, docID 025177, October 2015*) coś jest grubo pomylone przy opisie wewnętrznych linii EXTI. Nie zgadza się lista wewnętrznych linii w rozdziale *External and internal interrupt/event line mapping*. Jest tam notka głosząca co następuje:

„*EXTI lines 18, 21, 24, 27, 29, 31, 33, 34 and 35 are internal.*”

Już na pierwszy rzut oka widać, że jest źle. W wielu miejscach w dokumencie znajduje się informacja, że linii wewnętrznych jest osiem... a nie dziewięć. Ponadto jeśli popatrzasz na rozpiskę linii umieszczoną w tym samym rozdziale, to zauważysz, że wszystkie wymienione linie są niewykorzystane - *Reserved*. No i na koniec, ta lista nie zgadza się z informacjami umieszczonymi przy opisie rejestrów bloku EXTI (rozdział *EXTI Registers*, opis rejestrów *IMR1, IMR2*). Prawidłowa lista linii wewnętrznych (stworzona na podstawie opisu rejestrów) to: 23, 24, 25, 26, 27, 28, 34, 35.

Za pewne zastanawiasz się, o co w ogóle chodzi z tymi liniami wewnętrznymi? Linie te służą tylko do wybudzania mikrokontrolera z uśpienia. Różnią się od pozostałych linii EXTI następującymi cechami:

- przerwania od tych linii są domyślnie, po resecie, włączone (odpowiadające im bity w rejestrach EXTI_IMRx są ustawione)
- przerwania te, nie posiadają flag oczekiwania (*pending*) w bloku EXTI, flagi dostępne są w rejestrach układów peryferyjnych związanych z tymi liniami
- przerwania na tych liniach mogą być wygenerowane **tylko gdy procesor jest uśpiony** (*stop mode*)

Więcej informacji na temat „wybudzeniowych” możliwości układu EXTI, pojawi się (najprawdopodobniej) w rozdziale 11.

Ze względu na dużą liczbę linii EXTI, ich konfiguracja nie mieści się w jednym rejestrze. Z tego względu, w F334, występują:

- dwa rejesty włączające przerwania od danej linii (EXTI_IMR1 oraz EXTI_IMR2)
- dwa rejesty włączające zdarzenia od danej linii (EXTI_EMR1 oraz EXTI_EMR2)

- po dwa rejesty odpowiedzialne za konfigurację zbocza (EXTI_RTSR1, EXTI_RTSR2, EXTI_FTSR1, EXTI_FTSR2)
- dwa rejesty zawierające flagi przerwań (EXTI_PR1 oraz EXTI_PR2)
- dwa rejesty umożliwiające programowe wywołanie przerwania (EXTI_SWIER1 oraz EXTI_SWIER2)

Żeby nie było za prosto, w pliku nagłówkowym mikrokontrolera¹⁰⁶ zrezygnowano z sufiksów „,1”. Tylko „drugie” rejesty mają na końcu nazwy dodany numerek... bywa. Starczy, czas na naszą ulubioną część :]

Zadanie domowe 7.4: nuCleo ma tylko jeden przycisk (a zakładam, że nie chce nam się ruszać z krzesła w poszukiwaniu dodatkowego hardware'u), więc tym razem odpuścimy przykład z dwoma przyciskami. Zrobimy tak: przerwanie zewnętrzne (guzioł), na wybranym zboczu, zmienia stan diody na przeciwny. I tyle :) W gratisie dostaniemy ładny przykład udowadniający istnienie drgań styków. Do dzieła!

Przykładowe rozwiążanie (F334; diody: PA5 i PF1; przycisk PC13):

```

1. int main(void){
2.
3.     RCC->AHBENR = RCC_AHBENR_GPIOAEN | RCC_AHBENR_GPIOCEN | RCC_AHBENR_GPIOFEN;
4.     RCC->APB2ENR = RCC_APB2ENR_SYSCFGEN;
5.
6.     gpio_pin_cfg(GPIOA, PA5, gpio_mode_out_PP_LS);
7.     gpio_pin_cfg(GPIOF, PF1, gpio_mode_out_PP_LS);
8.     gpio_pin_cfg(GPIOC, PC13, gpio_mode_in_floating);
9.
10.    SYSCFG->EXTICR[3] = SYSCFG_EXTICR4_EXTI13_PC;
11.    EXTI->FTSR = EXTI_FTSR_TR13;
12.    EXTI->IMR = EXTI_IMR_MR13;
13.
14.    NVIC_EnableIRQ(EXTI15_10_IRQn);
15.    SysTick_Config(8000000/2);
16.    while(1);
17.
18. }
19.
20. void SysTick_Handler(void){
21.     GPIOF->ODR ^= PF1;
22. }
23.
24. void EXTI15_10_IRQHandler(void){
25.     if(EXTI->PR & EXTI_PR_PR13){
26.         EXTI->PR = EXTI_PR_PR13;
27.         //EXTI->PR = (EXTI->PR & 0x1F800000) | EXTI_PR_PR13;
28.         GPIOA->ODR ^= PA5;
29.     }
30. }
```

Cóż tu mamy ciekawego?

3) włączenie zegara dla wykorzystywanych portów

106 *stm32f334x8.h*, wersja V2.2.0 z 13.11.2015

4) włączenie zegara bloku SYSCFG w którym znajdują się rejestrzy EXTCR (tak jak w F429)

6 - 8) konfiguracja wyprowadzeń: PC13 - przycisk, PA5 - dioda, PF1 - dioda (a co! przylutowałem to się bawię - patrz dodatek 7)

10 - 12) konfiguracja przerwania od zbocza „z górką”, dokładnie tak samo jak w F429

14) włączenie przerwania w kontrolerze NVIC

15) konfiguracja SysTicka (migająca dioda w przerwaniu 20 - 22)

24) ISR przerwania zewnętrznego, wektor jest wspólny dla linii od EXTI10 do EXTI15

25) sprawdzenie źródła przerwania

26 - 27) tu się na chwilę zatrzymamy. Generalnie chcemy skasować flagę przerwania. Jeżeli popatrzymy do opisu rejestru EXTI_PR to zobaczymy, że wszystkie bity tego rejestru są kasowane poprzez wpisanie jedynki (oznaczenie *rc_w1*). Jest to (a właściwie „mogłoby być”) bardzo wygodne, bo chcąc skasować np. bit EXTI_PR_PR13, wystarczy wykonać prosty zapis jak z linii 26 listingu. Ładne, proste, finezyjne. I generalnie „wszyscy” tak robią - tak jest w większości przykładów w nacie, tak jest zrobione w kodzie funkcji z SPL, tak jest w kodzie biblioteki HAL, tak jest w przykładach z poprzednich dwóch rozdziałów, tak jest w przykładowych kodach od ST (*STM32 code snippets*)... a nie, tam to akurat zrobili zwykłe przypisanie z sumą bitową czym skasowali wszystkie flagi naraz - czyli fuszerka po całości (pisałem o tym w rozdziale poświęconym bit bandingowi - 4.6).

Wracając do meritum. „Wszyscy” tak robią, producent tak robi, działa... To czego się *szczywronek* czepia? Ano czepia, bo to rozwiązanie jest niezgodne z zaleceniami dokumentacji! Rozważ przypisanie z linii 26 listingu - wpisujemy jedynkę na pozycję PR13 (bo flaga kasowana jedynką), całą resztę rejestru zapisujemy równocześnie zerami. No i wszystko byłoby ok, gdyby nie to, że ten register zawiera pola *zastrzeżone (Reserved)*. Dokumentacja jasno preczyzuje, że do takich pól można zapisywać, co najwyżej, ich domyślną zawartość („*Reserved, must be kept at reset value.*”). Czyli jaką? Ano właśnie s... nijką. Nie znamy tej wartości. RM, dla tych rejestrów, podaje domyślną wartość - „*undefined*”. Więc nie możemy sobie ot tak wpisywać zer gdzie popadnie...

Poprawiona wersja kasowania flagi jest pokazana w linii 27. Odczytuję zawartość rejestru, kasuję wszystkie bity poza tymi z pól *zastrzeżonych* (zostają niezmienione), ustawiam PR13 i zapisuję z powrotem. Wygląda to paskudnie i nie wykorzystuje „potencjału” pól kasowanych wpisaniem jedynki, ale sumienie pozostaje czyste i śpi się lepiej :) A krótszy zapis (z linii 26) dalej korci... szczególnie, że nawet ST tak robi :] Zdecyduj sam!

Co warto zapamiętać z tego rozdziału?

- przerwania zewnętrzne F334 działają praktycznie tak samo jak w F429

8. LICZNIKI („*FESTINA LENTE!*”¹⁰⁷)

8.1. Wstęp

To będzie długi rozdział :) I trudny dla mnie, bo liczniki w STM32 są dosyć skomplikowane (w porównaniu z AVR8) ze względu na mnogość trybów pracy. Nie wiem czy uda mi się to jakoś zgrabnie zebrać do kupy. Może by ominąć temat liczników... może nikt nie zauważy :}

Do dyspozycji mamy trzy typy liczników:

- Advanced control timer (TIM1, 8)
- General purpose timers (TIM2..5, 9..14)
- Basic timers (TIM6, 7)

Postaram się omówić najbardziej rozbudowane z nich - *advanced*. Pozostałe grupy są po prostu uboższe w jakieś opcje, więc jeśli się ogarnie te pierwsze to reszta nie będzie problemem.

Podstawowe cechy *advanced timersów*:

- liczniki 16bitowe
- liczenie w górę lub w dół
- preskaler 16bitowy (podział z zakresu /1 ... /65535)
- możliwość generacji wszelkiej maści PWMów i impulsów
- 4 kanały wejść zatrzaskujących (*Input Capture*) lub wyjść porównawczych (*Output Compare*)
- wyjścia komplementarne z programowanym czasem martwym i funkcją *Break*¹⁰⁸
- synchronizacja kilku liczników
- generowanie całej masy przerwań, zdarzeń, żądań DMA
- sprzętowe wsparcie dla enkoderów i czujników hallotronowych
- różnorakie źródła sygnału zegarowego (w tym zewnętrzne)

Co warto zapamiętać z tego rozdziału?

- są trzy rodzaje liczników (zaawansowane, ogólnego przeznaczenia oraz podstawowe)
- liczniki grupy *zaawansowane* mają najwięcej funkcji, pozostałe są części z nich pozbawione

107 „*Spiesz się powoli!*”

108 sprzętowe mechanizmy wspierające sterowanie energoelektroniką - np. wszelkiej maści układami mostkowymi

8.2. Blok zliczający, prosty timer

Działanie każdego licznika opiera się na zliczaniu *czegoś*. Podstawowym blokiem układu jest blok zliczający¹⁰⁹ oparty o rejestr TIM_CNT. Zliczanie (np. taktów zegara) powoduje inkrementację lub dekrementację licznika (rejestru CNT). Licznik zlicza w zakresie od 0 do wartości rejestru przeładowania¹⁰⁹ (rejestr TIM_ARR). W zależności od konfiguracji zliczanie może następować:

- w górę: od wartości zero do wartości rejestru przeładowania ARR, potem się przekręca i liczy znowu od zera
- w dół: od ARR do 0, potem się przekręca i znowu liczy od ARR
- symetrycznie: w górę od zera do ARR-1 potem w dół do zera i znowu w górę...

Licznik może zliczać:

- sygnały zegarowe (z uwzględnieniem preskalera)¹¹⁰
- sygnały z zewnątrz, np. zbocza z jakiegoś pinu mikrokontrolera
- impulsy z innego licznika, np. licznik 2 może zliczać przepelenienia licznika 1

Zacznijmy od zabawy ze źródłami sygnału zegarowego. Skonfigurujmy licznik tak, aby generował przerwanie zegarowe co 1s. Proszę więc:

- przekartkować sobie rozdział opisujący licznik
- przeczytać dokładnie opis rejestrów licznika i wynotować to, co wydaje się przydatne
- samodzielnie poeksperymentować na żywym organizmie

Zadanie domowe 8.1: migająca dioda oparta o przerwanie licznika z grupy *advanced*.

Udało się? Jeśli się nie udało przez minimum 3 dni - i mam na myśli trzy dni solidnej pracy nad kodem i dokumentacją a nie godzinkę wieczorem :) To mała podpowiedź: konfiguracja licznika do tego zadania to ustawienie **aż** czterech rejestrów. Wspominałem, że licznik zlicza w przedziale od zera do TIM_ARR i ma preskaler (TIM_PSC)... poza tym trzeba włączyć licznik i generowanie przezeń przerwań. Więcej podpowiedzi nie będzie. Sio i do zobaczenia jak dioda zacznie migać!

109 to moja prywatna nazwa, więc proszę się nie przywiązywać i nie traktować jej zbyt poważnie :)

110 wtedy jest bardziej *Timerem* niż licznikiem (*Counterem*)

Przykładowe rozwiązanie (F103, dioda na PB0):

```
1. int main(void) {
2.
3.     RCC->APB2ENR = RCC_APB2ENR_IOPBEN | RCC_APB2ENR_TIM1EN;
4.     gpio_pin_cfg(GPIOB, PB0, gpio_mode_output_PP_2MHz);
5.
6.     TIM1->PSC = 999;
7.     TIM1->ARR = 3999;
8.     TIM1->DIER = TIM_DIER_UIE;
9.     TIM1->CR1 = TIM_CR1_CEN;
10.
11.    NVIC_EnableIRQ(TIM1_UP IRQn);
12.    while (1);
13.
14. } /* main */
15.
16.
17. __attribute__((interrupt)) void TIM1_UP_IRQHandler(void){
18.     if (TIM1->SR & TIM_SR UIF){
19.         TIM1->SR = ~TIM_SR UIF;
20.         BB(GPIOB->ODR, PB0) ^=1;
21.     }
22. }
```

3) włączenie zegara dla licznika TIM1 i portu By, potem konfiguracja pinu

6) od tej chwili zaczyna się konfiguracja licznika. Rejestr PSC to rejestr preskalera, ARR to rejestr przeładowania. Wzór na częstotliwość przekręcania się licznika gdzieś pewnie jest w dokumentacji¹¹¹, ale generalnie wszystkie liczniki w świecie mikrokontrolerów rządzą się tymi samymi prawami, więc będzie to coś w stylu:

$$f_{UEV} = \frac{F_{TIM}}{(ARR + 1) \cdot (PSC + 1)}$$

gdzie:

- f_{UEV} - częstotliwość występowania *Update Event* (zaraz się wyjaśni), czyli częstotliwość przekręcania (przepelniania) się licznika
- F_{TIM} - częstotliwość sygnału zegarowego taktującego blok licznika (nasze domyślne 8MHz w F103)
- ARR, PSC - wartości rejestrów przeładowania i preskalera

8) włączamy generowanie przerwań przy odświeżeniu licznika (*Update Event Interrupt*)

9) włączenie licznika - od tej chwili zaczyna zliczać

11) włączenie przerwania od licznika w kontrolerze NVIC

19) tu jest mikro pułapka: pamiętasz kasowanie flagi przerwania w EXTI_PR? Tam był rejestr kasowany wpisaniem jedynki (rc_w1), tutaj jest rejestr rc_w0 - myślę, że każdy sobie sam rozszyfruje :) Uprzedziłem, że trzeba na to uważać.

111 sporo wzorów jest w nocy aplikacyjnej: AN4013

Co warto zapamiętać z tego rozdziału?

- patrz rozdział 8.9

8.3. Update Event (UEV), buforowanie rejestrów i One Pulse Mode

UEV to zdarzenie¹¹², które może być generowane:

- przy przepelnieniu licznika (*overflow* lub *underflow*)
- programowo poprzez ustawienie bitu UG w rejestrze TIM_EGR
- na żądanie układu nadzorującego jeśli kilka liczników jest połączonych (dołączenia liczników dojdziemy w swoim czasie)

Po co ten UEV? UEV powoduje „wyzerowanie” wartości licznika (rejestru TIM_CNT). Licznik dolicza do maksymalnej wartości (do rejestru ARR), odpala się UEV i powoduje wyzerowanie rejestru CNT. Dzięki temu licznik się przekręca i zlicza od zera. Przy czym tu jest haczyk - rejestr CNT jest zerowany tylko jeśli licznik zlicza w góre. Jeśli licznik zlicza w dół to wyzerowanie go nie miałoby sensu bo już przecież doliczył do 0... przy zliczaniu w dół, CNT przyjmuje wartość rejestru przeładowania. Dzięki temu licznik znowu może zliczać od ARR w dół do zera

UEV może też generować przerwanie lub żądanie DMA. DMA jeszcze nie znamy, więc przykładów na razie nie będzie. Generowanie przerwań wykorzystałem w pierwszym przykładowym programie (zadanie 8.1): licznik po przepelnieniu generuje UEV a UEV odpala przerwanie.

UEV powoduje ponadto wpisanie nowej wartości do **rejestrów buforowanych**. Część rejestrów licznika (ARR, PSC, RCR, CCRx) jest buforowana. Zapisując coś do takiego rejestru, zapisujemy w rzeczywistości do rejestru tymczasowego. Nowa wartość z rejestru tymczasowego zostaje przepisana do prawdziwego rejestru właśnie w momencie odświeżania (UEV). Po co to?

Przykład: mamy licznik liczący od 0 do 200, który generuje przerwania przy przepelnieniu. Chcemy zmienić okres przerwań tak aby był o połowę krótszy. W tym celu zmieniamy górną granicę zliczania z 200 na 100. Dotąd jasne? A widzisz już pułapkę? Wcześniej licznik zliczał do 200. W chwili zmieniania górnego zakresu mógł więc mieć wartość np. 150. Wpisujemy nową górną wartość (100), która jest mniejsza od aktualnego stanu licznika ($100 < 150$). Efekt jest taki, że licznik zlicza dalej w góre... aż do momentu, gdy przekręci się rejestr CNT (16bit - 65535). Rejestr się przekręca i dalej już jest ok, bo liczy od zera do 100. Ale! Przez chwilę mieliśmy kosmicznie

112 czyli takie coś co powoduje coś innego...

długą przerwę, albowiem licznik musiał zliczyć do ponad 65 tysięcy. Niby tylko jeden raz, ale jednak. Właśnie po to aby wyeliminować ten problem, wprowadzono buforowanie. Nowe wartości rejestrów są wpisywane do nich, dokładnie w chwili przekręcania się licznika.

Czyli: my wpisujemy nową wartość do rejestru tymczasowego kiedy tylko chcemy, a licznik sprzętowo czeka na „bezpieczny moment” aby przepisać ją do prawdziwego rejestru. Prawda, że fajne rozwiązanie? Mamy też pewne możliwości zmian konfiguracji w kwestii UEV i buforowania:

- bit ARPE w rejestrze TIM_CR1 - umożliwia wyłączenie buforowania rejestru ARR, czyli nowa wartość będzie działała od razu po wpisaniu
- bit URS w rejestrze TIM_CR1 - po jego ustawieniu UEV jest generowany tylko przy przepełnieniu licznika, nie działa generowanie programowe (bitem UG w rejestrze TIM_EGR) i generowanie UEV przez nadrzędny licznik (przy łączeniu liczników)
- bit UDIS w rejestrze TIM_CR1 - pozwala wyłączyć całkowicie generowanie UEV, np. na czas wpisywania nowych wartości do kilku rejestrów buforowanych
- bit UG w rejestrze TIM_EGR - pozwala programowo wymusić UEV

Jeszcze jedno nowe pojęcie na koniec: One Pulse Mode. Opis tego „trybu” w RM jest doskonałym przykładem jak można zaciemnić coś prostego. Opierając się na opisie z rozdziału *One-pulse mode* (RM) można stwierdzić, że OPM to tryb, w którym licznik po uruchomieniu generuje na wyjściu impuls o ustawionej długości po ustalonym opóźnieniu od uruchomienia... Niby racja ale czy nie prościej powiedzieć, że bit OPM powoduje, że licznik zatrzyma się przy najbliższym UEV? Bo właśnie tak to działa. OPM powoduje, że najbliższe przekręcenie się licznika wyzeruje bit włączający licznik - TIM_CR1_CEN. I tyle w temacie.

Co warto zapamiętać z tego rozdziału?

- patrz rozdział 8.9

8.4. Schemat blokowy licznika

To jest chyba najważniejszy moment, żeby zaprzyjaźnić się ze schematem blokowym licznika (RM → *Advanced-control timer block diagram*). Przy AVRach jakoś nie zwracałem na schematy większej uwagi, wydawały się dodatkiem do opisu słownego. W STMach mam wrażenie, że jest odwrotnie. Zresztą, kto by chciał opisać słownie np. drzewko zegarowe :)

Popatrzmy więc na schemat blokowy licznika w RM. Nie przejmuj się, że nie rozumiesz $\frac{3}{4}$ z tego opisu - ja też wszystkiego nie rozumiem. Szczegóły można sobie doczytać w tekście. Nas

interesuje tylko ogólne spojrzenie i zaprzyjaźnienie się ze schematem. Na schemacie blokowym można łatwo odnaleźć drogę jakiegoś sygnału, od razu widać przez jakie bloki przechodzi i dokąd może dojść. I właśnie to odnajdywanie drogi nam się niedługo przyda. Jedziemy. Choć na razie będzie to wszystko dosyć abstrakcyjne.

Na samej górze widać wejście wewnętrznego sygnału zegarowego (CK_INT). Dochodzi on do bloków odpowiedzialnych za wyzwalanie licznika (*Trigger Controller*), pracę licznika w trybie podrzędnym (*Slave Mode Controller*) i obsługę enkoderów (*Encoder Interface*).

Do tego bloku dochodzi też sygnał z pinu ETR, który wcześniej przechodzi przez wykrywacz zbocz, preskaler i filtr. Ten sygnał (ETRF) trafia również na multiplekser z którego wychodzi sygnał TRGI.

Sygnał TRGI jest wykorzystywany przez układ sterujący licznikiem podrzędnym (*Slave Mode Controller*) do resetowania, wyzwalania, bramkowania i taktowania licznika. Innym źródłem sygnału TRGI może być np. jeden z sygnałów ITR (sygnał pochodzący z innego licznika).

Z drugiej strony bloku wychodzi sygnał TRGO, to jest sygnał który może być odebrany przez inny licznik (dla tego innego licznika to będzie sygnał ITR) lub przetwornik ADC/DAC (wyzwalanie konwersji).

Z kontrolera licznika wychodzi ponadto sygnał taktujący (CK_PSC), który następnie przechodzi przez blok preskalera i dochodzi do bloku zliczającego *CNT Counter*.

Z lewej strony schematu są nóżki czterech kanałów wejściowych licznika. Sygnały od tych nóżek (TI1, TI2, TI3, TI4) są doprowadzone do bloków filtrujących i wykrywających zbocza. Następnie dochodzą do kilku multiplekserów, z których wychodzą cztery sygnały IC1..IC4. Proszę zwrócić uwagę na to, że np. nóżka pierwszego kanału może być źródłem sygnałów:

- IC1 (TIMx_CH1 → TI1 → TI1FP1 → IC1)
- IC2 (TIMx_CH1 → TI1 → TI1FP2 → IC2)

Już niedługo z tego skorzystamy. Sygnały TI1FP1 oraz TI2FP2 są ponadto doprowadzone do kontrolera interfejsu enkodera i multipleksera od sygnału TRGI. Dla zwiększenia czytelności schematu, te połączenia nie są zaznaczone ciągłą linią tylko samymi etykietami sygnałów.

Sygnały ICx idą dalej na preskalery i na cztery bloki *Capture/Compare*. Sygnały IC będą wyzwalaly funkcje *zatrzaszkującą* (*Capture*), czyli będą zatrzaskiwały zawartość licznika CNT w rejestrze bloku *Capture/Compare*. Piorunki przy sygnałach oznaczają możliwość generowania przerwań przez sygnały ICxPS.

Z drugiej strony bloków *przechwytywająco-porównujących* (*Capture/Compare*) wychodzą sygnały OCxREF. Ich stan związany jest z funkcją porównującą (*Compare*), czyli porównywaniem

wartości CNT z wartością referencyjną bloku *Capture/Compare*. Sygnały przechodzą dalej przez generatory czasu martwego (DTG) i dochodzą do bloczków kontrolujących nóżki wyjściowe związane z licznikiem. Nóżka np. TIMx_CH1 po lewej i TIMx_CH1 po prawej stronie schematu, to fizycznie to samo wyprowadzenie mikrokontrolera.

Do pełni szczęścia mamy jeszcze kilka schematów szczegółowych. Np. sygnał wejściowy z nóżki TIMx_CH1 (TI1) wchodzi na jakieś filtry i detektory zboczy... ale jak to skonfigurować w praktyce? Zerknijmy na schemat *Capture/compare channel (example: channel 1 input stage)*. Zaczyna się on od sygnału TI1 (sprawdź na schemacie ogólnym co to za sygnał) potem pokazany jest blok filtrujący. Mamy tam informację, że za konfigurację tego bloku odpowiadają bity ICF w rejestrze (licznika) CCMR1¹¹³. Dalej jest detektor zboczy. Za to, na jakie zbocze zareaguje linia TI1FP1, odpowiada multiplekser. Ze schematu można odczytać, że za konfigurację multipleksera odpowiadają bity CC1P i CC1NP rejestru CCER.

I tak dalej :) Zaraz spróbujemy wykorzystać to w praktyce.

Co warto zapamiętać z tego rozdziału?

- patrz rozdział 8.9

8.5. Licznik pędzony z zewnątrz

Wracamy do *bloku zliczającego*. Wprowadźmy małą modyfikację do poprzedniego programu (zadanie 8.1). Zamiast zliczania impulsów zegarowych spróbujmy zliczać impulsy z zewnątrz - z jakiegoś pinu. W RMie (opis licznika, rozdział *Clock Selection*) mamy informację, że licznik może być pędzony z zewnątrz w jednym z dwóch trybów:

- *External Clock Source Mode 1* - impulsy brane są z jednej z nóżek mikrokontrolera związanych z tym licznikiem
- *External Clock Source Mode 2* - impulsy brane są z wejścia ETR

Rozpatrzmy pierwszą kropkę. Druga kropka, czyli wersja z wejściem ETR jest bardzo podobna i jak ktoś ogarnie jedno to i z drugim sobie poradzi

Zadanie domowe 8.2: na schemacie blokowym licznika proszę prześledzić jakie nóżki (poza ETR) mogą być wykorzystywane do taktowania licznika.

¹¹³ zwróć uwagę na to, że rejesty CCMR mają dwa osobne opisy w RM, jeden dla trybu *Output Compare*, drugi dla *Input Capture*

Zadanie domowe 8.3: proszę ustalić numery wyprowadzeń układu scalonego odpowiadające nóżkom wyznaczonym w poprzednim zadaniu. Zakładamy, że interesuje nas licznik TIM1 i mikrokontroler STM32F103VCT6 (obudowa LQFP100).

Nóżki, które mogą być wykorzystane do taktowania licznika to TIMx_CH1 i TIMx_CH2¹¹⁴. Tylko one mają połączenie z blokiem kontroli licznika (poprzez sygnały TI1F_ED, TI1FP1, TI2FP2 - patrz schemat blokowy). Numery wyprowadzeń można sprawdzić w datasheetcie:

- *TIM1_CH1* to alternatywna funkcja nóżki PA8, wyprowadzenie nr.: 67 lub 40 (PE9) jeśli wykorzystamy [remapping](#)
- *TIM1_CH2* to alternatywna funkcja nóżki PA9, wyprowadzenie nr.: 68 lub 42 (PE11) jeśli wykorzystamy [remapping](#)

Do dzieła: licznik TIM1 ma zliczać wybrane zbocza na nóżce PA8 i po zliczeniu dziesięciu zapalać diodę (w przerwaniu od przekręcenia). Zerknijmy na chwilę na schemat blokowy licznika oraz schemat blokowy dla wybranego trybu pracy licznika (*External Clock Source Mode 1*)¹¹⁵ i prześledźmy drogę sygnału od PA8 do bloku zliczającego. Praktycznie wszystko co wypunktowałem poniżej, pochodzi **tylko** z analizy tych dwóch schematów blokowych. Jedno oko na schematy blokowe, drugie na tekst Poradnika i jedziemy:

- PA8 to wejście pierwszego kanału licznika TIM1_CH1, sygnał z tego wejścia nazywa się TI1
- TI1 przechodzi przez filtr (konfigurowany bitami ICF w rejestrze CCMR1) i od teraz nazywa się TI1F
- dalej jest detektor zboczy, z którego wychodzą dwa sygnały (jeden dla zboczy rosnących, drugi dla malejących)
- za wybór konkretnego zbocza odpowiada multiplekser, możemy nim sterować za pomocą bitów CC1P w rejestrze CCER
- za multiplekserem mamy sygnał TI1FP1 i kolejny multiplekser sterowany bitami TS rejestr SMCR
- doszliśmy do sygnału TRGI, zgodnie ze schematem pozostała nam konfiguracja bitów: ECE i SMS w rejestrze SMCR i mamy sygnał CK_PSC

114 na upartego chyba też TIMx_CH3 - przez bramkę XOR, ale to bardzo udziwione rozwiązanie :)

115 *TI2 external clock connection example.* Uwaga! W dokumentacji jest pokazany przykładowy schemat dla kanału drugiego, my wykorzystujemy kanał pierwszy - więc nazwy sygnałów i bitów będą się ciut różnić: TI1 zamiast TI2, CC1P zamiast CC2P...

- na schemacie ogólnym widać, że CK_PSC to sygnał wchodzący na preskaler bloku liczącego - czyli doszliśmy tam gdzie chcieliśmy, mamy sygnał taktujący licznik pochodzący z nóżki PA8 :)

Wypiszmy sobie wszystkie bity konfiguracyjne, jakie pojawiły się przy analizie schematów:

- ICF1 w rejestrze CCMR1
- CC1P w rejestrze CCER
- TS w rejestrze SMCR
- ECE w rejestrze SMCR
- SMS w rejestrze SMCR

Do tego doliczmy jeszcze znane nam już:

- rejestr ARR - chcemy zliczyć 10 zboczy i mieć przerwanie (UEV)
- rejestr DIER - w nim włączaliśmy przerwanie w poprzednim przykładzie
- rejestr CR1 - w nim włączaliśmy licznik w poprzednim przykładzie

Suma summarum mamy listę sześciu rejestrów, w których najprawdopodobniej będziemy grzebać. I to głównie dzięki schematom blokowym znaleźliśmy te rejstry. Teraz proponuję otworzyć sobie rozdział z opisem rejestrów i poczytać opisy wynotowanych bitów.

Zadanie domowe 8.4: licznik TIM1 ma zliczać wybrane zbocza na nóżce PA8 i po zliczeniu dziesięciu zapalać diodę w przerwaniu. Do dzieła Czytelniku, porównamy efekty za chwilę :)

Przykładowe rozwiązanie (F103, zliczanie impulsów z PA8, dioda na PB0):

```
1. int main(void) {
2.
3.     RCC->APB2ENR = RCC_APB2ENR_IOPBEN | RCC_APB2ENR_IOPAEN | RCC_APB2ENR_TIM1EN;
4.     gpio_pin_cfg(GPIOB, PB0, gpio_mode_output_PP_2MHz);
5.     gpio_pin_cfg(GPIOA, PA8, gpio_mode_input_pull);
6.     BB(GPIOA->ODR, PA8) = 1;
7.
8.     TIM1->CCER = TIM_CCER_CC1P;
9.     TIM1->SMCR = TIM_SMCR_SMS | TIM_SMCR_TS_0 | TIM_SMCR_TS_2;
10.    TIM1->ARR = 10;
11.    TIM1->DIER = TIM_DIER_UIE;
12.    TIM1->CR1 = TIM_CR1_CEN;
13.
14.    NVIC_EnableIRQ(TIM1_UP_IRQn);
15.
16.    SysTick_Config(800000);
17.
18.    while (1);
19.
20. } /* main */
21.
22.
23.
24.
25. __attribute__((interrupt)) void TIM1_UP_IRQHandler(void){
26.     if (TIM1->SR & TIM_SR UIF){
27.         TIM1->SR = ~TIM_SR UIF;
28.         BB(GPIOB->ODR, PB0) ^=1;
29.     }
30. }
31.
32.
33. __attribute__((interrupt)) void SysTick_Handler(void){
34.     BB(GPIOA->ODR, PA8) ^= 1;
35. }
```

3) włączamy zegar dla dwóch portów i licznika

4) wyjście diody (*push-pull*)

5) PA8 to wejście licznika, ustawiam jako zwykłe wejście z podciąganiem (bo to F103! w F429 należałoby wybrać konfigurację alternatywną)

6) włączam pull-up wejścia PA8

9) zaczynamy konfigurować licznik, CC1P odpowiada za detektor zbocza

10) bit SMS to wybór trybu pracy (*External Clock Source Mode 1*), TS to źródło sygnału TI1FP1

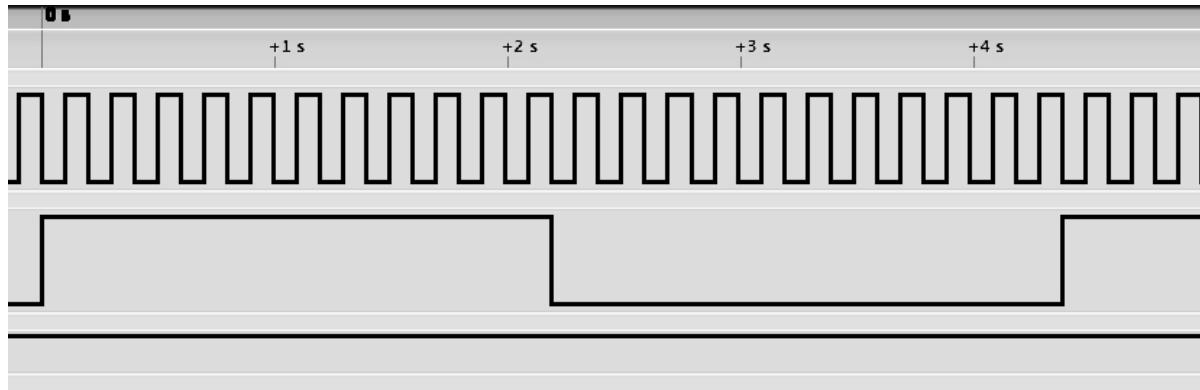
12, 13, 14) ustawiamy górną wartość zliczania¹¹⁶ i włączamy przerwanie od UEV jak w poprzednim przykładzie

18) geniusz lenistwa, nie chciało mi się iść po generator więc tak sobie wykombinowałem, że sygnał dla naszego licznika będzie generowany przez mikrokontroler. SysTick jest ustawiony tak aby w przerwaniu (co 0,1s) zmieniał podciąganie wejście PA8 (pull-up, pull-down, pull-up...). Efekt jest taki, że stan nóżki się zmienia – jest przebieg prostokątny, jest sygnał dla naszego licznika :)

28) w przerwaniu licznika macham diodą

116 tu dałem ciała - do ARR powinno się wpisać 9 bo zliczanie jest od zera :)

Efekt:



Rys. 8.1. Licznik zliczający impulsy zewnętrzne

Górny przebieg to PA8, czyli wejście licznika. SysTick macha tym wejściem co 0,1s za pomocą rezystorów podciągających. Licznik powinien zliczyć 11 (ARR = 10, a zliczanie jest od zera) zboczy opadających i na następnym (12-tym) się przekręcić. Działa? Działa.

Przed napisaniem kodu przygotowaliśmy, na podstawie schematów blokowych, listę bitów do modyfikacji. I co? I się praktycznie wszystko zgadza. „Pomyliliśmy” się tylko o jeden rejestr - nie skorzystaliśmy z opcji filtrowania sygnału, więc nie ruszaliśmy rejestru TIM_CCMR1. I to jest właśnie potęga schematów blokowych! Udało się bez problemu skonfigurować licznik i nie musieliszy przy tym czytać żadnych dłużnych opisów. Gdzie te tysiące stron dokumentacji, którymi straszy się początkujących?!

W tym momencie zachęcam do zabawy - proszę się nie bać i coś pozmieniać w konfiguracji. Zmiana zawartości rejestrów konfiguracyjnych debuggerem „*on the fly*” jest szybka, wygodna i pomaga przy okazji zapoznać się ze środowiskiem i korzystaniem z debuggera. I nie męczy pamięci Flash pierdylionem programowań. Do dzieła!

Zadanie domowe 8.5: skonfigurować licznik tak, aby zliczał oba zbocza sygnału z PA8 (podpowiedź: sygnał TI1_ED)

Co warto zapamiętać z tego rozdziału?

- patrz rozdział 8.9

8.6. Filtrowanie sygnałów zewnętrznych

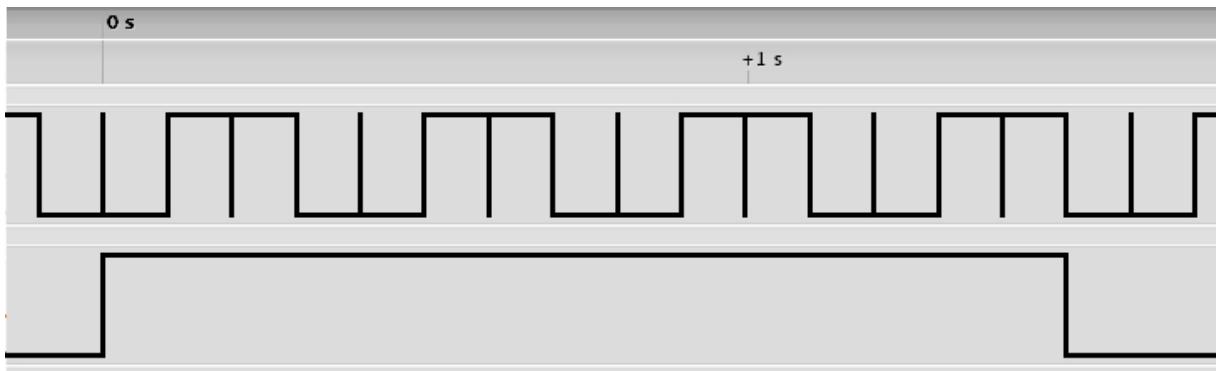
Jeszcze słówko o filtrowaniu. Uprzedzam, konkretów nie będzie bo średnio ogarniam ten temat. Generalnie chodzi o to, że poprzez bloki filtrujące mamy wpływ na częstotliwość

próbkowania i filtrowanie sygnału wejściowego. Polecam pobawić się bitami IC1F w TIM_CCMR1 i CKD w TIM_CR1. Poniżej przykład przerobionej funkcji od SysTicka (z zadania 8.5) tak aby generowała dodatkowe, krótkie szpilki „symulujące” zakłócenia:

Przerwanie SysTicka z symulatorem zakłóceń:

```
1. _attribute_((interrupt)) void SysTick_Handler(void){  
2.  
3.     static uint32_t delay=0;  
4.     delay++;  
5.  
6.     if (delay%2){  
7.         BB(GPIOA->ODR, PA8) ^= 1;  
8.     } else {  
9.         BB(GPIOA->ODR, PA8) ^= 1;  
10.        BB(GPIOA->ODR, PA8) ^= 1;  
11.    }  
12. }
```

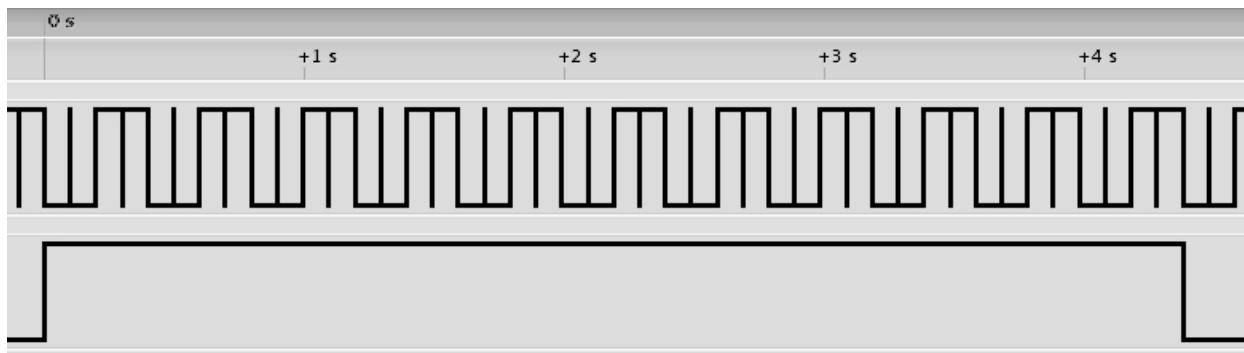
Tak to wygląda na analizatorze:



Rys. 8.2. Licznik taktowany zakłóconym sygnałem zewnętrznym (bez filtrowania)

Konfiguracja licznika nie została zmieniona, została taka jak w zadaniu 8.5. Licznik dalej zlicza 11 zboczy sygnału. Jak widać licznik zlicza również zbocza szpilek. A my chcielibyśmy je odsiąć. Jak ustawić filtrowanie? Na pewno da się to opisać matematycznie... ale to nie jest temat tego poradnika. Tak czy siak:

Zadanie domowe 8.6: dobrać filtrowanie sygnału wejściowego tak, aby licznik nie zliczał zboczy szpilek (można eksperymentować, może być metodą prób i błędów). Poniżej dowód na to, że się da:



Rys. 8.3. Licznik taktowany zakłóconym sygnałem zewnętrznym (z filtrowaniem)

Co warto zapamiętać z tego rozdziału?

- patrz rozdział 8.9

8.7. Tryb enkodera

Pewną wariacją trybu zliczania sygnałów zewnętrznych jest *Encoder Interface Mode*. Ten tryb, jak łatwo się domyśleć, służy do sprzętowej obsługi enkodera inkrementalnego. W trybie enkodera licznik zlicza zbocza sygnałów z enkodera i dodatkowo jest w stanie rozpoznać kierunek obrotów (zlicza w górę lub w dół). Niestety licznik sprzętowo nie obsługuje trzeciego/indeksującego kanału enkodera, ale bez problemu można to obejść np. za pomocą przerwania zewnętrznego czy czegoś w tym stylu.

Schemat blokowy licznika w dłoń! Sygnały z enkodera należy doprowadzić do wejść dwóch kanałów licznika. Potem muszą trafić do bloczku *Encoder Interface*. Dochodzą tam tylko sygnały TI1FP1 i TI2FP2, stąd wniosek (skądinął słuszny), że sygnały z enkodera muszą być podpięte do kanałów 1 i 2 licznika. Niestety sygnałów z kanałów 3 i 4 nie da się doprowadzić do bloku sterownika licznika.

Myślę, że po tym co już zdziałaliśmy, ustawienie licznika w tryb enkodera nie będzie specjalnym problemem. RTFM! Dodatkowo w rozdziale *encoder interface mode* jest podany prosty przepis jak skonfigurować licznik do pracy w tym trybie.

Zadanie domowe 8.7: skonfigurować licznik do pracy w trybie enkodera. Ma zliczać impulsy i wywalić przerwanie co kilkanaście impulsów. A w przerwaniu niech przełącza diodę, bo to zacne, ładne i efektowne. Do dzieła a potem porównamy efekty :)

Przykładowe rozwiązanie (F103, enkoder na PA8 i PA9, dioda na PB0):

```
1. int main(void) {
2.
3.     RCC->APB2ENR = RCC_APB2ENR_IOPBEN | RCC_APB2ENR_IOPAEN | RCC_APB2ENR_TIM1EN;
4.     gpio_pin_cfg(GPIOB, PB0, gpio_mode_output_PP_2MHz);
5.     gpio_pin_cfg(GPIOA, PA8, gpio_mode_input_pull);
6.     gpio_pin_cfg(GPIOA, PA9, gpio_mode_input_pull);
7.     BB(GPIOA->ODR, PA8) = 1;
8.     BB(GPIOA->ODR, PA9) = 1;
9.
10.    TIM1->SMCR = TIM_SMCR_SMS_1;
11.    TIM1->CCMR1 = TIM_CCMR1_IC1F | TIM_CCMR1_IC2F;
12.    TIM1->CCER = TIM_CCER_CC1P;
13.    TIM1->ARR = 10;
14.    TIM1->DIER = TIM_DIER_UIE;
15.    TIM1->CR1 = TIM_CR1_CEN;
16.
17.    NVIC_ClearPendingIRQ(TIM1_UP_IRQn);
18.    NVIC_EnableIRQ(TIM1_UP_IRQn);
19.
20.    while (1);
21.
22. } /* main */
23.
24. __attribute__((interrupt)) void TIM1_UP_IRQHandler(void){
25.     if (TIM1->SR & TIM_SR UIF){
26.         TIM1->SR = ~TIM_SR UIF;
27.         BB(GPIOB->ODR, PB0) ^=1;
28.     }
29. }
```

Tym razem sygnały z zewnątrz nie są symulowane tak jak ostatnio. Znowu lenistwo wygrało. Łatwiej było znaleźć w szufladzie „żywy” enkoder niż napisać symulator :)

Początek chyba nie budzi wątpliwości. Enkoder jest podłączony do TIM1_CH1 i TIM1_CH2 (PA8 i PA9). Wejścia są podciagnięte do „plusa”, enkoder zwiera je do masy.

10) wybór trybu pracy licznika. Tryby enkoderowe są trzy:

- licznik zlicza impulsy tylko z pierwszego kanału a drugi kanał wykorzystuje jedynie do wyznaczenie kierunku obrotu
- licznik zlicza impulsy tylko z drugiego kanału a pierwszy wykorzystuje jedynie do detekcji kierunku
- licznik zlicza impulsy z obu kanałów z uwzględnieniem kierunku obrotu

W moim przykładzie zliczam impulsy tylko z jednego kanału. Dlaczego? Bo mam enkoder skokowy¹¹⁷ i na jeden krok przypadają dwa impulsy. Ja chciałbym aby jeden krok powodował inkrementację/dekrementację licznika o jeden. Uzyskałem to właśnie dzięki temu, że zliczam impulsy tylko z jednego kanału.

11) włączam filtrowanie... a co! kto mi zabroni?

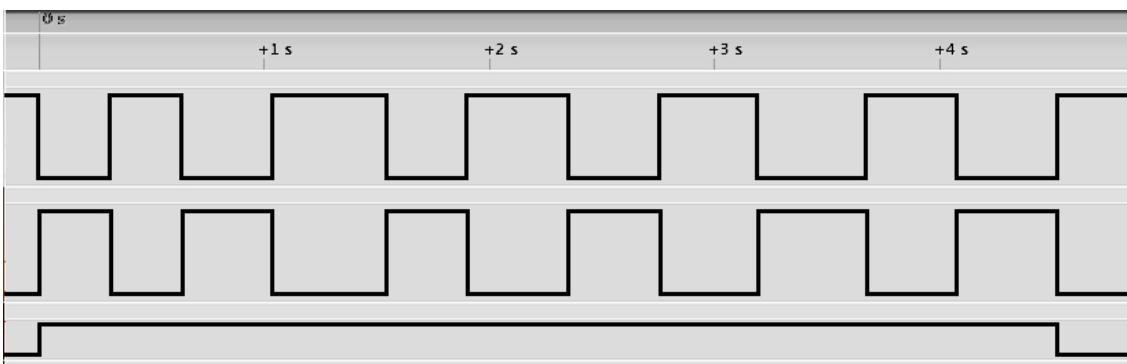
¹¹⁷ czy jak to się tam fachowo nazywa, taki trykający przy kręceniu (aretowany?)

12) tutaj zmieniam polaryzację jednego kanału. Czemu? Bo przy kręceniu osią enkodera w prawo licznik zliczał w dół. Po zanegowaniu jednego kanału kierunek zlicza się zmienił na taki jak chciałem.

13) dalej już bez zmian. Jak licznik doliczy do ARR to się przekręca i wywala przerwanie. W każdej chwili można oczywiście odczytać liczbę impulsów z rejestru TIM_CNT – polecam sobie podglądać w debuggerze.

17) a to tak dla urozmaicenia przykładu. W rozdziale 5.3 wspomniałem, że jeśli pojawi się przerwanie które nie będzie włączone w NVICu, to wejdzie w stan oczekiwania (pending). I teraz, jeżeli włączymy to przerwanie (oczekujące) w NVICu to zostanie ono natychmiast obsłużone. A nie zawsze tego chcemy! Jeżeli chcemy się zabezpieczyć przed taką sytuacją to należy kasować stan oczekiwania (linia 17 kodu) przed włączeniem przerwania w NVICu. Ale to tylko taki dodatek dla wzbogacenia przykładu :)

Niestety enkoder znalazłem jakiś paskudny i przebieg z niego jest mocno taki sobie. Grunt, że działa. Licznik zlicza 11 zboczy z jednego kanału i na następnym się przekręca. Wykrywanie kierunku też działa (tylko nie mam jak pokazać). Efekt na analizatorze (dwa kanały enkodera¹¹⁸ i dioda):



Rys. 8.4. Licznik zliczający impulsy enkodera

Co warto zapamiętać z tego rozdziału?

- patrz rozdział 8.9

8.8. Taktowanie licznika innym licznikiem

Zostało nam jeszcze jedno źródło sygnału taktującego licznik - inny licznik. Taka konfiguracja może być wykorzystana do stworzenia licznika o długości większej niż 16 bitów. Np. dwa liczniki połączone szeregowo to już będą 32 bity.

¹¹⁸ tam są przesunięcia między tymi prostokątami... serio! widać je niestety dopiero po rozciagnięciu przebiegu

Działanie tego trybu polega na tym, że licznik nadzorowany (master) wysyła sygnał wyjściowy - TRGO (np. przy UEV). Licznik podrzędny (slave) odbiera ten sygnał jako swój sygnał ITR (schemat blokowy Twoim przyjacielem). Odsyłam do RMa i rozdziału *Using one timer as prescaler for another* po szczegóły. Łączenie liczników zostanie omówione, z przykładem, troszkę później.

Co warto zapamiętać z tego rozdziału?

- patrz rozdział 8.9

8.9. Podsumowanie źródeł taktowania

W tym momencie kończymy omawianie *bloku zliczającego* i możliwych źródeł sygnału taktującego. Przechodzimy do innych bloków licznika, które mogą pracować niezależnie od wybranego źródła taktowania.

Z premedytacją pominąłem kilka zagadnień, według mnie mniej ważnych zagadnień... ale to tylko moja opinia. Raczej nie będę ich omawiał dokładniej w przyszłości, bo nie mogę omówić wszystkiego. Chciałbym kiedyś skończyć ten Poradnik :) Pominięte zagadnienia zebrałem w rozdziale 8.18.

Co warto zapamiętać z tych rozdziałów?

- UEV to zdarzenie, które występuje gdy licznik się przekręca
- UEV może generować przerwania i żądania DMA
- część rejestrów licznika może być buforowana
- rejestr preskalera buforowany jest **zawsze**, czy nam się to podoba czy nie
- OPM powoduje zatrzymanie licznika przy najbliższym UEV
- schemat blokowy Twoim przyjacielem!
- jeśli oswoimy się ze schematem blokowym to potrzebne informacje odnajdziemy o wiele szybciej, niż posługując się ciągłym tekstem
- licznik może być pędzony jednym z trzech sygnałów:
 - wewnętrznym zegarowym
 - zewnętrznym z nóżki kanału 1 lub 2 lub wejścia ETR
 - sygnałami z innego licznika (sygnały ITR)
- licznik może sprzętowo obsługiwać enkoder inkrementalny

- sporo przykładów dotyczących liczników można znaleźć w notach aplikacyjnych:
 - AN4013 *STM32F0, STM32F1, STM32F2, STM32F4, STM32L1 series, STM32F30x, STM32F3x8, STM32F373 lines timer overview*
 - AN2581 *STM32F10xxx TIM application examples*
 - AN2592 *How to achieve 32-bit timer resolution using the link system in STM32F10x and STM32L15x microcontrollers*

8.10. Blok porównujący - PWM

A więc mamy już licznik, który coś zlicza (żeby nie utrudniać sobie życia, w przykładach będzie to głównie zliczanie impulsów zegara wewnętrznego). Jednym z bajerów jakie można do niego dokleić, jest blok porównujący *compare*. Sprawa jest banalnie prosta. Po włączeniu tego bloku, układ będzie sprzętowo porównywał wartość rejestru TIM_CNT i wartość porównawczą podaną w rejestrze bloku compare (rejestr TIM_CCRx). Mamy cztery bloki porównujące, więc i cztery rejesty CCRx. Z każdego bloku porównującego wychodzi sygnał referencyjny OCxREF, zależny od wyniku porównania. O relacji między wynikiem porównania a sygnałem OCxREF decydują bity konfiguracyjne OCxM w rejestrze TIM_CCMR1¹¹⁹.

A po co nam to porównywanie? Ano sygnał OCxREF możemy wykorzystać do:

- generowania przebiegów na wyjściu mikrokontrolera
- zmieniania stanów wyjść mikrokontrolera
- generowania przerwań i żądań DMA
- sterowania licznikiem podrzędnym
- wyzwalania przetworników ADC i DAC
- pewnie do czegoś tam jeszcze się nadają...

Jedziemy z pierwszą kropką, czyli generowaniem przebiegów. Oczywiście chodzi tu o PWM. Działa to tak (jak zresztą każdy PWM...), że stan nóżki wyjściowej jest zależny od wyniku porównania rejestrów licznika i bloku porównującego (czyli od sygnału referencyjnego). Przykładowo założmy, że nóżka ma stan niski wtedy, gdy CNT < CCRx oraz:

- ARR = 100
- CCRx = 50
- CNT = 0

¹¹⁹ patrz schemat blokowy: *Output stage of capture/compare channel*

Licznika zlicza od zera do góry, zrównuje się z CCRx. W tym momencie stan nóżki się zmienia na wysoki. Licznik zlicza dalej, aż do wartości ARR. Następuje przekręcenie licznika i znowu zlicza od zera. Po przekręceniu licznika CNT znowu jest mniejsze od CCRx, czyli nóżka jest w stanie niskim. I tak w kółko. Rejestry CCRx i ARR determinują odpowiednio wypełnienie i częstotliwość sygnału PWM.

Zachowanie PWMa zależy od wybranego trybu pracy (*PWM Mode 1* lub *PWM Mode 2* - właściwie to ja nie widzę sensu istnienia tych dwóch trybów, ale o tym będzie za chwilę...) oraz konfiguracji licznika (zliczanie w góre, w dół, symetryczne). Ustawienia te determinują związek między wynikiem porównania rejestrów CNT i CCRx a sygnałem OCxREF. Po szczegóły odsyłam do opisów i przebiegów pokazanych w dokumentacji. Mnie się nie chce tego analizować więc tylko podrzucę kilka zrzutów z różnymi trybami, żeby każdy sobie sam obaczył różnice. Ale to pod koniec rozdziału :) No to tyle wstęp.

Zadanie domowe 8.8: generator PWM. Częstotliwość 10kHz. Wypełnienie:

- kanał 1 - 50%
- kanał 2 - 90%
- kanał 3 - 10%

Kto się czuje na siłach niech zaczyna (czas start! tik, tak), kto nie - niech przeczyta jeszcze kawałek.

Schemat blokowy Twoim przyjacielem! Szukamy więc schematu *Output stage of capture/compare channel*¹²⁰. I, podobnie jak miało to miejsce przy jakimś tam poprzednim przykładzie, stworzymy sobie listę bitów którym przyjrzymy się bliżej. Do dzieła:

- wynik porównania rejestrów CNT i CCRx trafia do czarnej skrzynki o nazwie *Output Mode Controller*
- czarna skrzynka jest konfigurowana przez bity OC1CE i OC1M w rejestrze TIM_CCMR1 (jedynki w nazwach bitów i rejestrów odnoszą się do pierwszego kanału compare)
- dalej mamy sygnał OC1REF i jakiś kociokwik...
- popatrzmy od drugiej strony: na schemacie ogólnym znajdzmy wyjście pierwszego kanału (nóżkę) i zobaczymy jaki sygnał tam dochodzi - OC1
- teraz wiemy gdzie chcemy dojść w tym bałaganie
- czas martwy nam nie potrzebny, więc sygnał OC1REF może przejść „nad” generatorem czasu martwego do multipleksera

¹²⁰ schemat ogólny licznika też się przydaje - warto go sobie wydrukować żeby był zawsze pod ręką!

- multiplekser jest konfigurowany bitami CC1E¹²¹ w rejestrze TIM_CCER
- na multiplekserze podali nam nawet wartości bitów CC1E dla poszczególnych wejść (nas interesuje opcja 0b01)
- dalej jest negator i kolejny multiplekser
- multiplekserem wybieramy czy chcemy zanegować nasz sygnał¹²² - bity CC1P w CCER
- na końcu mamy czarną skrzynkę - jakiś bufor wyjściowy - i całą gamę bitów: CC1E w CCER oraz MOE, OSS1, OSSR w BDTR

Uff. Podsumujmy:

- OC1CE i OC1M w CCMR1
- CC1E w CCER
- CC1P w CCER
- CC1E w CCER - WTF? to już było :)
- MOE, OSS1, OSSR w BDTR

Do pełni szczęścia na pewno dojdą jeszcze, znane nam już, rejesty:

- PSC - preskaler
- ARR - rejestr przeładowania
- CCR1,2,3 - rejesty bloków porównawczych
- CR1 - rejestr konfiguracyjny (włączenie licznika)

No to teraz czas odpalić opis rejestrów i doczytać szczegółowo. Widzimy się jak PWM będzie działać! Aha moment. A co z wzorem na częstotliwość i współczynnik wypełnienia? Cóż... czy to AVR, czy STM... wszystkie liczniki rządzą się tymi samymi prawami i 32b STM nic tu nie zmienia:

$$f_{PWM} = \frac{f_{TIM}}{(PSC + 1) \cdot (ARR + 1)}$$

$$d_{PWM} = \frac{CCRx}{ARR + 1} \cdot 100 \%$$

Do dzieła mój Szogunie!

121 bity CC1NE dotyczą nóżki komplementarnej, olewamy je na razie

122 właśnie dlatego wcześniej pisałem że PWM Mode 2 jest bez sensu, sygnał można zanegować w kilku miejscach co tylko wprowadza bajzel... albo ja czegoś nie rozumiem... (zaraz się wyjaśni)

Przykładowe rozwiązanie (F103, PWM na nóżkach PA8, PA9, PA10):

```
1. int main(void) {
2.
3.     RCC->APB2ENR = RCC_APB2ENR_IOPBEN | RCC_APB2ENR_IOPAEN | RCC_APB2ENR_TIM1EN;
4.     gpio_pin_cfg(GPIOA, PA8, gpio_mode_alternate_PP_2MHz);
5.     gpio_pin_cfg(GPIOA, PA9, gpio_mode_alternate_PP_2MHz);
6.     gpio_pin_cfg(GPIOA, PA10, gpio_mode_alternate_PP_2MHz);
7.
8.     TIM1->CCMR1 = TIM_CCMR1_OC1PE | TIM_CCMR1_OC1M_2 | TIM_CCMR1_OC1M_1;
9.     TIM1->CCMR1 |= TIM_CCMR1_OC2PE | TIM_CCMR1_OC2M_2 | TIM_CCMR1_OC2M_1;
10.    TIM1->CCMR2 = TIM_CCMR2_OC3PE | TIM_CCMR2_OC3M_2 | TIM_CCMR2_OC3M_1;
11.
12.    TIM1->CCER = TIM_CCER_CC1E | TIM_CCER_CC2E | TIM_CCER_CC3E;
13.    TIM1->BDTR = TIM_BDTR_MOE;
14.
15.    TIM1->PSC = 7;
16.    TIM1->ARR = 99;
17.    TIM1->CCR1 = 50;
18.    TIM1->CCR2 = 90;
19.    TIM1->CCR3 = 10;
20.
21.    TIM1->EGR = TIM_EGR_UG;
22.    TIM1->CR1 = TIM_CR1_ARPE | TIM_CR1_CEN;
23.
24.    while (1);
25.
26. } /* main */
```

3) zegary dla portu A i B oraz TIM1. Port B nie jest potrzebny, się zapłatał przez przypadek :)

4, 5, 6) konfiguracja nóżek: trzy kanały licznika TIM1 - wyjścia funkcji alternatywnych

Jako ciekawostkę powiem, że do tej chwili (gdy to piszę), byłem przekonany że aby korzystać z alternatywnej konfiguracji portów (np. wyjścia PWM) konieczne jest włączenie zegara dla bloku AFIO¹²³. Ale właśnie pisząc ten przykładowy program, zapomniałem o tym zegarze i... program działał ok. Zacząłem drążyć temat i doczytałem, że AFIO to tylko do remapu i EXTI. Także żeby nie było – ja też się uczę dzięki temu poradnikowi :)

8) czas na konfigurację licznika, kanał pierwszy. Z ciekawostek włączam buforowanie rejestru porównującego (bity OCxPE). Nie jest to potrzebne, ale tak jest edukacyjniej. Jak ktoś nie pamięta to: rozdział 8.3.

9) to samo dla drugiego kanału. Uwaga na sumę logiczną przy wpisywaniu tych ustawień, żeby nie nadpisać poprzedniej linijki :)

10) i to samo dla trzeciego kanału. Uwaga na zmianę rejestru (CCMR1 - kanały 1 i 2; CCMR2 - kanały 3 i 4)

12) włączenie trzech kanałów

13) MOE (Main Output Enable) - to zasługuje na dłuższe oględzenie. Będzie za chwilę ([funkcja break](#)).

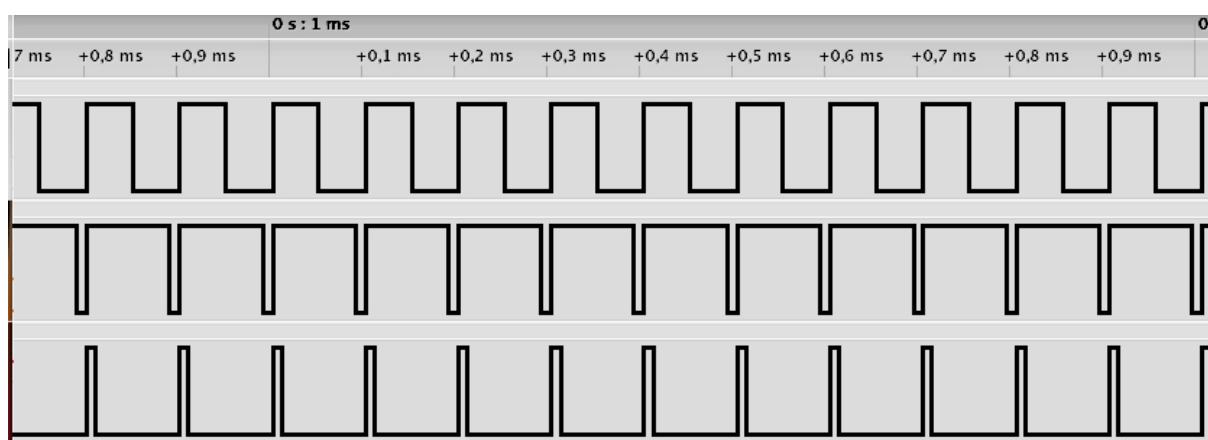
123 niby blok *Alternate Function...*

15) zaczyna się „czasowa” konfiguracja licznika. Wzorek na częstotliwość podałem wcześniej. Wartość rejestru ARR warto dobrąć tak, aby potem łatwo dało się przeliczyć współczynnik wypełnienia z wyrażonego (np.) w procentach.

17, 18, 19) współczynniki wypełnienia dla kolejnych kanałów lądują w odpowiadających im rejestrach porównawczych

21) tutaj jest programowo generowany UEV. Po co? Bo włączyłem buforowanie rejestrów CCRx, więc ich wartości zostaną zapisane dopiero przy UEV. Bez programowego wymuszenia UEV, nowe wartości rejestrów zadziałyłyby dopiero przy UEV wynikającym z „naturalnego” przekręcenia licznika.

22) włączenie licznika i buforowania ARR – czemu? a czemu nie :)



Rys. 8.5 Wyjścia PWM (od góry: CH1 - 50%, CH2 - 90%, CH3 - 10%)

Skomplikowane? Niby nie, ale łatwo coś przegapić. Co by nie powiedzieć, schematy blokowe pomagają, bo wiadomo którym bitom się przyjrzeć dokładniej. Kto by wpadł np. na to nieszczęsne MOE :)

Liczniki zaawansowane w STM32 mają kilka funkcji wspomagających ich wykorzystanie przy sterowaniu energoelektroniką (np. jakimiś układami mostkowymi). Do takich funkcji należą:

- wyjścia komplementarne
- funkcja break

Wyjście komplementarne działają w ten sposób, że wyjście zanegowane przyjmuje zawsze przeciwny stan niż komplementarne z nim wyjście nie-zanegowane. Przy czym ta przeciwność jest konfigurowalna... możemy ustawić tak, że oba będą naraz w stanie wysokim. Wszystko zależy na czym nam zależy, czym sterujemy. Można to wykorzystać np. przy sterowaniu dwoma

tranzystorami gałęzi mostka. Mamy nawet do dyspozycji programowalny generator czasu martwego. Temat wielce intrigujący, ale koniec z tym bo już szykuje się kolejny OT. Niech zainteresowani rozpłyną się w dokumentacji samodzielnie :)

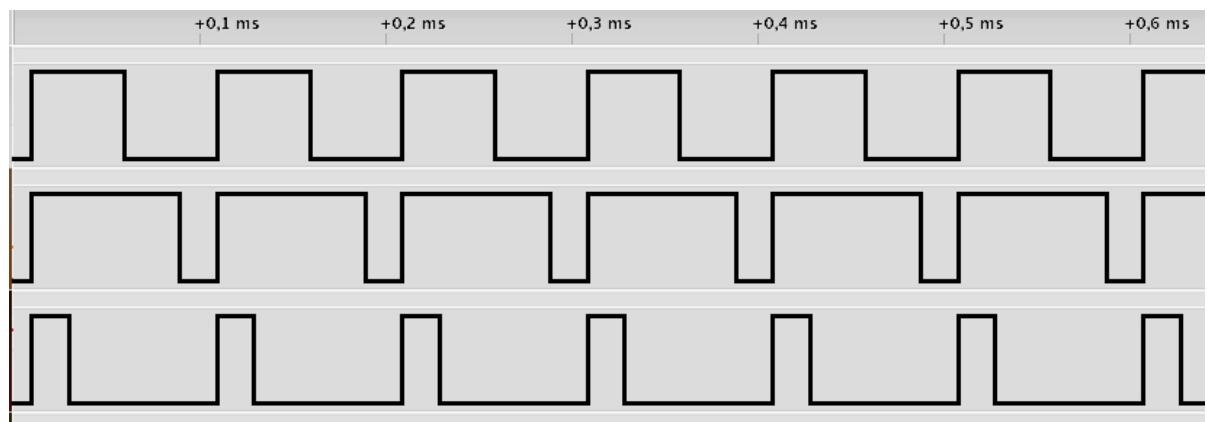
Funkcja BREAK znowu jest ukłonem w stronę energoelektroniki. To taki wyłącznik awaryjny generowania sygnałów. Po odebraniu sygnału BREAK następuje sprzętowe wyłączenie wyjść licznika i wymuszenie na nich bezpiecznych stanów. Bezpieczne stany wyjść konfiguruje się gdzieś w opcjach licznika w zależności od tego czym sterujemy i co ma się pojawić na wyjściach (stan niski/wysoki) w sytuacji awaryjnej. Do tego jest cała masa innych bajerów (generowanie przerwania przy sygnale break, konfiguracja czy po zaniku sygnału break układ ma wracać do pracy automatycznie czy pozostać w stanie bezpiecznym...). A skąd ten break? Z:

- pinu *break input* - np. od jakiegoś zewnętrznego układu kontroli prądu, temperatury, wyłącznika awaryjnego, itd...
- z układu kontroli zegara (CSS) - sygnał break pojawi się w przypadku uszkodzenia zewnętrznego rezonatora (*Clock Failure Event*)
- można też wymusić programowo... jak wszystko

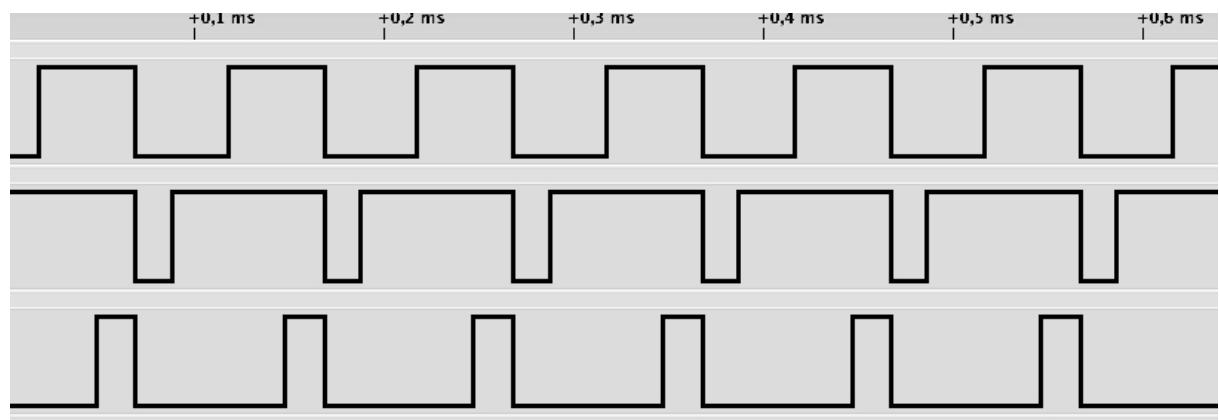
I teraz dochodzimy do bitu **TIM_BDTR_MOE**. To jest taki główny wyłącznik wyjść licznika. Działanie funkcji break polega właśnie na skasowaniu tego bitu, co wyłącza wyjścia. Jest on również **domyślnie skasowany po resecie**. Właśnie dlatego musimy go ustawić, mimo że w naszym przykładzie nie korzystamy z funkcji break.

Swoją drogą wejście ETR też może wymusić określony stan na wyjściach: funkcja nazywa się *Clearing the OCxREF signal on an external event*. Do doczytania we własnym zakresie. Koniec OT.

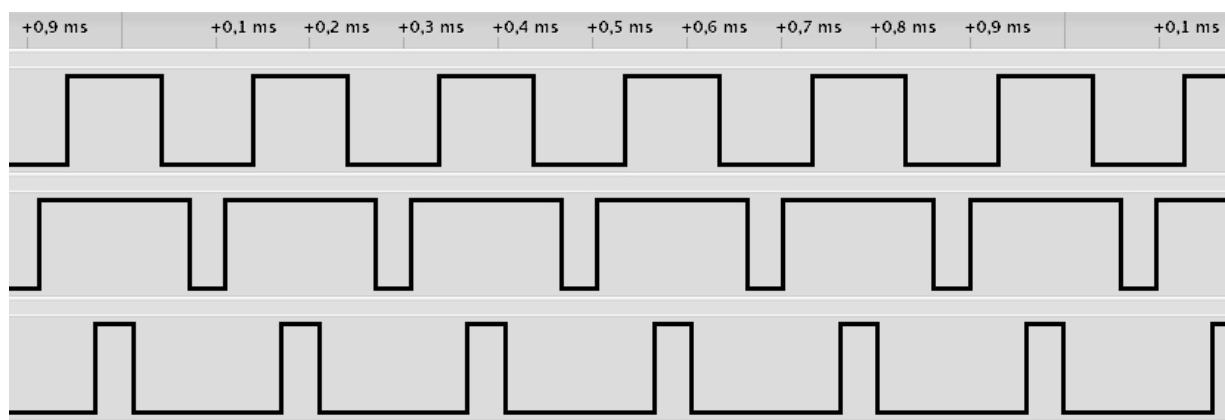
Żeby wykręcić się z dokładnego omawiania trybów pracy PWM, obiecałem pokazać przebiegi PWM dla różnych trybów. No i dotrzymuję słowa :) Trzy kanały PWM (50%, 80%, 20%) i różne tryby. Międz sobą zabawy w szukanie różnic na obrazkach :)



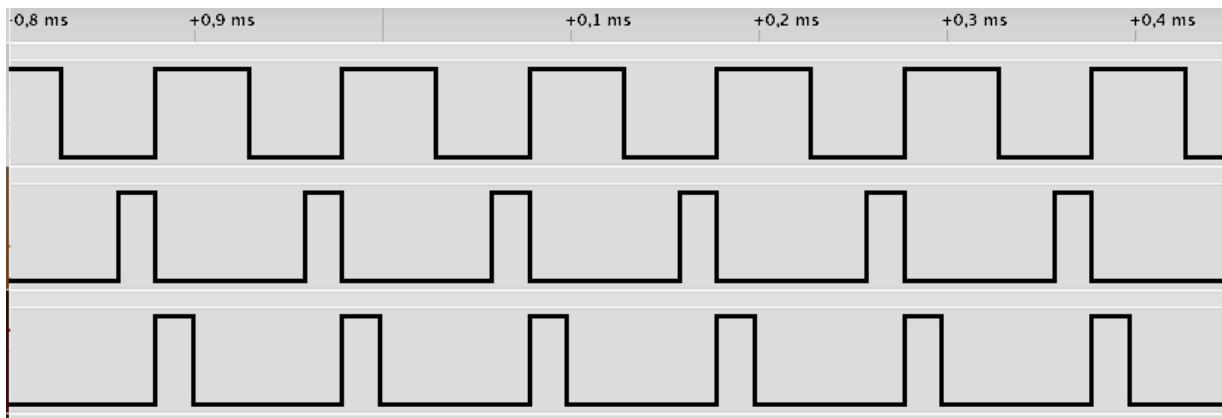
Rys. 8.6 Licznik zliczający do góry, wszystkie kanały w trybie PWM Mode 1
(współczynniki wypełnienia 50%, 80%, 20%)



Rys. 8.7 Licznik zliczający w dół, wszystkie kanały w trybie PWM Mode 1
(współczynniki wypełnienia 50%, 80%, 20%)



Rys. 8.8 Licznik zliczający symetrycznie, wszystkie kanały w trybie PWM Mode 1
(współczynniki wypełnienia 50%, 80%, 20%)



Rys. 8.9 Licznik zliczający do góry, kanały 1 i 3 w *PWM Mode 1*, kanał 2 w *PWM Mode 2*
(współczynniki wypełnienia 50%, 80%, 20%)

Co można zauważyć?

- przy zliczaniu w góre wszystkie przebiegi mają zsynchronizowane zbocze rosnące
- przy zliczaniu w dół wspólną fazę mają zbocza opadające
- przy zliczaniu symetrycznym... no widać przecież
- włączenie *PWM Mode 2* spowodowało coś w rodzaju zanegowania kanału

Dalej nie widzę sensu istnienia tych dwóch trybów (*Mode 1* i *Mode 2*). Zanegować przebieg to sobie można bitami `TIM_CCMRx_OCxP` jeśli mnie pamięć nie myli...

Co warto zapamiętać z tego rozdziału?

- patrz rozdział 8.12

8.11. Blok porównujący - przerwania

Do czego jeszcze można wykorzystać bloki *compare*? Np. do generowania przerwań w momencie zrównania się wartości rejestru licznika i rejestru porównawczego. Pragnę przypomnieć, że bloki *compare* działają zupełnie niezależnie od źródła taktowania licznika. Licznik może współpracować np. z enkoderem podpiętym do wału maszyny, a bloki porównujące w określonych położeniach wału będą odpalać przerwania. Możliwości są nieograniczone :) To może coś w tym guście właśnie!

Zadanie domowe 8.9: enkoder obrotowy skokowy. Licznik zlicza maksymalnie 14 skoków enkodera. Jeżeli wartość licznika wynosi 10 to zapala się led. Jeżeli wynosi 8 to led gaśnie. Dodatkowo jakiś przycisk powoduje wyzerowanie licznika – taka symulacja obecności indeksującego kanału enkodera. Podpowiem, że najprościej to będzie skopiować projekt z enkoderem i dopisać do niego konfigurację bloków porównujących. Czas start :)

Przykładowe rozwiązanie (F103, enkoder na PA8 i PA9, przycisk PB2, dioda: PB0):

```

1. int main(void) {
2.
3.     RCC->APB2ENR = RCC_APB2ENR_IOPAEN | RCC_APB2ENR_IOPBEN | RCC_APB2ENR_TIM1EN |
4.             RCC_APB2ENR_AFIOEN;
5.
6.     gpio_pin_cfg(GPIOB, PB0, gpio_mode_output_PP_2MHz);
7.     gpio_pin_cfg(GPIOB, PB2, gpio_mode_input_floating);
8.     gpio_pin_cfg(GPIOA, PA8, gpio_mode_input_pull);
9.     gpio_pin_cfg(GPIOA, PA9, gpio_mode_input_pull);
10.    BB(GPIOA->ODR, PA8) = 1;
11.    BB(GPIOA->ODR, PA9) = 1;
12.
13.    TIM1->DIER = TIM_DIER_CC1IE | TIM_DIER_CC2IE;
14.    TIM1->CCR1 = 9;
15.    TIM1->CCR2 = 7;
16.
17.    TIM1->SMCR = TIM_SMCR_SMS_1;
18.    TIM1->CCMR1 = TIM_CCMR1_IC1F | TIM_CCMR1_IC2F;
19.    TIM1->CCER = TIM_CCER_CC1P;
20.    TIM1->ARR = 13;
21.    TIM1->CR1 = TIM_CR1_CEN;
22.
23.    AFIO->EXTICR[0] = AFIO_EXTICR1 EXTI2_PB;
24.    EXTI->IMR = EXTI_IMR_MR2;
25.    EXTI->RTSR = EXTI_RTSR_TR2;
26.
27.    NVIC_EnableIRQ(TIM1_CC_IRQn);
28.    NVIC_EnableIRQ(EXTI2_IRQn);
29.
30.    while (1);
31.
32. } /* main */
33.
34. __attribute__((interrupt)) void EXTI2_IRQHandler(void) {
35.     if (EXTI->PR & EXTI_PR_PR2) {
36.         EXTI->PR = EXTI_PR_PR2;
37.         TIM1->EGR = TIM_EGR_UG;
38.     }
39. }
40.
41. __attribute__((interrupt)) void TIM1_CC_IRQHandler(void){
42.     if (TIM1->SR & TIM_SR_CC1IF){
43.         TIM1->SR = ~TIM_SR_CC1IF;
44.         BB(GPIOB->ODR, PB0) = 1;
45.     }
46.
47.     if (TIM1->SR & TIM_SR_CC2IF){
48.         TIM1->SR = ~TIM_SR_CC2IF;
49.         BB(GPIOB->ODR, PB0) = 0;
50.     }
51. }
```

To chyba najdłuższy przykład jak dotąd. Ale większość kodu się powtarza - aż do 13 linijki nie ma nic nowego.

13) włączam dwa przerwania od kanałów porównujących 1 i 2. W kolejnych linijkach ustawiam wartości porównawcze. Licznik liczy od 0 stąd wartości są o 1 mniejsze od tych z treści zadania.

17 - 21) skopiowany kod programu z przykładu z enkoderem. Tylko wyrzuciłem przerwanie od UEV i zmieniłem rozdzielcość.

23 - 25) przycisk obsługuje przez przerwanie¹²⁴, konfiguracja przerwania zewnętrznego też skopiowana z poprzednich przykładów. Wszystko już było :)

34) tu jest coś ciekawego! Przycisk miał zerować timer, jednak zamiast operacji typu TIM_CNT = 0, programowo wymuszam UEV. Jak ktoś nie pamięta to proszę wrócić... o tu: rozdział 8.3. Taka re-inicjalizacja licznika. Przypominam o haczyku: to czy rejestr CNT przy UEV się wyzeruje czy przyjmie wartość ARR zależy od aktualnego kierunku zliczania (wartości bitu TIM_CR1_DIR).

41) warto zwrócić uwagę na to, że przerwanie od kanałów porównujących to inny wektor niż od UEV. Programowo trzeba sprawdzić, który z kanałów się wyzwolił i skasować flagę. Dalej nic nowego nie ma.

Co warto zapamiętać z tego rozdziału?

- patrz rozdział 8.12

8.12. Blok porównujący - podsumowanie

O blokach *compare* już wystarczy. W przykładach pokazałem dwie IMHO najważniejsze ich możliwości:

- generowanie sygnału PWM
- generowanie przerwań

Co pominąłem z premedytacją i nie bardzo żałuję:

- to, że sygnał OCxREF można wykorzystać do zmiany stanu wyjścia; ale nie do generowania PWM tylko aby jednorazowo ustawić / zanegować albo skasować wyjście (patrz bity OCxM w rejestrze TIM_CCMR1)
- to, że sygnał OCxREF można wykorzystać do sterowania innym, podrzędnym licznikiem (patrz bity MMS w rejestrze TIM_CR2)

124 wiem wiem, tak się nie robi...

- to, że można programowo wymusić określony stan sygnału OCxREF niezależnie od stanu porównania (patrz bity OCxM w rejestrze TIM_CCMR1¹²⁵)
- pewnie coś by się jeszcze znalazło...

Co warto zapamiętać z tych rozdziałów?

- schemat blokowy Twoim przyjacielem!
- bloki porównujące sprzętowo porównują chwilową wartość rejestru licznika i wartości z rejestrów TIM_CCRx
- wynik porównywania można wykorzystać m.in. do:
 - generowania przebiegów PWM
 - generowania przerwań
 - sterowania wyprowadzeniami mikrokontrolera i licznikami podrzędnymi
- bloki porównujące działają niezależnie od tego, z jakiego źródła jest taktowany licznik (sygnał zegarowy, impulsy z zewnątrz, enkoder, ...)

8.13. Blok przechwytyujący - Input Capture

Właściwie to trochę oszukuję z tym rozgraniczaniem na osobne bloki *compare* i *capture*. To są te same bloki tylko mogą pracować w jednym albo drugim trybie - w obu naraz niet :)

Paczamy na schemat blokowy licznika i *Capture/compare channel* (*example: channel 1 input stage*). Działanie bloków kapturowych (od *capture*) polega na tym, że po pojawienniu się sygnału kapturującego ICxPS¹²⁶ aktualna wartość licznika CNT jest zatrzaszkowana w rejestrze CCRx danego kanału. To są te same rejesty, które wykorzystywaliśmy przy blokach porównujących. Do tego mamy całą gamę bajerów typu generowanie przerwań czy żądań DMA. Prościzna. Szczególnie, że w rozdziale *Input Capture Mode* jest ładny przepis co, krok po kroku, ustawić.

Zadanie domowe 8.10: timer sobie zlicza, naciśnięcie przycisku powoduje zapisanie aktualnej wartości rejestru timera do rejestru CCR1. Uprzedzam, że to będzie strasznie skomplikowane... jak wszystko w STMach, bo operacje na 32 bitowych rejestrach są trudne... Jak ktoś się nie czuje na siłach to czyta jeszcze kawałek :)

125 przypominam, że rejesty CCMR mają w RMie dwa odrębne opisy: jeden dla trybu *Output Compare*, drugi dla *Input Capture*

126 *Input Capture*, końcówka PS bo za preskalerem

Na początek analiza schematu blokowego obwodu wejściowego, tak na szybko, bo już tyle razy to robiliśmy:

- sygnał z wejścia CH1 wchodzi na filtr i detektor zboczy
- zbocze wybieramy bitami CC1P w TIM_CCER (przy czym jeśli pasuje nam domyślne ustawienie to nie ma oczywiście parcia żeby to zmieniać)
- potem jest multiplekser CC1S w TIM_CCMR1
- jakiś dzielnik (ICPS w TIM_CCMR1 oraz CC1E w TIM_CCER)
- i mamy sygnał IC1PS o który nam chodziło

Przykładowe rozwiązanie (F103, przycisk PA8):

```
1. int main(void) {
2.     RCC->APB2ENR = RCC_APB2ENR_IOPAEN | RCC_APB2ENR_TIM1EN;
3.
4.     gpio_pin_cfg(GPIOA, PA8, gpio_mode_input_pull);
5.     BB(GPIOA->ODR, PA8) = 1;
6.
7.     TIM1->CCMR1 = TIM_CCMR1_CC1S_0;
8.     TIM1->CCER = TIM_CCER_CC1E;
9.
10.    TIM1->PSC = UINT16_MAX;
11.    TIM1->ARR = UINT16_MAX;
12.
13.    TIM1->CR1 = TIM_CR1_CEN;
14.
15.    while (1);
16.
17. }
18. /* main */
```

No koszmar po prostu. Bez biblioteki nie podchodź. Nawet nie bardzo jest co tu omawiać... Preskaler i ARR ustawione na maks, żeby licznik liczył wolno i długo. Wartość CNT zapisywana jest w CCR1. Podglądałem debuggerem – działa.

Jak można by to skomplikować?

- można włączyć generowanie przerwania lub żądania DMA po odebraniu sygnału *capture*
- można dodać filtrowanie wejścia *capture* tak jak to robiliśmy przy zewnętrznym taktowaniu
- można włączyć preskaler na wejściu *capture*, wtedy zakapturzenie wartości będzie się odbywało nie przy pierwszym zboczu na *capture* tylko przy n-tym
- można sobie zmienić polaryzację wejścia *capture*
- oczywiście można mieszać *capture* i *compare* na **różnych** kanałach (w sumie mamy cztery kanały do zabawy)

Jak zawsze zachęcam do prób i studiowania dokumentacji. Nic trudnego w tym nie ma. W kolejnym rozdziale połączymy kapturzenie z kontrolerem licznika¹²⁷ i będzie ciekawszy przykład.

Co warto zapamiętać z tego rozdziału?

- funkcja przechwytywania (*capture*) polega na tym, że na skutek jakiegoś zdarzenia aktualny stan rejestru licznika jest zapisywany w rejestrze bloku przechwytyjącego

8.14. Synchronizacja sygnałem zewnętrznym

External Trigger Synchronization (synchronizacja zewnętrznym trygierzem) polega na sterowaniu pracą licznika poprzez jakiś sygnał zewnętrzny (z poza licznika). Odpowiada za to *Slave Mode Controller*. Zewnętrzny trygierz może w szczególności:

- resetować licznik (żeby zliczał od zera)
- bramkować zliczanie (tzn. kiedy sygnał jest aktywny licznik zlicza, kiedy jest nieaktywny licznik nie zlicza)
- wyzwalać licznik (rozpoczynać zliczanie)

Co może być tym sygnałem?

- sygnał wejściowy z kanału 1 lub 2 licznika, bo tylko te kanały są połączone z *Slave Mode Controllerem* (patrz schemat blokowy)
- zdarzenie pochodzące od innego licznika (sygnał TRGO z innego licznika) – o tym więcej w rozdziale o synchronizacji liczników (rozdział 8.16)

W poprzednim rozdziale obiecałem frapujące przykłady¹²⁸... Tak na szybko przyszło mi do głowy coś takiego:

- sprzętowy pomiar okresu sygnału zewnętrznego (np. pomiar prędkości silnika na podstawie impulsów z enkodera)
- opóźniona reakcja na sygnał z zewnątrz (pojawia się sygnał zewnętrzny, czekamy x czasu i np. zapalamy diodę)

127 *Slave Mode Controller* - nie wiem jak to zgrabnie spolszczyć - nadzorca niewolnika?

128 no i teraz masz babo placek...

Punkt pierwszy to pomiar okresu. Założymy, że z zewnątrz przychodzi jakiś sygnał (np. jeden impuls na obrót wału maszyny), a my chcemy znać okres tego sygnału (prędkość obrotową wału maszyny). Koncepcja jest taka:

- uruchamiamy timer, który liczy czas
- doklejamy do niego funkcję *capture* - impuls z zewnątrz będzie zatrzaskiwał rejestr licznika (czyli zliczony czas)
- *Slave Mode Controller* konfigurujemy tak aby ten sam sygnał, który wyzwalał kapturowanie, powodował również restart licznika i zliczanie od zera

Czyli: licznik zlicza okres sygnału, przychodzi impuls (czy tam zbocze) i zliczony czas zostaje zapisany a licznik zresetowany. Licznik zlicza znowu od zera aż do następnego impulsu. Tym sposobem cały czas mamy zakapturzoną informację o długości ostatniego okresu sygnału. I to w pełni sprzętowo! Rdzeń można uśpić :) No to startujemy:

Zadanie domowe 8.11: sprzętowy pomiar okresu sygnału z zewnątrz. Podpowiedź: skopiować przykład z *Input Capture Mode* i dodać jedną linię kodu¹²⁹:

Przykładowe rozwiązanie (F103, Capture na kanale CH1 - PA8):

```

1. int main(void) {
2.
3.     RCC->APB2ENR = RCC_APB2ENR_IOPAEN | RCC_APB2ENR_TIM1EN;
4.     gpio_pin_cfg(GPIOA, PA8, gpio_mode_input_pull);
5.     BB(GPIOA->ODR, PA8) = 1;
6.
7.     TIM1->CCMR1 = TIM_CCMR1_CC1S_0;
8.     TIM1->CCER = TIM_CCER_CC1E;
9.
10.    TIM1->SMCR = TIM_SMCR_SMS_2 | TIM_SMCR_TS_2 | TIM_SMCR_TS_1;
11.    TIM1->PSC = UINT16_MAX;
12.    TIM1->ARR = UINT16_MAX;
13.    TIM1->CR1 = TIM_CR1_CEN;
14.
15.    while (1);
16.
17. } /* main */

```

Jedyna różnica to nowa linijka numer 10. Powoduje ona dwie rzeczy:

- ustawienie licznika w tryb *Slave Reset Mode* (bit SMS): po odebraniu sygnału TRGI licznik się zresetuje
- wybranie źródła sygnału TRGI dla kontrolera Slave'a (bit TS): wybrano TI1FP1

129 ach te skomplikowane STMy

Drugi punkt programu: opóźniona reakcja na sygnał z zewnątrz. Plan działania jest taki:

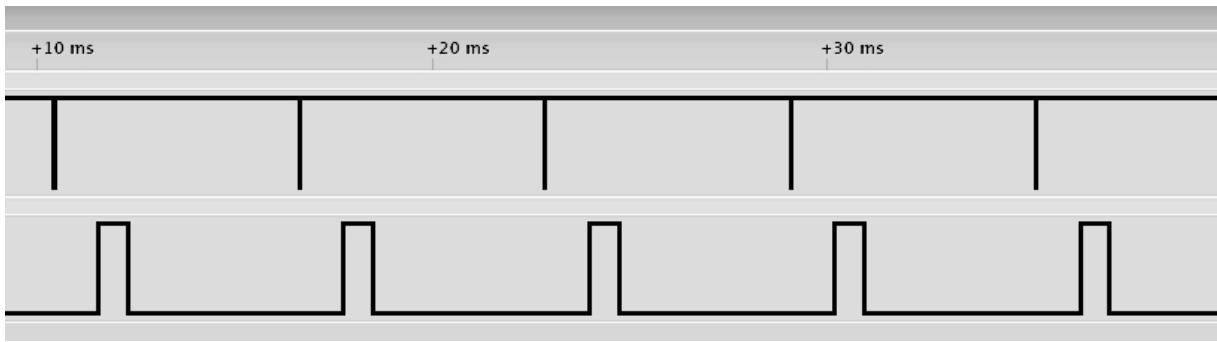
- konfigurujemy licznik w trybie *Slave Trigger Mode* - czyli będzie czekał na sygnał zewnętrzny który go wystartuje
- dodatkowo dorzucamy *One Pulse Mode* - żeby działał tylko raz po wyzwoleniu
- licznik ma generować krótki impuls na wyjściu kanału numer 2 (tryb PWM)

Zadanie domowe 8.12: na wejście *Input Capture CH1* przychodzi impuls z zewnątrz. Po małej zwłoce licznik generuje impuls określonej długości na wyjściu CH2. Do dzieła. Tym razem nie pomagam :)

Przykładowe rozwiązanie (F103, wejście - PA8, wyjście - PA9):

```
1. int main(void) {
2.     RCC->APB2ENR = RCC_APB2ENR_IOPAEN | RCC_APB2ENR_TIM1EN;
3.
4.     BB(GPIOA->ODR, PA8) = 1;
5.     gpio_pin_cfg(GPIOA, PA8, gpio_mode_input_pull);
6.     gpio_pin_cfg(GPIOA, PA9, gpio_mode_alternate_PP_2MHz);
7.
8.     TIM1->SMCR = TIM_SMCR_SMS_2 | TIM_SMCR_SMS_1 |
9.                  TIM_SMCR_TS_2 | TIM_SMCR_TS_0;
10.
11.    TIM1->CCMR1 = TIM_CCMR1_CC1S_0
12.        | TIM_CCMR1_OC2M_2 | TIM_CCMR1_OC2M_1 | TIM_CCMR1_OC2M_0;
13.    TIM1->CCER = TIM_CCER_CC1P | TIM_CCER_CC2E;
14.    TIM1->BDTR = TIM_BDTR_MOE;
15.
16.
17.    TIM1->CCR2 = 9000;
18.    TIM1->ARR = 15000;
19.
20.    TIM1->CR1 = TIM_CR1_OPM;
21.
22.    SysTick_Config(50000);
23.
24.    while (1);
25.
26. } /* main */
27.
28. __attribute__((interrupt)) void SysTick_Handler(void){
29.     BB(GPIOA->ODR, PA8) = 0;
30.     __NOP(); __NOP(); __NOP(); __NOP(); __NOP();
31.     __NOP(); __NOP(); __NOP(); __NOP(); __NOP();
32.     __NOP(); __NOP(); __NOP(); __NOP(); __NOP();
33.     __NOP(); __NOP(); __NOP(); __NOP(); __NOP();
34.     BB(GPIOA->ODR, PA8) = 1;
35. }
```

Znowu wykorzystałem „trik” z mechanizmem pinem w przerwaniu SysTicka. Analizę pozostawiam Czytelnikowi. Jedynie zwróci uwagę na to, że w tym przykładzie nie jest ustawiany bit włączający licznik (TIM_CR1_CEN). To dlatego, że za włączenie licznika odpowiada *Slave Controller* (tryb *Trigger Mode*). Efekt:



Rys. 8.10. Opóźniona reakcja (dolny przebieg) na zewnętrzne impulsy (górnny przebieg)

W przerwaniu SysTick użyłem pustych instrukcji (*nop*) w celu uzyskania krótkiego opóźnienia. To nie jest najszczepliwsze rozwiązanie. Cortex jest na tyle inteligentną bestią, że może olać instrukcje *nop* (nic nie robią). Używanie *nopów* do uzyskania opóźnienia nie jest zalecane... tu akurat zadziałało, ale nie ma gwarancji! Już lepsza byłaby jakaś instrukcja barierowa (np. *dsb*).

Co warto zapamiętać z tego rozdziału?

- synchronizacja zewnętrznym trygierzem pozwala na sterowanie pracą licznika sygnałem spoza tego licznika
- możliwe jest: resetowanie, bramkowanie i wyzwalanie licznika sygnałem zewnętrznym
- sygnał może pochodzić z jednego z wejść licznika (kanał 1 lub 2) lub od innego licznika
- synchronizację można połączyć np. z przechwytywaniem co pozwala czysto sprzętowo mierzyć (dajmy na to) okres sygnału

8.15. PWM Input Mode

Tytułowy *PWM Input Mode* to taka wariacja na temat *Input Capture Mode*. Pozwala mierzyć okres i wypełnienie sygnału PWM. Działa to tak, że jedno wejście jest połączone z dwoma kanałami *capture*. Pierwszy kanał reaguje na zbocze rosnącego sygnału PWM, kapturuje stan licznika i zeruje go. Licznik zaczyna więc zliczać czas od zbocza rosnącego. Drugi kanał reaguje na zbocze opadające - kapturuje stan licznika. Tym sposobem mamy zmierzony czas trwania stanu wysokiego, co można przeliczyć na współczynnik wypełnienia. Licznik liczy dalej, aż do kolejnego zbocza rosnącego. Wtedy pierwszy kanał znowu zapisuje stan licznika (czyli okres sygnału) i zeruje go. Ta dam. Czujesz siłę i moc? Właśnie zmierzliśmy częstotliwość i współczynnik wypełnienia sygnału PWM w czysto sprzętowy sposób! Zacne, nieprawdaż?

Jak to skonfigurować? W RM jest przepis jak to włączyć, więc nie powinno być problemu. Generalnie plan jest taki:

- PWM dochodzi do wejścia CH1 (TI1)
- przechodzi przez detektor zboczy
- IC1 ma wyzwać pierwszy kanał capture przy zboczu rosnącym
- IC2 ma wyzwać drugi kanał capture przy zboczu opadającym
- TI1FP1 dodatkowo ma resetować licznik

Zadanie domowe 8.13: no wiadomo – uruchomić tryb *PWM Input Mode*.

Przykładowe rozwiązanie (F103, wyjście PWM na PA0, wejście PWM na PA8):

```
1. int main(void) {
2.
3.     RCC->APB2ENR = RCC_APB2ENR_IOPAEN | RCC_APB2ENR_TIM1EN;
4.     RCC->APB1ENR = RCC_APB1ENR_TIM2EN;
5.
6.     /* Testowy PWM */
7.     gpio_pin_cfg(GPIOA, PA0, gpio_mode_alternate_PP_2MHz);
8.
9.     TIM2->CCMR1 = TIM_CCMR1_OC1PE | TIM_CCMR1_OC1M_2 | TIM_CCMR1_OC1M_1;
10.    TIM2->CCER = TIM_CCER_CC1E;
11.    TIM2->PSC = 100;
12.    TIM2->ARR = 100;
13.    TIM2->CCR1 = 37;
14.    TIM2->EGR = TIM_EGR_UG;
15.    TIM2->CR1 = TIM_CR1_CEN;
16.
17.    /* PWM Input Mode */
18.    gpio_pin_cfg(GPIOA, PA8, gpio_mode_input_floating);
19.
20.    TIM1->ARR = UINT16_MAX;
21.    TIM1->PSC = 50;
22.    TIM1->CCMR1 = TIM_CCMR1_CC1S_0 | TIM_CCMR1_CC2S_1;
23.    TIM1->CCER = TIM_CCER_CC2P | TIM_CCER_CC1E | TIM_CCER_CC2E;
24.    TIM1->SMCR = TIM_SMCR_TS_2 | TIM_SMCR_TS_0 | TIM_SMCR_SMS_2;
24.    TIM1->CR1 = TIM_CR1_CEN;
26.
27.    while(1);
28. }
```

Początek programu to ustawienie TIM2 tak, aby generował PWM na nóżce PA0. Proszę zwrócić uwagę na to, gdzie włączany jest zegar dla TIM2 (w innym rejestrze niż TIM1). Tym sposobem mamy wygenerowany PWM, który zaraz będziemy mierzyć... symbioza liczników :) Sygnał potem doprowadzony jest do PA8. Ja już... mam „wypływ proteinowy” na myśl o licznikach, więc analizę kodu pozostawiam Tobie. Nic nowego tu nie ma.

Za pomocą debuggera odczytałem wyniki pomiaru:

- $\text{TIM1}-\rightarrow\text{CCR1} = 0xC7$
- $\text{TIM1}-\rightarrow\text{CCR2} = 0x49$

Policzmy czy się zgadza:

- częstotliwość PWM generowanego przez licznik TIM2:

$$f_{\text{PWM}} = \frac{f_{\text{TIM2}}}{(PSC_{\text{TIM2}} + 1) \cdot (ARR_{\text{TIM2}} + 1)} = \frac{8e6}{101^2} \approx 784,24 \text{ Hz}$$

- współczynnik wypełnienia generowanego PWMa:

$$d_{\text{PWM}} = \frac{\text{CCR1}_{\text{TIM2}}}{ARR_{\text{TIM2}} + 1} \cdot 100 = \frac{37}{1,01} \approx 36,63 \text{ %}$$

- zmierzona częstotliwość sygnału PWM:

$$f_x = \frac{f_{\text{TIM1}}}{(PSC_{\text{TIM1}} + 1) \cdot (\text{CCR1}_{\text{TIM1}} + 1)} = \frac{8e6}{51 \cdot 200} \approx 784,31 \text{ Hz}$$

- zmierzony współczynnik wypełnienia PWM:

$$d_x = \frac{\text{CCR2}_{\text{TIM1}}}{\text{CCR1}_{\text{TIM1}}} \cdot 100 = \frac{73}{199} \cdot 100 \approx 36,68 \text{ %}$$

Jak dla mnie działa.

Co warto zapamiętać z tego rozdziału?

- *PWM input mode*, choć ST zrobiło z tego osobny tryb, nie jest niczym innym jak zgrabnym połączeniem synchronizacji i przechwytywania
- ten tryb pozwala na sprzętowy pomiar okresu i współczynnika wypełnienia zewnętrznego sygnału PWM

8.16. Synchronizacja kilku liczników

Pojęcie synchronizacji i łączenia liczników pojawiało się już tyle razy, że chyba mimochodem każdy czuje o co chodzi. Synchronizacja liczników polega na tym, że licznik nadzorowany (*master... of counters*), steruje licznikiem podrzędnym (*slave*). *Master* wysyła sygnał TRGO, kontroler *slave'a* odbiera go u siebie jako sygnał ITR. Kiedy *master* może wysyłać TRGO:

- przy UEV (wysyła impuls)
- gdy jest włączony (ustawiony bit TIM_CR1_CEN) i zlicza (wysyła sygnał ciągły)
- bloki porównujące (sygnał OCxREF) mogą generować TRGO (impuls w chwili zrównania lub sygnał ciągły zależny od wyniku porównania)
- po resetie wymuszonym przez bit TIM_EGR_UG (impuls)

Z kontrolera *slave* już korzystaliśmy w poprzednich przykładach. Przypomnę tylko, że możliwe są następujące tryby jego pracy:

- praca w trybie enkodera - to nas nie interesuje aktualnie
- impuls z *mastera* może wyzwalać licznik podrzędnego
- sygnał ciągły z *mastera* może bramkować taktowanie licznika podrzędnego
- impuls z *mastera* może resetować licznik podrzędnego
- impuls z *mastera* może taktować licznik podrzędnego

Żeby było śmieszniej łączyć można więcej niż dwa liczniki. Mogą być np. trzy połączone¹³⁰: TIM1 będzie masterem dla TIM2, zaś TIM2 masterem dla TIM3... Najbardziej spektakularnie na analizatorze będzie prezentował się tryb z bramkowaniem licznika podrzędnego, toteż i taki sobie zaserwujemy. Plan jest następujący:

- *masterem* będzie TIM1, będzie generował PWM z wykorzystaniem bloku porównującego
- PWM z TIM1 (właściwie sygnał OCxREF) będzie wysyłany jako TRGO (nie będzie wyprowadzony na nóżkę mikrokontrolera!)
- TIM2 będzie *slave'em*
- *slave* będzie bramkowany sygnałem z mastera
- *slave* ma generować PWM na nóżce mikrokontrolera, żeby było co złapać analizatorem

130 przykład poglądowy - nie gwarantuję, że akurat taka konfiguracja jest możliwa

Konfiguracja *mastera* to banał: wystarczy skopiować przykład z PWM, wyrzucić z niego to co odpowiada za wyprowadzenie sygnału na nóżkę mikrokontrolera i dorzuć ustawienie go w trybie *master*. Konfiguracja *slave'a* jest jeszcze prostsza: wystarczy skopiować przykład z PWM i dorzucić do niego konfigurację w trybie *slave*.

Zadanie domowe 8.14: TIM1 w trybie *master* bramkuje TIM2, który generuje PWM na nóżce mikrokontrolera.

Przykładowe rozwiązanie (F103, wyjście PWM na PA0):

```

1. int main(void) {
2.
3.     RCC->APB2ENR = RCC_APB2ENR_IOPAEN | RCC_APB2ENR_TIM1EN;
4.     RCC->APB1ENR = RCC_APB1ENR_TIM2EN;
5.
6.     /* Master */
7.     TIM1->CR2 = TIM_CR2_MMS_2;
8.     TIM1->CCMR1 = TIM_CCMR1_OC1M_2 | TIM_CCMR1_OC1M_1;
9.     TIM1->PSC = 2;
10.    TIM1->ARR = 500;
11.    TIM1->CCR1 = 200;
12.    TIM1->CR1 = TIM_CR1_CEN;
13.
14.     /* Slave */
15.     gpio_pin_cfg(GPIOA, PA0, gpio_mode_alternate_PP_2MHz);
16.
17.     TIM2->CCMR1 = TIM_CCMR1_OC1M_2 | TIM_CCMR1_OC1M_1;
18.     TIM2->CCER = TIM_CCER_CC1E;
19.     TIM2->ARR = 100;
20.     TIM2->CCR1 = 50;
21.     TIM2->SMCR = TIM_SMCR_SMS_2 | TIM_SMCR_SMS_0;
22.     TIM2->CR1 = TIM_CR1_CEN;
23.
24.     while(1);
25. }
```

I widok z analizatora:



Rys. 8.11. TIM2 generuje PWM i jest bramkowany z TIM1

Wydaje mi się, że jedyna rzecz jaka może tu budzić wątpliwości to wybór sygnału ITR w konfiguracji *slave'a*. Na schemacie ogólnym licznika widać, że do kontrolera *slave* dochodzi kilka sygnałów ITRx. To o który sygnał nam chodzi zależy od tego jakie liczniki łączymy. Zerknij proszę, do opisu bitów TIM_SMCR_SMS w rozdziale o liczniku TIM2. Jest tam taka tabela:

Tabela 8.1 Źródła sygnałów ITR liczników TIM2 - TIM5

Slave TIM	ITR0 (TS = 000)	ITR1 (TS = 001)	ITR2 (TS = 010)	ITR3 (TS = 011)
TIM2	<i>TIM1</i>	<i>TIM8</i>	<i>TIM3</i>	<i>TIM4</i>
TIM3	<i>TIM1</i>	<i>TIM2</i>	<i>TIM5</i>	<i>TIM4</i>
TIM4	<i>TIM1</i>	<i>TIM2</i>	<i>TIM3</i>	<i>TIM8</i>
TIM5	<i>TIM2</i>	<i>TIM3</i>	<i>TIM4</i>	<i>TIM8</i>

W pierwszej kolumnie ujęte są liczniki podrzędne. W kolejnych kolumnach wymienione są liczniki, które mogą być nadzorowanymi dla *slave'a* z danego wiersza pierwszej kolumny. W pierwszym wierszu podane są wartości bitów TIM_SMCR_TS odpowiadające danej konfiguracji. Np. licznikami nadzorowanymi dla TIM4 mogą być:

- *TIM1*, poprzez sygnał ITR0, konfiguracja bitów TS = 0b000
- *TIM2*, poprzez sygnał ITR1, konfiguracja bitów TS = 0b001
- *TIM3*, poprzez sygnał ITR2, konfiguracja bitów TS = 0b010
- *TIM8*, poprzez sygnał ITR3, konfiguracja bitów TS = 0b011

Trywialne? Jeśli jakiegoś licznika nie ma w tabeli to znaczy, że taka konfiguracja nie jest możliwa. Np. TIM5 nie może być *masterem* dla TIM4.

Co warto zapamiętać z tego rozdziału?

- synchronizacja liczników polega na tym, że jeden licznik steruje pracą drugiego
- licznik nadzorowany wysyła sygnał TRGO
- licznik podrzędny odbiera ten sygnał jako ITR
- master może wysyłać TRGO: w wyniku działania bloku porównującego, przy UEV, po ustawieniu bitu UG lub CEN
- kontroler licznika podrzecznego może: resetować, wyczewalać, bramkować, taktować licznik

8.17. Liczniki ogólnego przeznaczenia i podstawowe

Wszystko co zostało do tej pory powiedziane, dotyczyło jednej grupy liczników: *advanced*. Pozostały jeszcze dwie grupy: *general purpose* oraz *basic*. Dobra wiadomość jest taka, że od *advance'ów* różnią się tylko tym, że nie mają określonych funkcji. Przeto mamy je już właściwie opanowane :) Porównanie najważniejszych ficzerów liczników z różnych grup w formie macierzy:

Tabela 8.2 Porównanie typów liczników

Licznik (numery)	Kierunek zliczania	Kanały CC ¹³¹	Uwagi	Źródła przerwań i żądań DMA
advanced 1, 8	góra, dół, symetrycznie	4	- wyjścia komplementarne - generator czasu martwego - funkcja break - licznik powtórzeń - interfejs enkodera	UEV, trigger, capture, compare, break
general 2 - 5	góra, dół, symetrycznie	4	- interfejs enkodera	UEV, trigger, capture, compare
general 9, 12	tylko w górę	2	-	(tylko przerwania) UEV, trigger, capture, compare
general 10, 11, 13, 14	tylko w górę	1	- brak możliwości synchronizacji z innymi licznikami	(tylko przerwania) UEV, capture, compare
basic 6, 7	tylko w górę	0	- dedykowany do wyzwalania przetwornika DAC	UEV

Przypominam, że nie każdy konkretny model mikrokontrolera ma wszystkie wymienione liczniki. Szczegóły należy sobie odszukać w datasheetie kostki.

Co warto zapamiętać z tego rozdziału?

- skoro poznaliśmy liczniki zaawansowane to pozostałe już nam nie podskoczą
- liczniki zaawansowane mają dodatkowe funkcje związane z sterowaniem energoelektroniką (wyjścia komplementarne, generatory czasów martwych, funkcje break)
- liczniki podstawowe (*basic*) są najprostsze i są dedykowane do wyzwalania przetworników ADC i DAC (w sposób sprzętowy rzecz jasna)

8.18. Luźne uwagi na koniec

Rozdział, choć długi, nie wyczerpuje tematu liczników. Pominąłem kilka trybów i funkcji, które IMHO wykazują mniejszą przydatność¹³². Zainteresowanych jak zawsze odsyłam do dokumentacji. A co konkretnie pominąłem (najważniejsze punkty):

- licznik powtórzeń RCR
- 6-step PWM generation
- wejściową bramkę XOR na wejściu kanałów 1, 2 i 3 oraz *Hall Sensor Interfacing*

131 Capture / Compare

132 albo ich po prostu nie pojąłem :)

- kwestie związane z *DMA Burst Mode*
- po macoszemu potraktowałem kwestię generowania przerwań (i żądań DMA), w przykładach pojawiło się przerwanie od UEV i *compare* ale możliwości jest sporo więcej

Domyślam się, że po przeczytaniu tego rozdziału masz niemiłosierny pierdolnik w głowie. Jak to wszystko ogarnąć? Wydaje mi się, że najważniejsze jest to, żeby z grubsza wiedzieć jakie opcje są dostępne i mając konkretne zadanie do zrealizowania, spróbować się jakoś wpasować w możliwości liczników. Warto poświęcić sporo czasu na takie rozplanowanie wykorzystania liczników (i ogólnie peryferiów) aby jak najwięcej funkcji było realizowanych sprzętowo. Kosztem czasu spędzonego nad dokumentacją i konfiguracją, zyskujemy znaczne odciążenie części programowej.

W AVRach to było jakoś tak (przynajmniej ja tak to odbierałem), że licznik należało skonfigurować w jakimś konkretnym trybie i koniec zabawy. Przy STMach trzeba się przestawić. Różne tryby i funkcje nie działają odrębnie. Można je łączyć i mieszać w ramach jednego licznika. Np. licznik może być taktowany enkoderem obrotowym, mieć włączone dwa bloki porównawcze generujące przerwania a do tego włączoną funkcję kapturowania. I na dokładkę niech steruje jeszcze innym licznikiem podrzędnym. Nie ma się co bać eksperymentów – najwyżej nie zadziała i tyle :)

A ważna sprawa! Przy korzystaniu z debuggera mamy możliwość zatrzymania pracy rdzenia. Pojawia się pytanie - a co z licznikami? Mamy tu pewne, małe, pole do popisu. Domyślnie liczniki nie są zatrzymywane. To znaczy, że po zatrzymaniu rdzenia licznik jest taktowany i np. generacja PWM czy przerwań działa w najlepsze. Przerwania oczywiście nie zostaną obsłużone bo rdzeń jest zatrzymany, ale flagi się ustawią i będą czekały na odblokowanie rdzenia. Nie zawsze nam to pasuje - np. jeśli sterujemy jakimś czymś i musimy zatrzymać sterowany obiekt przy haltowaniu rdzenia... a może właśnie nie możemy sobie pozwolić na to by nagle zniknął sygnał PWM... W sukurs przychodzi nam wtedy rejestr DBGMCU_CR i bity DBG_TIMx_STOP. Za ich pomocą możemy skonfigurować zachowanie licznika po zatrzymaniu rdzenia. Tak trochę na wyrost ale od razu wspomnę o tym, że w tym rejestrze możemy również ustawić jak mają się zachowywać *watchdogi* (w końcu też liczniki) po zatrzymaniu rdzenia.

Literatura dodatkowa do wygooglania we własnym zakresie:

- AN2581: *STM32F10xxx TIM application examples*
- AN4013: *STM32F0, STM32F1, STM32F2, STM32F4, STM32L1 series, STM32F30x, STM32F3x8, STM32F373 lines timer overview*

- AN2580: *STM32F10xxx TIM1 application examples*
- AN2592: *How to achieve 32-bit timer resolution using the link system in STM32F10x and STM32L15x microcontrollers*

Noty pochodzą oczywiście ze strony ST, przy czym oficjalna wyszukiwarka strony jakoś ich nie znajduje i trzeba wspomóc się google'ami. Szczególnie polecam AN4013. AN2580/1 opisuje przykładowe kody z biblioteki SPL.

Co warto zapamiętać z tego rozdziału?

- kilka ciekawych funkcji nie zostało opisanych w tym rozdziale!
- nie można się tego uczyć na pamięć...

8.19. Różnice między F103 i F429 (F103 i F429)

Liczniki w F103 i F429 są praktycznie identyczne... albo nawet i całkiem identyczne - nie chce mi się porównywać RMów literka po literce, więc mogłem coś przegapić. Na pewno w F429 inaczej konfiguruje się nóżki do współpracy z układami peryferyjnymi i w innych rejestrach będzie się włączać taktowanie liczników.

Rozpracujmy więc te „różnice”. Zadanie na początek jest proste: zidentyfikować nóżki odpowiadające kanałom 1, 2, 3 licznika TIM1 i obczać sposób ich konfiguracji. Kto uważnie czyta Poradnik od początku, ten może pamięta że to już było (rozdział 3.5) :) Tak czy owak polecam poćwiczyć.

Zadanie domowe 8.15: przerobić program do generowania sygnału PWM z zadania 8.8 tak, aby działał na STM32F429.

Przykładowe rozwiązanie (F429, PWM na PE9, PA9, PA10):

```
1. int main(void) {
2.
3.     RCC->AHB1ENR |= RCC_AHB1ENR_GPIOAEN | RCC_AHB1ENR_GPIOEEN;
4.     RCC->APB2ENR = RCC_APB2ENR_TIM1EN;
5.     __DSB();
6.
7.     gpio_pin_cfg(GPIOE, PE9, gpio_mode_AF1_OD_PU_LS);
8.     gpio_pin_cfg(GPIOA, PA9, gpio_mode_AF1_OD_PU_LS);
9.     gpio_pin_cfg(GPIOA, PA10, gpio_mode_AF1_OD_PU_LS);
10.
11.
12.    TIM1->CCMR1 = TIM_CCMR1_OC1PE | TIM_CCMR1_OC1M_2 | TIM_CCMR1_OC1M_1;
13.    TIM1->CCMR1 |= TIM_CCMR1_OC2PE | TIM_CCMR1_OC2M_2 | TIM_CCMR1_OC2M_1;
14.    TIM1->CCMR2 = TIM_CCMR2_OC3PE | TIM_CCMR2_OC3M_2 | TIM_CCMR2_OC3M_1;
15.
16.    TIM1->CCER = TIM_CCER_CC1E | TIM_CCER_CC2E | TIM_CCER_CC3E;
17.    TIM1->BDTR = TIM_BDTR_MOE;
18.
19.    TIM1->PSC = 7;
20.    TIM1->ARR = 99;
21.    TIM1->CCR1 = 50;
22.    TIM1->CCR2 = 80;
23.    TIM1->CCR3 = 20;
24.
25.    TIM1->EGR = TIM_EGR_UG;
26.    TIM1->CR1 = TIM_CR1_ARPE | TIM_CR1_CEN;
27.
28.    while (1);
29.
30. } /* main */
```

Jest tu coś ciekawego? Hmm:

- 3) zastosowano sumę bitową a nie przypisanie, gdyż domyślna wartość rejestru nie jest równa zero
- 5) instrukcja „barierowa” ze względu na mały bug w procku (jak ktoś nie pamięta: [klik](#))
- 7, 8, 9) konfiguracja pinów. Wybrałem tryb *open-drain* z *pull-upem* bez jakiś głębszych powodów...
bo tak

Cała reszta kodu jest bez zmian.

Zadanie domowe 8.16: przerobić przykład z licznikiem taktowanym sygnałem zewnętrznym (z zadania 8.4) tak, aby działał w STM32F429.

Przykładowe rozwiązanie (F429, wyjście PWM na PA5, wejście sygnału na PA9):

```
1. int main(void) {
2.
3.     RCC->AHB1ENR |= RCC_AHB1ENR_GPIOAEN;
4.     RCC->APB2ENR = RCC_APB2ENR_TIM1EN;
5.     __DSB();
6.
7.     gpio_pin_cfg(GPIOA, PA5, gpio_mode_out_PP_LS);
8.     gpio_pin_cfg(GPIOA, PA9, gpio_mode_AF1_PP_PD_LS);
9.
10.    TIM1->CCER = TIM_CCER_CC1P;
11.    TIM1->SMCR = TIM_SMCR_SMS | TIM_SMCR_TS_1 | TIM_SMCR_TS_2;
12.
13.    TIM1->ARR = 10;
14.    TIM1->DIER = TIM_DIER_UIE;
15.    TIM1->CR1 = TIM_CR1_CEN;
16.
17.    NVIC_EnableIRQ(TIM1_UP_TIM10_IRQn);
18.
19.    SysTick_Config(800000);
20.
21.    while (1);
22.
23. } /* main */
24.
25.
26. __attribute__((interrupt)) void TIM1_UP_TIM10_IRQHandler(void){
27.     if (TIM1->SR & TIM_SR UIF){
28.         TIM1->SR = ~TIM_SR UIF;
29.         BB(GPIOA->ODR, PA5) ^= 1;
30.     }
31. }
32.
33. __attribute__((interrupt)) void SysTick_Handler(void){
34.     BB(GPIOA->PUPDR, GPIO_PUPDR_PUPDR9_0) ^= 1;
35.     BB(GPIOA->PUPDR, GPIO_PUPDR_PUPDR9_1) ^= 1;
36. }
```

Nowości:

8) w F103 nóżki działające jako wejścia układów peryferyjnych konfigurowaliśmy w trybie wejściowym, w F429 natomiast używana jest konfiguracja alternatywna! Trochę to mylące bo konfigurujemy jako *push-pull* itd., ale peryferial sam sobie ustawi że to ma być wejście. Dla bezpieczeństwa można by przenieść konfigurację nóżki „za” konfigurację peryferiala - tak na wszelki wypadek - nie ufam temu wybieraniu kierunku przez peryferial...

17) uwaga! inna nazwa przerwania niż w F103

26) wektor też się nazywa inaczej niż w F103! Z rozpetto zapomniałem wyrzucić atrybuty przy ISRach... nie są potrzebne, ale i w niczym nie przeszkadzają.

34, 35) machanie pinem poprzez zmianę podciągania (góra / dół)

Co warto zapamiętać z tego rozdziału?

- a trzeba tu coś zapamiętywać?

8.20. Licznikowe indywidualne (F334)

No nie mogli zostawić liczników bez zmian, musieli namieszać w F334. Spróbujemy to jakoś ogarnąć sensownie. Pozostaniemy przy sprawdzonej konwencji, tzn. przerobimy sobie najbardziej zaawansowany licznik, a potem sprawdzimy czego pozostałe liczniki nie potrafią. Najpierw będzie kilka mniej znaczących i nudnych zmian, te ciekawsze i bardziej widowiskowe zostawimy na deser.

Mikrokontroler F334 posiada jeden licznik z grupy *advanced* (TIM1). Fajną nowością, w tym liczniku, jest obecność aż sześciu kanałów CC. Przy czym dwa nowe kanały (5 i 6) mogą pracować tylko jako *compare* i nie mają obwodów wyjściowych. To znaczy, że nie mogą generować sygnałów wyjściowych (np. pwm na nóżce mikrokontrolera). Generują jedynie wewnętrzny sygnał referencyjny wykorzystywany w bloku licznika (np. do generowania przerwań). Oczywistą oczywistością jest, że w związku z większą liczbą kanałów, zwiększyła się również liczba bitów konfiguracyjnych. Warto zwrócić szczególną uwagę na położenie bitów w rejestrach. W kilku przypadkach, wielo-bitowe pola zostały rozbite tak, że poszczególne bity tych pól nie leżą obok siebie. Np. w rejestrze TIM_CCMR1 jest cztero-bitowe pole OC1M - trzy bity tego pola zajmują pozycje 4, 5, 6 (w rejestrze), zaś ostatni bit leży kawałek dalej (bit numer 16 w rejestrze). Jakby kogoś interesowała przyczyna - chodzi o zachowanie kompatybilności z innymi wersjami układów licznika (np. z licznikami z F103/F429).

Nowość numer dwa, to dodatkowy tryb pracy w konfiguracji licznika podrzędnego (patrz pole bitowe TIM_SMCR_SMS). Nowy tryb to *combined reset + trigger*. Jak sama nazwa wskazuje, w tym trybie trygierz powoduje wyzerowanie licznika i jego wyzwolenie (zlicza od nowa). Ta konfiguracja jest predysponowana do współpracy z trybami generowania pojedynczych impulsów. Zresztą zaraz ją wykorzystamy.

Zmiana numer trzy dotyczy funkcji *break*. W F334 dostępne są dwa „sygnały” break, każdy powiązany z szeregiem możliwych źródeł. I tak, pierwszy sygnał *brk* może pochodzić od:

- źródeł „zewnętrznych”:
 - pin *bkin*
 - wyjście komparatora COMP4
- źródeł „wewnętrznych” (tak je określa dokumentacja, nie koniecznie są „wewnętrzne” w fizycznym znaczeniu):
 - układ kontroli sygnału zegarowego (CSS, więcej w rozdziale 17)
 - programowalny układ kontroli napięcia (PVD, więcej w rozdziale 11.4)

- układ kontroli pamięci SRAM (*SRAM parity check*)¹³³
- wyjścia komparatorów COMP1, 2, 3, 5, 6
- rdzenia (sygnał pojawia się w przypadku wystąpienia błędu *Hard Fault*)

Drugi sygnał break (twórczo i ambitnie nazwany *brk2*), jest związany z następującymi źródłami:

- pin *bkin2*
- wyjścia wszystkich komparatorów (COMP1...7)

Ponadto sygnały mogą być wygenerowane w sposób programowy. Działanie obu sygnałów break jest podobne. Drobna różnica polega na tym, że *brk* potrafi wyłączać generowanie sygnałów wyjściowych oraz może wymusić ustalone, bezpieczne stany na wyjściach mikrokontrolera. Drugi sygnał (*brk2*), może jedynie wyłączyć generowanie.

Pojawił się drugi sygnał wyjściowy licznika pracującego jako *master* (TRGO2). Sygnał TRGO2 jest dedykowany do wyzwalania przetworników ADC. Jego konfiguracja jest możliwa za pomocą bitów TIM_CR2_MMS2. Tyle.

Następna zmiana z gatunku małych i nudnawych. Flaga przepełnienia licznika (UIF) może być dostępna w rejestrze licznika (CNT). Działa to tak, że po włączeniu bitu TIM_CR1_UIFREMAP, flaga jest automatycznie i sprzętowo kopiowana do rejestru CNT (bit na pozycji 31). Dzięki temu możemy za jednym zamachem odczytać wartość rejestru licznika i flagę. Zapewnia to spójność danych (jeżeli odczytalibyśmy osobno np. flagę a potem CNT, to istniałoby ryzyko, że licznik przekręci się **po** odczycie flagi). Na pewno można znaleźć bardzo dużo, ciekawych zastosowań tej funkcji...

No i wreszcie duża zmiana, warta grzechu :) Większa liczba trybów pracy kanałów wyjściowych. Nowe tryby pracy to (niech to dunder świśnie, a ja się gubiłem już przy dwóch trybach pwm):

- *retriggerable one pulse mode 1*
- *retriggerable one pulse mode 2*
- *asymmetric pwm mode 1*
- *asymmetric pwm mode 2*
- *combined pwm mode 1*

¹³³ to jakaś sprzętowa funkcja umożliwiająca weryfikację poprawności danych w ramie, czy coś w tym guście

- *combined pwm mode 2*
- *combined 3-phase pwm mode 1*
- *combined 3-phase pwm mode 2*

Dwa pierwsze tryby polegają na tym, że licznik generuje impuls o określonej długości, w odpowiedzi na sygnał (trygierz) z zewnątrz (sygnał TRGI). Dodatkowo, jeżeli nowy trygierz pojawi się przed końcem poprzedniego impulsu, to czas trwania impulsu będzie się liczył od nowa. Nie pozostaje nam nic innego jak pobawić się nowym trybem.

Zadanie domowe 8.17: uruchomić „retrygierzalny jednoprzelotowy”. Niech wyzwalaczem będzie np. przycisk na płytce Nucleo. W odpowiedzi na trygierz, niech licznik generuje impuls o długości ~500ms.

Przykładowe rozwiązanie (F334, połączone przewodem wyprowadzenia PA5 i PA9):

```

1. int main(void){
2.
3.     RCC->AHBENR = RCC_AHBENR_GPIOAEN | RCC_AHBENR_GPIOCEN | RCC_AHBENR_GPIOFEN;
4.     RCC->APB2ENR = RCC_APB2ENR_TIM1EN;
5.
6.     gpio_pin_cfg(GPIOA, PA5, gpio_mode_out_PP_LS);
7.     gpio_pin_cfg(GPIOA, PA9, gpio_mode_AF6_OD_PD_LS);
8.     gpio_pin_cfg(GPIOF, PF1, gpio_mode_out_PP_LS);
9.     gpio_pin_cfg(GPIOC, PC13, gpio_mode_in_floating);
10.    gpio_pin_cfg(GPIOC, PC0, gpio_mode_AF2_PP_HS);
11.
12.    TIM1->PSC = 8000-1;
13.    TIM1->ARR = 500-1;
14.
15.    TIM1->CCMR1 = TIM_CCMR1_OC1M_3;
16.    TIM1->SMCR = TIM_SMCR_SMS_3 | TIM_SMCR_TS_2 | TIM_SMCR_TS_1;
17.    TIM1->CCER;
18.    TIM1->BDTR = TIM_BDTR_MOE;
19.    TIM1->CR1 = TIM_CR1_OPM;
20.
21.    SysTick_Config(8000000/2);
22.
23.    while(1) GPIOA->BSRR = (GPIOC->IDR & PC13 ? GPIO_BSRR_BS_5 : GPIO_BSRR_BR_5);
24.
25. }
26.
27. void SysTick_Handler(void){
28.     GPIOF->ODR ^= PF1;
29. }
```

No i tak to wygląda. W przerwaniu SysTicka machana jest dioda podłączona do PF1 (patrz dodatek 7). Tryb ROPM (*retriggerable one pulse mode*) musi być wyzwalany sygnałem zewnętrzny (TRGI). Trochę kombinowałem, ale nie udało mi się zmusić tej konfiguracji do wyzwolenia programowego. Trudno. W programie licznik wyzwalany jest wejściem kanału drugiego. To jest szósta funkcja alternatywna wyprowadzenia PA9.

Dla wygody testowania, zrobiłem taki myk, że połączylem wyprowadzenia PA5 i PA9. Naciśnięcie przycisku na płytce *nucleo* (PC13) powoduje zmianę stanu wyjścia PA5. Do tego pinu w zestawie podłączona jest leda świecąca, więc mamy ładne miganie, a ponieważ PA5 jest połączone z PA9 (przewodem) to mamy również wyzwalanie licznika. Sygnał wyjściowy (impuls) generowany jest na kanale pierwszym licznika (PC0).

Długość impulsu wyznacza wartość rejestru preskalera (linia 12) i rejestru przeładowania (linia 13). Mikrokontroler działa z domyślną częstotliwością równą 8MHz. Po podzieleniu przez preskaler ($PSC + 1$) licznik zlicza z częstotliwością 1kHz. Czyli jeden okres sygnału zegarowego trwa 1ms. Pół sekundy to będzie 500 tyknięć zegara. Zliczanie następuje od zera, stąd „-1” w linii 13.

15) wybór konfiguracji dla kanału pierwszego licznika, kanał pracuje w konfiguracji wyjściowej (tryb *retriggerable one pulse mode I*). Dokumentacja zaleca, aby wartość rejestru porównawczego (CCR_x) ustawić na zero. Jest to domyślna wartość po resetie, więc nic nie zmieniam.

16) licznik należy skonfigurować dodatkowo w trybie *slave - combined reset + trigger*. Tak przynajmniej zaleca RM. Odpowiada za to konfiguracja pola TIM_SMCR_SMS. Uwaga na „nieciągłość” pola w rejestrze! Bit SMS[3] nie leży obok pozostałych bitów tego pola. Bity TS odpowiadają za wybór sygnału wyzwalającego licznik. Wybieramy taki układ, aby wyzwalanie (TRGI) pochodziło od drugiego kanału (TI2FP2), polecam popatrzeć na schemat blokowy licznika :)

17) włączenie pierwszego kanału capture/compare. Drugiego kanału nie musimy włączać, bo nie korzystamy z jego funkcji capture lub compare.

18) odblokowanie możliwości generowania sygnałów (jak nie pamiętasz to cofnij się do opisu funkcji break, o tu [funkcja break](#))

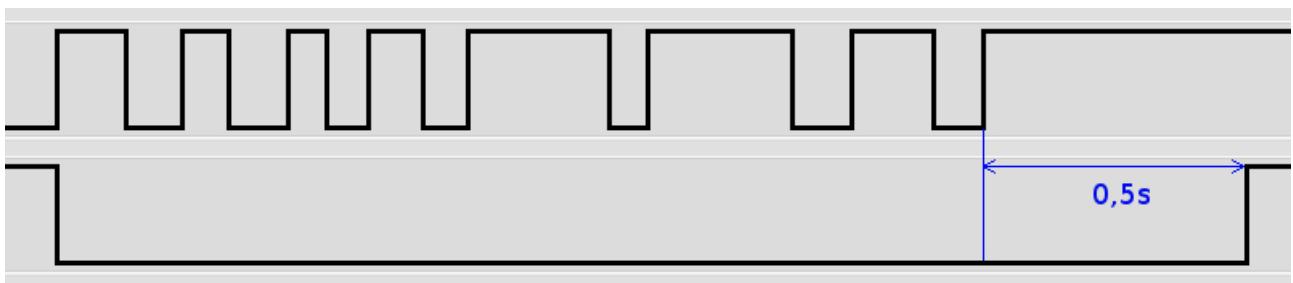
19) wiadomo

A efekt naszych poczynań przedstawia się tak jak na rysunkach 8.12 i 8.13. Górnny przebieg na rysunku 8.12 to PA5 (i PA9). Licznik reaguje na rosnące zbocze sygnału wyzwalającego (można to zmienić w konfiguracji kanału 1 licznika). Po wykryciu zbocza, niezwłocznie, generowany jest „ujemny” impuls. Czas trwania impulsu wynosi ~500ms. Wszystko działa tak jak miało działać.

Na kolejnym rysunku jest ta sama sytuacja, ale w czasie trwania impulsu pojawiają się kolejne sygnały wyzwalające. Zgodnie z zasadą działania trybu *ROPM*, każdy kolejny trygierz powinien resetować czas trwania impulsu. Innymi słowy impuls ma trwać te 500ms od ostatniego trygierza. Sprawdzone - działa.

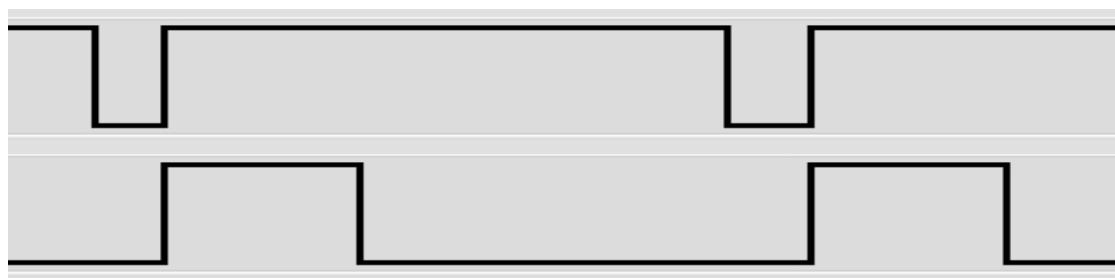


Rys. 8.12 Tryb *retriggerable one pulse mode 1* (góra: sygnał wyzwalający, dół: wygenerowany impuls)

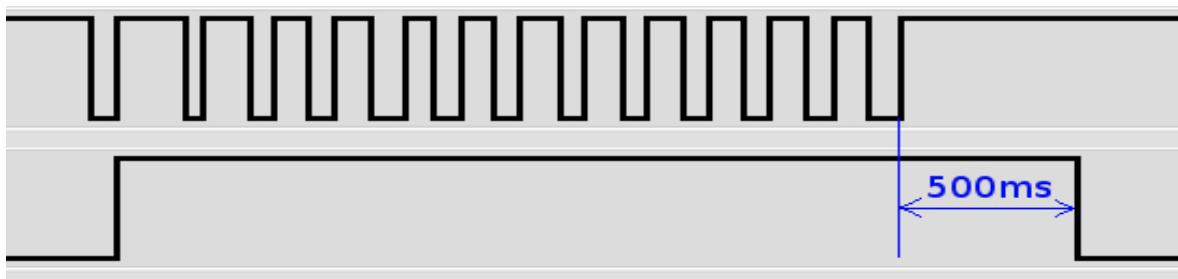


Rys. 8.13 Tryb *retriggerable one pulse mode 1* (góra: sygnał wyzwalający, dół: wygenerowany impuls)

Z pewnością, tak samo jak i ja, nie możesz doczekać się, aby poznać odpowiedź na pytanie: czym różni się *ROP¹³⁴ mode 1* od *ROP mode 2*. No więc powyższy program, po zmianie trybu na „2” działa tak samo, jeno impuls się odbił w pionie - patrz rysunki 8.14 i 8.15. Teraz impuls jest dodatni. I znowu, w swojej ignorancji, nie widzę sensu istnienia tych dwóch „mode’ów”. Nie wgryzałem się dokładnie w opis w RMie, ale odbicie sygnału mogę uzyskać za pomocą bitów TIM_CCER_CCxP (zmiana polaryzacji wyjścia). Sprawdziłem - po odwróceniu polaryzacji wyjścia w trybie *mode 1*, przebieg wygląda identycznie jak w *mode 2*. Także ten...

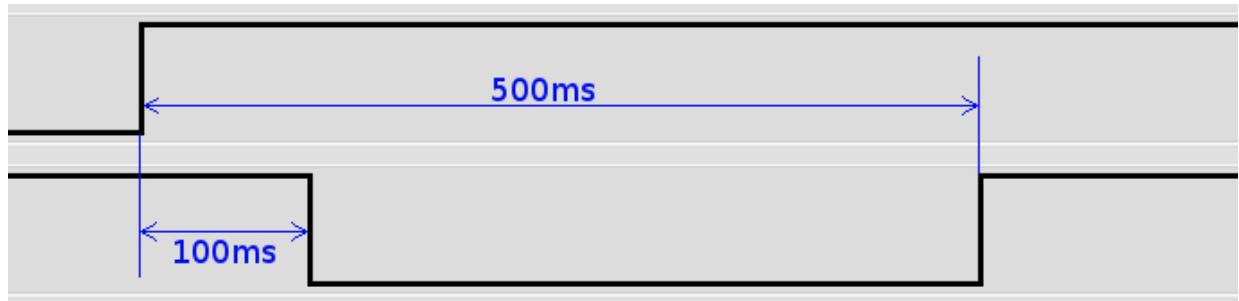


Rys. 8.14 Tryb *retriggerable one pulse mode 2* (góra: sygnał wyzwalający, dół: wygenerowany impuls)



Rys. 8.15 Tryb *retriggerable one pulse mode 2* (góra: sygnał wyzwalający, dół: wygenerowany impuls)

Oczywiście najciekawsza zabawa zaczyna się wtedy, kiedy przestajemy słuchać dobrych rad cioci dokumentacji. No więc spróbujmy nieco *mess around*. Dokumentacja jasno zaleca, aby w trybie *ROPM* ustawić wartość rejestru CCRx na 0 (albo rejestru ARR jeżeli licznik zlicza w dół). Nie byłbym sobą, gdybym nie sprawdził co będzie, gdy CCR zerem nie będzie :) Na rysunku 8.16 pokazany jest efekt działania kodu jak wcześniej, z wartością CCR1 ustawioną na 100. Jak widać, spowodowało to opóźnienie impulsu względem trygierza. No i fajnie. Żeby było śmieszniej, dokładnie to samo można uzyskać bez trybu *retriggerable one pulse mode*. Opóźniony impuls można zrobić w oparciu o zwykły *OPM* i konfigurację *pwm mode*. W ogóle wydaje mi się, że tym co odróżnia tryb *ROPM* od pozostałych jest to, że generowanie impulsu zaczyna się równo z wystąpieniem trygierza. W każdym innym trybie musi być między nimi opóźnienie... albo ja coś przegapiłem.



Rys. 8.16 Tryb *retriggerable one pulse mode 1*; $CCR1 = 100$ (góra: sygnał wyzwalający, dół: wygenerowany impuls)

Co dalej można popsuć? Można nie włączyć bitu OPM. Nic ciekawego to nie powoduje. Bez bitu OPM, licznik nie zatrzymuje się po zakończeniu generowania impulsu - przekręca się i tak w kółko od nowa. Przy czym nie zmienia to w żaden sposób wygenerowanego impulsu! Impuls na wyjściu jest tylko jeden i taki jak powinien być. Po prostu licznik się nie zatrzymuje i kręci „w tle”. Pewnie zwiększa to pobór prądu albo coś.

Dokumentacja zaleca konfigurację licznika w trybie *slave combined reset + trigger*. Czyli czas sprawdzić, jak to zadziała przy innych konfiguracjach. Opcja „reset + trigger” odpowiada za dwie rzeczy: resetowanie licznika przy każdym trygierzu co powoduje wydłużanie impulsu jeśli

pojawi się kilka wyzwoleń oraz za wyzwolenie licznika pracującego w trybie opm. Jeżeli zmienimy konfigurację na samo „*slave trigger mode*” to wszystko będzie działać, ale czas trwania impulsu nie będzie się liczył od ostatniego wyzwolenia tylko od pierwszego - czyli nie będzie wydłużania impulsu, jeśli w czasie jego trwania pojawi się nowy trygier. Jeżeli natomiast wybierzemy tryb „*slave reset mode*” to kontroler licznika nie będzie go wyzwalał (brak „*trigger*”), więc nie możemy włączyć opcji opm. Z kolei bez opm, licznik się nie zatrzyma po wygenerowaniu impulsu (będzie się kręcił w tle), ale poza tym wszystko będzie działać (impuls będzie generowany). Dosyć tej zabawy, jedziemy dalej.

Kolejny punkt programu to dwa tryby ***asymmetric pwm***. Przypomnij sobie jak wyglądał ***center-aligned pwm*** (rysunek 8.8). A teraz wyobraź sobie, że przebiegi są przesunięte względem siebie i każdym zboczem możemy sterować niezależnie... ot i cała filozofia. Działa to stosunkowo prosto. W tym trybie z każdym kanałem związane są dwa rejestrów CCRx. Licznik musi być skonfigurowany z trybie zliczania symetrycznego (w górę od 0 do ARR, potem w dół do zera i zapetlij). Jeden rejestr CCRx wyznacza pierwsze zbocze generowanego przebiegu gdy licznik zlicza w górę. Drugie zbocze zależy od drugiej wartości CCRx i występuje w momencie, gdy licznik liczy w dół. Tym sposobem, za pomocą dwóch rejestrów CCRx (dla jednego kanału), możemy sterować oboma zboczami sygnału niezależnie. Tak na dobrą sprawę, to w tym trybie mogą sensownie działać tylko dwa kanały licznika. To dlatego, że CH1 i CH2 oraz CH3 i CH4 są opisywane przez tą samą parę rejestrów CCRx, więc nie można ich konfigurować w niezależny sposób. Myślę, że tabelka powie więcej niż 1000 słów.

Tabela 8.3. Konfiguracja trybu *asymmetric pwm*

kanał	pierwsze zbocze sygnału	drugie zbocze sygnału
CH1	<i>CCR1</i>	<i>CCR2</i>
CH2	<i>CCR2</i>	<i>CCR1</i>
CH3	<i>CCR3</i>	<i>CCR4</i>
CH4	<i>CCR4</i>	<i>CCR3</i>

Zadanie domowe 8.18: uruchomić, przetestować, pobawić się, zrozumieć, opanować.

Przykładowe rozwiązanie (F334):

```
1. int main(void){  
2.  
3.     RCC->AHBENR = RCC_AHBENR_GPIOAEN | RCC_AHBENR_GPIOCEN | RCC_AHBENR_GPIOFEN;  
4.     RCC->APB2ENR = RCC_APB2ENR_TIM1EN;  
5.  
6.     gpio_pin_cfg(GPIOA, PA5, gpio_mode_out_PP_LS);  
7.     gpio_pin_cfg(GPIOC, PC10, gpio_mode_out_PP_LS);  
8.     gpio_pin_cfg(GPIOC, PC0, gpio_mode_AF2_PP_LS); //ch1  
9.     gpio_pin_cfg(GPIOC, PC2, gpio_mode_AF2_PP_LS); //ch3  
10.  
11.    TIM1->PSC = 8000-1;  
12.    TIM1->ARR = 200-1;  
13.  
14.    TIM1->CCR1 = 50;  
15.    TIM1->CCR2 = 50;  
16.    TIM1->CCR3 = 20;  
17.    TIM1->CCR4 = 150;  
18.  
19.    TIM1->CCMR1 = TIM_CCMR1_OC1M_3 | TIM_CCMR1_OC1M_2 | TIM_CCMR1_OC1M_1  
20.        | TIM_CCMR1_OC2M_3 | TIM_CCMR1_OC2M_2 | TIM_CCMR1_OC2M_1;  
21.    TIM1->CCMR2 = TIM_CCMR2_OC3M_3 | TIM_CCMR2_OC3M_2 | TIM_CCMR2_OC3M_1  
22.        | TIM_CCMR2_OC4M_3 | TIM_CCMR2_OC4M_2 | TIM_CCMR2_OC4M_1;  
23.    TIM1->CCER = TIM_CCER_CC1E | TIM_CCER_CC3E;  
24.    TIM1->BDTR = TIM_BDTR_MOE;  
25.    TIM1->DIER = TIM_DIER_UIE;  
26.  
27.    TIM1->CR1 = TIM_CR1_CMS_0 | TIM_CR1_CEN;  
28.    SysTick_Config(8000000/2);  
29.    NVIC_EnableIRQ(TIM1_UP_TIM16 IRQn);  
30.    while(1);  
31.  
32. }  
33.  
34. void TIM1_UP_TIM16_IRQHandler(void){  
35.     if(TIM1->SR & TIM_SR UIF){  
36.         TIM1->SR &= ~TIM_SR UIF;  
37.         GPIOC->ODR ^= PC10;  
38.     }  
39. }  
40.  
41. void SysTick_Handler(void){  
42.     GPIOA->ODR ^= PA5;  
43. }
```

19 - 22) konfiguracja kanałów w trybie *asymmetric pwm mode 1*. W programie wykorzystywane są dwa wyjścia (CH1 i CH3). Zgodnie z tabelką 8.3 za konfigurację kanału pierwszego odpowiadają rejestrów CCR1 i CCR2. Należy skonfigurować **oba** bloki *capture/compare* związane z tymi rejestrami (pierwszy i drugi), przy czym:

- przynajmniej jeden z nich musi być ustawiony w trybie *asymmetric pwm* - na tym wyjściu generowany będzie przebieg „asymetryczne pwm”
- drugi kanał może być skonfigurowany w dowolnym trybie pwm - na wyjściu związanym z tym kanałem dostępny będzie przebieg wynikający z wybranego trybu (może być asymetryczne pwm, może być zwykłe pwm)

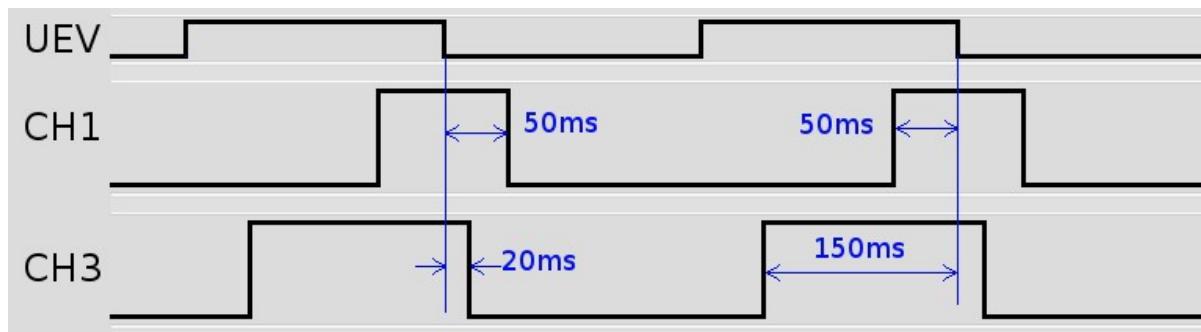
Analogicznie w rejestrze CCMR2 konfiguruje się blok trzeci i czwarty.

23) włączam wyjścia z dwóch kanałów (1 i 3)

25) dorzuciłem przerwanie od przepelnienia licznika, żeby na analizatorze mieć punkt odniesienia. W przerwaniu macham pinem PC10.

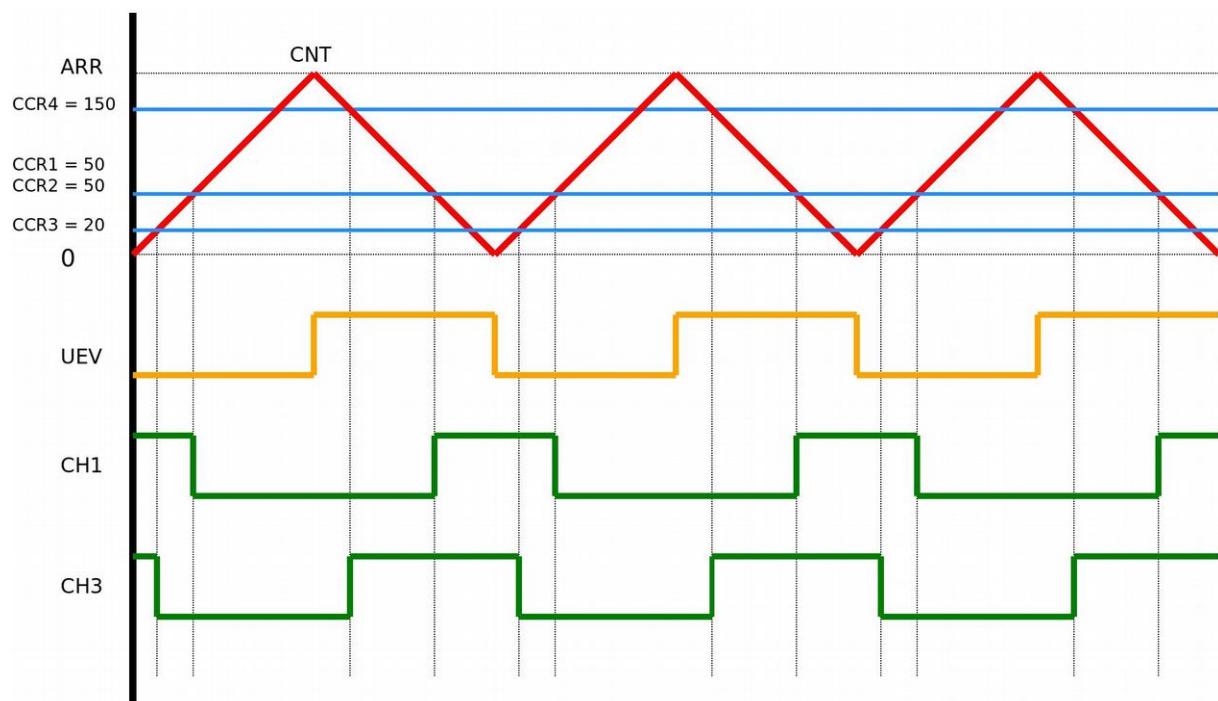
27) zwracam uwagę na bit CMS (*center-aligned mode*), asymetryczny pwm działa tylko gdy licznik zlicza symetrycznie. Wybór jednego z trzech dostępnych trybów (CMS = 0b01/0b10/0b11) nie ma tu znaczenia.

Efekt działania powyższego programu przedstawiony jest na poniższym rysunku.



Rys. 8.17 Przebieg uzyskany w trybie *asymmetric pwm mode 1* (CCR1 = 50, CCR2 = 50, CCR3 = 20, CCR4 = 150)

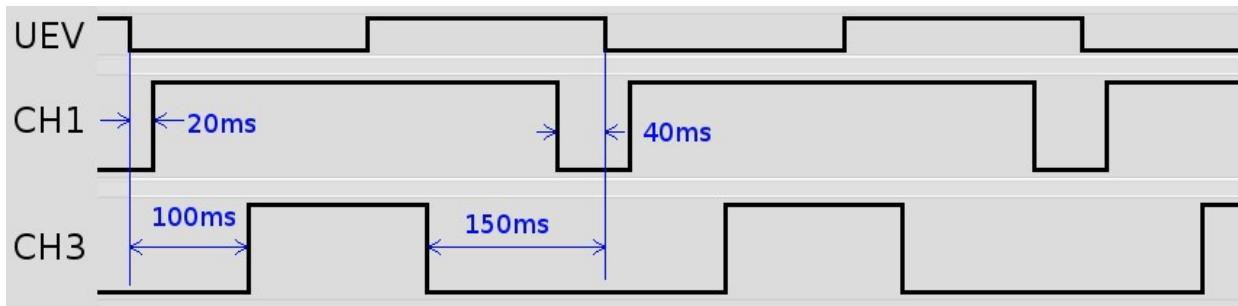
Jako, że się ostatnio rozwinąłem graficznie w Poradniku, to spłodziłem jeszcze jedną kolorowaną (rysunek 8.18). Może służyć pomocą przy dalszym opisie.



Rys. 8.18 Zasada działania trybu *asymmetric pwm*

Licznik zlicza symetrycznie (czerwony przebieg na rysunku 8.18). Górnny przebieg (na rysunku 8.17 oraz żółty na 8.18) zmienia się co przepelnienie licznika (UEV). Zbocze opadające przebiegu wyznacza moment, kiedy licznik rozpoczyna zliczanie w góre. Preskaler licznika został dobrany tak, aby wartości rejestrów CCRx odpowiadały (mniej więcej) milisekundom. No więc licznik zaczyna liczyć od zera, w góre. Dochodzi do wartości CCR3 (20). Ta wartość wyznacza pierwsze zbocze przebiegu generowanego na kanale trzecim. W trybie *mode 1* jest to akurat zbocze opadające. Licznik liczy dalej, aż do osiągnięcia kolejnej wartości porównawczej - CCR1 (50). Wtedy generowane jest pierwsze zbocze przebiegu na pierwszym kanale. Licznik kontynuuje zliczanie, aż do wartości rejestru przeładowania ARR, następuje UEV. Kierunek zliczania jest odwracany, licznik rozpoczyna zliczanie od ARR, w dół. Gdy dojdzie do wartości CCR4 (150), pojawia się drugie zbocze sygnału na trzecim kanale. Wartość z rejestru CCR2 (50) wyznacza drugie zbocze (rosnące) sygnału z kanału pierwszego. I zapetlij.

Standardowo jak zawsze, nie wiem po co jest *mode 1* i *mode 2*, gdyż na moje oko ten sam efekt można uzyskać negując wyjście. Na rysunku 8.19 pokazany jest przebieg uzyskany w trybie *mode 2* dla wartości CCRx jak w podpisie rysunku.



Rys. 8.19 Przebieg uzyskany w trybie *asymmetric pwm mode 2* (CCR1 = 20, CCR2 = 100, CCR3 = 40, CCR4 = 150)

Jeśli ktoś ma jeszcze ochotę na eksperymenty, to proponuję pomieszać *mode 1* i *mode 2* w ramach jednej grupy. Tzn. żeby dwa bloki sterujące jednym wyjściem (np. CCR1 i CCR2 sterujące kanałem pierwszym) miały ustawione różne *mode'y*. Przyjemnej zabawy :>

Jest jeszcze trzeci tryb - *combined pwm*. Podobnie jak poprzednio, konfiguracja *combined pwm*, umożliwia uzyskanie dwóch pwmów z ustalonym przesunięciem/opóźnieniem. Działanie tej konfiguracji opiera się na sumie (*combined pwm mode 1*) lub iloczynie (*combined pwm mode 2*) sygnałów OC1REF i OC2REF (lub OC3REF i OC4REF). Na pewno jest to bardzo ciekawa opcja, która ma mnóstwo zastosowań, i tak dalej... i w ogóle... i wcale nie jest to już nudne...

I jest też czwarty... *combined 3-phase pwm*. To taki tryb, w którym generowane są maksymalnie trzy przebiegi pwm. Każdy może być wynikiem iloczynu logicznego sygnału

referencyjnego OCxREF (gdzie x to numer kanału) i sygnału OC5REF (patrz rejestr TIM_CCR5). Doczytaj we własnym zakresie, mnie już pwmy obrzydły :)

Na wypadek, gdyby ktoś stracił wątek przypominam, że omawiamy nowości w licznikach układu F334. Właśnie skończyliśmy z nowymi trybami pwm. Na koniec zostało nam podsumowanie różnic między poszczególnymi rodzajami liczników w F334. Czas na tabelę :]

Tabela 8.4 Porównanie typów liczników mikrokontrolera F334

Licznik (numery)	Kierunek zliczania	Kanały CC ¹³⁵	Uwagi	Źródła przerwań i żądań DMA
advanced 1	góra, dół, symetrycznie	$4 + 2^{136}$	- wyjścia komplementarne - generator czasu martwego - funkcja break (x2) - licznik powtórzeń - interfejs enkodera	UEV, init, trigger, capture, compare, compare, break
general purpose 2^{137}, 3	góra, dół, symetrycznie	4	- interfejs enkodera	UEV, init, trigger, capture, compare, compare
general purpose 15	tylko w góre	2 (pwm tylko edge-aligned)	- wyjścia komplementarne (tylko 1 kanał) - generator czasu martwego (tylko 1 kanał) - funkcja break - licznik powtórzeń	UEV, init, trigger, capture, compare, compare, break
general purpose 16, 17	tylko w góre	1 (pwm tylko edge-aligned)	- wyjścia komplementarne - generator czasu martwego - funkcja break - licznik powtórzeń - brak możliwości synchronizacji z innymi licznikami	UEV, trigger, capture, compare, break
basic 6, 7	tylko w góre	0	- dedykowany do wyzwalania przetwornika DAC	UEV

Na zakończenie, dla rozweselenia po tym nudnym rozdziale, zaśpiewajmy wspólnie z (nu)Cleo:

„My Słowianie wiemy jak licznik ma nam działać!”

I tym optymistycznym akcentem przejdźmy do podsumowania :]

Co warto zapamiętać z tego rozdziału?

- liczniki w F334 działają podobnie jak w F103 i F429, mają tylko kilka nowych funkcji

135 Capture / Compare

136 tylko funkcja porównująca (compare) i brak obwodów wyjściowych

137 licznik TIM2 jest 32-bitowy! Rejestry CNT, ARR, CCRx tego licznika są 32-bitowe.

- *Retriggerable One Pulse Mode* - impuls o określonej długości generowany niezwłocznie po wystąpieniu trygierza
- *Asymmetric PWM* - dwa przebiegi pwm przesunięte zgodnie z życzeniem programisty

9. BATTERY BACKUP DOMAIN (“*NON OMNIS MORIAR*”¹³⁸)

9.1. Wstęp

Tak się zastanawiam jak przetłumaczyć tytuł rozdziału... *Google-translate* podpowiada, że *Battery Backup Domain* to „*bateria zapasowa domeną*”. No niezbyt szczęśliwie mu to wyszło. Chociaż i tak lepiej niż inne propozycje: np. aby przetłumaczyć *domain* jako *dominium*¹³⁹... Mniejsza z tym. Proponuję niezbyt ambitne tłumaczenie: *strefa/domena podtrzymywana baterijnie*.

Jak zawsze przy STMach sprawa jest niebywale prosta. Część układów mikrokontrolera, przy braku głównego zasilania na nóżce V_{DD} , może być zasilana z baterii (nóżka V_{BAT}). Do układów strefy baterijnej należą:

- rejesty podtrzymywane baterijnie - *BKP* i *backup SRAM* (tylko F429)
- zegar czasu rzeczywistego - *RTC*
- zewnętrzny oscylator zegarkowy - *LSE*
- rejestr kontrolny domeny baterijnej - *RCC_BDCR*

Co do kwestii elektrycznych:

- napięcie baterii:
 - F103: 1,8 - 3,6V
 - F334: 1,65 - 3,6V
 - F429: 1,65 - 3,6V
- pobór prądu (orientacyjnie):
 - F103: ~1,5 μ A
 - F334: ~1 μ A
 - F429: ~10 μ A

W zestawie HY-mini jest gniazdo na baterię CR1220. Swoją włożyłem z dwa lata temu i dalej działa :) W STM32F429i-Disco nóżka V_{BAT} jest połączona z głównym zasilaniem. Bez drobnych modyfikacji nie jest możliwe dołączenie zewnętrznego źródła podtrymania. W zestawie Nucleo-F334R8 w ogóle nie ma możliwości podłączenia źródła do V_{bat} . To wyprowadzenie jest na stałe połączone z V_{cc} i nie przewidziano możliwości zmiany konfiguracji.

138 „*Nie wszyscy umrę.*”

139 co to kur(sy)wa jest *dominium*!?

Co warto zapamiętać z tego rozdziału?

- *battery backup domain* to po prostu część układów mikrokontrolera, które mogą mieć zasilanie podtrzymywane z baterii

9.2. Backup Registers (F103)

Mikrokontrolery rodziny STM32F1 mają pamięć Flash i SRAM. Brak im natomiast, znanej z AVRów, pamięci EEPROM. Pojawia się więc pytanie - jak przechowywać dane, które mają przetrwać reset czy zanik zasilania? Opcji jest kilka:

- zewnętrzna pamięć – to chyba nie wymaga specjalnego komentarza
- zapisywanie danych w wbudowanej pamięci Flash – proste i skuteczne rozwiązanie, szczególnie że pamięci jest dużo; ST wydało nawet jakąś notę zgłębiającą temat
- rejesty *backup*

Nas interesuje to ostatnie rozwiązanie. Rejestry *backup* to czterdzieści dwa 16 bitowe rejesty (BKP_DRx) leżące w *strefie baterijnej*. Czyli po ludzku to taki kawałek pamięci, który jest podtrzymywany baterijnie po zaniku zasilania. Dodatkowo rejesty te **nie są zerowane przy resecie**. Reset domeny baterijnej można wymusić poprzez bit BDRST w rejestrze RCC_BDCR.

Czy to lepsze rozwiązanie niż EEPROM? Inne. Wszystko ma swoje wady i zalety, nie mnie osądzać. Na pewno zaletą jest to, że jest to pamięć SRAM, czyli nie ulega degradacji podczas zapisów tak jak EEPROM. Konieczność podtrzymywania zasilania jest pewną wadą. Coś za coś.

Ciekawą funkcją rejestrów BKP jest *Tamper Detection* (detekcja sabotażu / majstrowania). Wykrycie próby majstrowania (a dokładniej zmiana stanu pinu PC13, który jest wejściem *tamper*) powoduje sprzętowe skasowanie zawartości BKP. Po co? A np. próba włamania do centralki alarmowej powoduje skasowanie jakiś kodów ze sterownika, kluczy szyfrujących. Czy cokolwiek w tym guście :) Wystarczy mały styk przy obudowie i przewodzik do wejścia *tamper*.

Zadanie domowe 9.1: zapisać coś do rejestrów BKP i sprawdzić czy dane przetrwały przerwę w zasilaniu. W zależności od wyniku testu zapalić jedną lub drugą diodę. Dodatkowe zadanie „z gwiazdką” - dodać wykrywanie sabotażu.

Przykładowe rozwiązanie (F103, diody na PB0 i PB1, przycisk tamper na PC13):

```
1. int main(void) {
2.
3.     RCC->APB2ENR = RCC_APB2ENR_IOPBEN;
4.     gpio_pin_cfg(GPIOB, PB0, gpio_mode_output_PP_2MHz);
5.     gpio_pin_cfg(GPIOB, PB1, gpio_mode_output_PP_2MHz);
6.
7.     RCC->APB1ENR = RCC_APB1ENR_PWREN | RCC_APB1ENR_BKPEN;
8.     PWR->CR = PWR_CR_DBP;
9.     BKP->CR = BKP_CR_TPAL | BKP_CR_TPE;
10.
11.    if ((BKP->DR1 == 152) && (BKP->DR2 == 348)) {
12.        BB(GPIOB->ODR, PB0) = 1;
13.    } else {
14.        BKP->DR1 = 152;
15.        BKP->DR2 = 348;
16.        BB(GPIOB->ODR, PB1) = 1;
17.    }
18.
19.    while(1){
20.        if (BKP->CSR & BKP_CSR_TEF){
21.
22.            BB(GPIOB->ODR, PB0) = 1;
23.            BB(GPIOB->ODR, PB1) = 1;
24.
25.            BB(BKP->CSR, BKP_CSR_CTE) = 1;
26.
27.            BB(BKP->CR, BKP_CR_TPE) = 0;
28.            BB(BKP->CR, BKP_CR_TPE) = 1;
29.        }
30.    }
31. }
```

Uprzedzam, żeby nie było, że ten przykład nie działa w 100% tak jakbym to sobie życzył... i co jest jeszcze bardziej frustrujące - nie jestem pewny dlaczego :)

3) włączenie zegarów portu B (diody) i portu C (wejście tamper)... a nie zaraz!! nie ma zegara dla portu C! Wejście tamper jest dosyć specyficzne - np. jest w stanie wykryć próbę sabotażu nawet, gdy procek nie jest zasilany (tzn. jest ale tylko z V_{BAT}). Wtedy na pewno nie ma zegara portu a jednak działa... Z tego co widzę w przykładach do biblioteki SPL¹⁴⁰, zegar portu nie jest włączany przy korzystaniu z funkcji tamper... Pobawiłem się i różnicy między włączonym a nie włączonym nie widzę - więc po co przepłacać.

Czego jeszcze nie ma? Konfiguracji pinu PC13. Nie ma potrzeby! W rozdziale opisującym sposób konfiguracji pinów do współpracy z układami peryferyjnymi¹⁴¹ jest informacja, że konfiguracja tamper pin jest *forced by hardware*. No i fajnie.

7) włączamy dwa bloki: BKP i PWR. I tu znowu ciekawostka. To co włączamy to tak naprawdę nie taktowanie bloku BKP, tylko taktowanie interfejsu, który pozwala nam się dobrać do rejestrów domeny backup. Z kolei PWR musimy włączyć bo tam siedzi specjalny bit (patrz 8 linijka kodu) odblokowujący możliwość zapisu do rejestrów domeny. To taka dodatkowa ochrona danych zapisanych w BKP.

140 w chwilach zwątpienia można wesprzeć się przykładowymi programami dołączonymi do biblioteki, np. aby sprawdzić kolejność wykonywania jakichś operacji

141 *GPIO configurations for device peripherals*

9) włączenie funkcji tamper i ustalenie polaryzacji wejścia

11 - 17) sprawdzam zawartość dwóch rejestrów BKP i jeśli jest tam moja magiczna liczba to zapalam diodę na PB0, w przeciwnym wypadku zapisuję moją magiczną liczbę w BKP i zapalam drugą diodę

20) w pętli nieskończonej, sprawdzam czy jest ustawiona flaga tamper, czyli czy funkcja zadziałała. Jeśli tak to sygnalizuję to zapaleniem obu diod. Potem kasuję flagę i na chwilę wyłączam tamper (tak każe RM). Zamiast pollingu w pętli, można również wykorzystać przerwanie od funkcji tamper, ale nie chciałem zaciemniać przykładu.

Generalnie idea programu jest chyba oczywista. Po pierwszym uruchomieniu, gdy w BKP nie będzie mojego tajnego kodu, program ma go tam wpisać i zasygnalizować to diodą na PB1. Kolejne uruchomienia programu mają dowieźć, że w BKP jest to co tam wpisałem (mimo np. resetu czy zaniku głównego zasilania) - co ma sygnalizować dioda na PB0. Dodatkowo włączona jest funkcja wykrywania sabotażu.

Część związana z BKP działa nieskazitelnie. Problem jest natomiast z wykrywaniem sabotażu. Na płytce HY-mini pin PC13 jest podciagnięty do V_{dd} i przyciskiem zwierany do masy. Niezbyt to szczęśliwe rozwiązanie bo wymusza aktywowanie funkcji tamper stanem niskim (po wciśnięciu przycisku). Po wyłączeniu zasilania płytka podciaganie do V_{DD} nie działa i mikrokontroler odczytuje na wejściu stan niski → odpala tamper. Obszedłem to naokoło, łącząc PC13 z V_{bat} żeby zawsze miał stan wysoki. Problem polega na tym, że pomimo tego, raz na kilkanaście wyłączeń/włączeń zasilania tamper się aktywuje. Przypuszczam, że to problem bardziej sprzętowy niż programowy. Tak czy siak uczciwie uprzedzam :)

Ważna uwaga: konfiguracja funkcji tamper siedzi w backup domain. A to oznacza, że przy resecie procesora nie jest zerowana (wspomniałem o tym w poprzednim rozdziale)! Czyli jeśli wgramy program, który włącza tamper. A potem nam się znudzi i wgramy program który tampera nie włącza, to tamper nadal będzie włączony. Trzeba go ręcznie wyłączyć, bo bit TPE w BKP_CR przy resecie się nie skasuje. Ewentualnie można zresetować całą domenę backup (bit BDRST w RCC_BDCR).

I tyle w temacie. Miłej zabawy.

Co warto zapamiętać z tego rozdziału?

- rejesty backup zachowują zawartość podczas resetu mikrokontrolera i przerw zasilania
- funkcja anty-sabotażowa umożliwia sprzętowe wymazanie zawartości rejestrów backup

9.3. RTC (F103)

Miało już nie być liczników a tu... RTC (*Real Time Clock*). Zegar czasu rzeczywistego w F103 jest po prostu 32 bitowym licznikiem leżącym w domenie bekap. Wskazanie licznika i jego konfiguracja¹⁴² są podtrzymywane baterijne. Najważniejsze cechy licznika:

- 32 bity
- preskaler ze stopniem podziału do 2^{20}
- taktowanie licznika (się rozjaśni w rozdziale 17):
 - *HSE/128*
 - *LSE*
 - *LSI*
- trzy dedykowane przerwania:
 - *Alarm Interrupt* – taki budzik, który można ustawić na konkretny czas
 - *Second Interrupt* – przerwanie zegarowe wywoływanie przez sygnał wychodzący z preskalera RTC a wchodzący na właściwy licznik RTC; preskaler można dobrać tak aby licznik tykał co sekundę, dzięki temu licznik będzie zliczał czas w sposób wygodny do obróbki¹⁴³ a przerwanie *second* będzie się odpalało co sekundę
 - *Overflow Interrupt* – przerwanie od przekręcenia licznika¹⁴⁴
- funkcję alarmu można wykorzystać do wybudzenia procka z najgłębszego trybu uśpienia (*Standby*) – patrz rozdział o uśpieniach (11.6)

Zadanie domowe 9.2: uruchomić zegar RTC. Włączyć przerwanie sekundowe, w jego ISR migać diodą. Za pomocą funkcji alarmu i jego przerwania, migać drugą diodą co 10s.

142 tylko rejestrów RTC_PRL, RTC_ALR, RTC_CNT i RTC_DIV

143 hasło klucz: *Unix Time, POSIX Time*

144 już niedługo :) 32bitowy Unix Time przekręci się w styczniu 2038r.

Przykładowe rozwiązanie (F103, diody na PB0 i PB1):

```
1. int main(void) {
2.
3.     RCC->APB2ENR = RCC_APB2ENR_IOPBEN;
4.     gpio_pin_cfg(GPIOB, PB0, gpio_mode_output_PP_2MHz);
5.     gpio_pin_cfg(GPIOB, PB1, gpio_mode_output_PP_2MHz);
6.
7.     RCC->APB1ENR = RCC_APB1ENR_PWREN | RCC_APB1ENR_BKPPEN;
8.     PWR->CR |= PWR_CR_DBP;
9.
10.    RCC->CSR |= RCC_CSR_LSION;
11.    while ( BB(RCC->CSR, RCC_CSR_LSIRDY) == 0 );
12.    RCC->BDCR |= RCC_BDCR_RTCEN | RCC_BDCR_RTCSEL_LSI;
13.
14.    BB(RTC->CRL, RTC_CRL_RSF) = 0;
15.    while ( BB(RTC->CRL, RTC_CRL_RSF) == 0 );
16.
17.    while ( BB(RTC->CRL, RTC_CRL_RTOFF) == 0 );
18.    BB(RTC->CRL, RTC_CRL_CNF) = 1;
19.
20.    RTC->CRH = RTC_CRH_ALRIE | RTC_CRH_SECIE;
21.    RTC->PRLL = 40000-1;
22.    RTC->PRLH = 0;
23.    RTC->CNTL = 0;
24.    RTC->CNTH = 0;
25.    RTC->ALRL = 10-1;
26.    RTC->ALRH = 0;
27.
28.    BB(RTC->CRL, RTC_CRL_CNF) = 0;
29.    while ( BB(RTC->CRL, RTC_CRL_RTOFF) == 0 );
30.
31.    NVIC_ClearPendingIRQ(RTC_IRQn);
32.    NVIC_EnableIRQ(RTC_IRQn);
33.
34.    while(1);
35. }
36.
37. __attribute__((interrupt)) void RTC_IRQHandler(void){
38.
39.     if ( BB(RTC->CRL, RTC_CRL_ALRF) == 1 ){
40.
41.         while ( BB(RTC->CRL, RTC_CRL_RTOFF) == 0 );
42.         BB(RTC->CRL, RTC_CRL_CNF) = 1;
43.
44.         BB(RTC->CRL, RTC_CRL_ALRF) = 0;
45.         RTC->CNTL = 0;
46.         RTC->CNTH = 0;
47.
48.         BB(RTC->CRL, RTC_CRL_CNF) = 0;
49.         while ( BB(RTC->CRL, RTC_CRL_RTOFF) == 0 );
50.
51.         BB(GPIOB->ODR, PB1) ^= 1;
52.     }
53.
54.     if ( BB(RTC->CRL, RTC_CRL_SECF) == 1 ){
55.         while ( BB(RTC->CRL, RTC_CRL_RTOFF) == 0 );
56.         BB(RTC->CRL, RTC_CRL_SECF) = 0;
57.         BB(GPIOB->ODR, PB0) ^= 1;
58.     }
59. }
```

Długaśne to wyszło...

3 - 5) zegar, diody - *nihil novi*

7, 8) włączam zegar dla bloków PWR i BKP, następnie odblokowuję możliwość zapisu do rejestrów domeny

10, 11) włączam wewnętrzny oscylator małej częstotliwości (LSI) i czekam aż się rozbuja (więcej szczegółów nt. oscylatorów będzie w rozdziale o systemie zegarowym - 17)

12) włączenie RTC i wybór źródła taktowania (polecam zerknąć na drzewko zegarowe)

14, 15) to jest ciekawe. RTC (w tym jego rejesty konfiguracyjne) są taktowane innym sygnałem zegarowym niż interfejs przez który się z nimi komunikujemy. Powoduje to, że po włączeniu zegara¹⁴⁵ musi minąć chwilka, żeby obie domeny zegarowe się zsynchronizowały. W przeciwnym razie mamy sporą szansę na to, że odczytamy z rejestrów RTC śmieci. RTC_CRL_RSF to flaga ustawiana po zsynchronizowaniu się obu domen. I na niej opiera się mechanizm, który pozwala nam upewnić się, że wszystko jest ok. Flagę kasujemy, po czym czekamy aż się znowu ustawi. To daje nam pewność, że wszystko jest zsynchronizowane :) Tzn. może trochę bzdurzę... ale staram się jak mogę!

17, 18) kolejna ciekawostka i znowu chodzi o te zegary. Zapis do rejestru licznika RTC nie jest natychmiastowy, bo licznik i jego rejesty działają na swoim wolnym oscylatorku. Bit RTC_CRL_RTOFF pozwala sprawdzić czy poprzednia operacja zapisu już się zakończyła. RM podaje przepis na zapisanie czegoś do rejestrów RTC:

- poczekać na zakończenie poprzedniej operacji (RTOFF = 1)
- wejść w tryb konfiguracji¹⁴⁶ (CNF = 1)
- zapisać nowe wartości rejestrów¹⁴⁷ licznika
- opuścić tryb konfiguracji (CNF = 0)
- poczekać na zakończenie zapisu (RTOFF = 1)

Czas na właściwą konfigurację licznika:

20) włączam przerwania *alarm* i *second*

21, 22) ustawiam podział preskalera na 40 000 (LSI ma coś koło 40kHz), dzięki temu licznik będzie zliczał (pi razy drzwi) sekundy, a przerwanie sekundowe będzie się odpalało rzeczywiście co circa sekundę

23, 24) zeruję sobie rejestr licznika

25, 26) alarm ustawiam na 10 zliczeń licznika RTC (czyli mniej więcej 10 sekund bo tak dobraliśmy preskaler), licznik zlicza od 0 stąd „-1”

28, 29) wychodzę z trybu konfiguracji i czekam na zakończenie zapisu

37) tu mamy przerwanie od RTC. W przerwaniu sprawdzane są flagi, aby określić jego źródło. Jeśli przerwanie zostało wywołane przez *alarm* to zeruję licznik RTC (oczywiście przechodzę całą procedurę z bitami RTOFF i CNF) i macham diodą. Zerowanie licznika nie jest specjalnie

145 dotyczy również sytuacji, w której zegar był wyłączony ze względu na uśpienie procka

146 tryb konfiguracyjny jest wymagany przy zmianie wartości rejestrów RTC_PRL, RTC_CNT, RTC_ALR

147 w RM jest mowa o rejestrach (liczba mnoga), w przykładach do SPL bit RTOFF jest sprawdzany po każdym zapisie do pojedynczego rejestru... bałagan...

eleganckim rozwiązaniem, ale dzięki temu będzie zliczał znowu od zera i za 10s znowu pojawi się przerwanie od alarmu.

54) w przerwaniu *sekundowym*, z kolei, czekam na zakończenie poprzedniej operacji zapisu i kasuję flagę przerwania oraz migam diodą. Nie wchodzę tutaj w tryb konfiguracyjny (CNF) gdyż jest on wymagany tylko przy modyfikacji rejestrów RTC_PRL, RTC_CNT, RTC_ALR. Nie sprawdzam również bitu RTOFF na końcu ISR bo... wydaje mi się, że wystarczy sprawdzać flagę RTOFF przed zapisem i chciałem sprawdzić czy tak będzie działać - działa... ale jest nie do końca zgodnie z RM

Z przerwaniami od RTC jest jeszcze jedna ciekawostka. Może pamiętasz, w rozdziale o przerwaniach zewnętrznych (rozdział 7.1) wspominałem, że *Alarm RTC* jest też podpięty do 17-tej linii EXTI. Czyli wychodzi na to, że jest on związany z dwoma przerwaniami (RTC_IRQn oraz RTCAlarm_IRQn). Po co to zamieszanie? Otóż linia EXTI17 (przerwanie RTCAlarm_IRQn) jest wykorzystywana do budzenia procesora z trybu uśpienia Standby.Więcej na ten temat będzie w rozdziale poświęconym trybom uśpienia (rozdział 11.6).

Co warto zapamiętać z tego rozdziału?

- w F103 wbudowany jest układ RTC
- RTC to dużo powiedziane, bo po prostu odmierza sekundy... ma to swoje wady i zalety
- Alarm zegara RTC może wybudzać mikrokontroler ze stanu uśpienia Standby

9.4. Backup Registers (F429)

Tym razem mamy dwadzieścia 32 bitowych rejestrów RTC_BKPxR. Jak widać, rejesty backup stały się częścią bloku zegara RTC. Druga różnica to to, że jest troszkę więcej opcji konfiguracji funkcji *tamper*. Poza tym nic się nie zmienia w stosunku do F103.

Wejście *tamper* może być na jednym z dwóch pinów (PC13 lub PI8¹⁴⁸). Wybór wejścia dokonywany jest poprzez bit RTC_TAFCR_TAMP1INSEL. Do wyboru są dwie opcje:

- RTC_AF1 used as TAMPER1
- RTC_AF2 used as TAMPER1

Pierwsza kropka to PC13, druga to PI8 (patrz rozdział *Selection of RTC_AF1 and RTC_AF2 alternate functions* w RM). Przy czym nie należy tych AF1 i AF2 mylić z funkcjami alternatywnymi wybieranymi przy konfiguracji portów GPIO. Włączenie tampera całkowicie

148 jeśli mikrokontroler występuje w odpowiednio dużej obudowie

przejmuje kontrolę nad pinem. Nie jest wymagana żadna inna konfiguracja ani włączanie zegara portu. Jedyna konfiguracja to ta w rejestrze RTC_TAFCR.

STM32F429 ma to czego brakowało w STM32F103, czyli możliwość włączenia pull-upów dla wejścia tamper. W celu ograniczenia zużycia energii, w końcu jedynym źródłem zasilania może być baterijka podtrzymująca pamięć, podciąganie nie jest włączone na stałe. Pull-upy włączają się na ustalony okres tuż przed sprawdzeniem stanu panującego na pinie. Chodzi o to aby podładować ewentualne pojemności na linii tamper. Stąd też i w RM jest to określane jako *precharge*. Za pomocą bitu RTC_TAFCR_TAMPPUDIUS możemy wyłączyć to podładowywanie (domyślnie jest włączone). Ponadto bity RTC_TAFCR_TAMPPRCH pozwalają regulować czas ładowania (czyli jak długo mają być włączone rezystory podciągające *tamper*). Podciąganie działa tylko, jeśli funkcja anty-sabotażowa jest aktywowana stanem (a nie zboczem).

Dalej mamy możliwość filtrowania sygnału wejściowego. Za pomocą bitów RTC_TAFCR_TAMPFLT i RTC_TAFCR_TAMPFREQ wybieramy długość i częstotliwość próbkowania wejścia. Do pełni szczęścia są jeszcze bity RTC_TAFCR_TAMPxTRG pozwalające ustawić poziom lub zbocze aktywujące funkcję.

Ostatni bajer to funkcja *timestamp*, która zapisuje aktualny czas (pobrany z RTC) po wykryciu próby sabotażu. Po szczegółach odsyłam wiadomo gdzie.

Zadanie domowe 9.3: przerobić kod z zadania 9.1 tak, aby działał na STM32F429.

Przykładowe rozwiązanie (F429¹⁴⁹, diody na PG13 i PG14, wejście anty-sabotażowe na PC13):

```
1. int main(void) {
2.
3.     RCC->AHB1ENR |= RCC_AHB1ENR_GPIOGEN;
4.     RCC->APB1ENR = RCC_APB1ENR_PWREN;
5.     __DSB();
6.
7.     gpio_pin_cfg(GPIOG, PG13, gpio_mode_out_PP_LS);
8.     gpio_pin_cfg(GPIOG, PG14, gpio_mode_out_PP_LS);
9.
10.
11.    RCC->CSR |= RCC_CSR_LSION;
12.    while( !(RCC->CSR & RCC_CSR_LSION) );
13.    RCC->BDCR |= RCC_BDCR_RTCSEL_1 | RCC_BDCR_RTCEN;
14.    __DSB();
15.
16.    PWR->CR = PWR_CR_DBP;
17.    RTC->TAFCR = RTC_TAFCR_TAMPPRCH | RTC_TAFCR_TAMPFLT | RTC_TAFCR_TAMPFREQ_2
18.        | RTC_TAFCR_TAMP1E;
19.
20.    if ((RTC->BKP0R == 152) && (RTC->BKP1R == 348)) {
21.        BB(GPIOG->ODR, PG13) = 1;
22.    } else {
23.        RTC->BKP0R = 152;
24.        RTC->BKP1R = 348;
25.        BB(GPIOG->ODR, PG14) = 1;
26.    }
27.
28.    while (1) {
29.        if (RTC->ISR & RTC_ISR_TAMP1F) {
30.
31.            BB(GPIOG->ODR, PG13) = 1;
32.            BB(GPIOG->ODR, PG14) = 1;
33.
34.            BB(RTC->ISR, RTC_ISR_TAMP1F) = 0;
35.
36.            BB(RTC->TAFCR, RTC_TAFCR_TAMP1E) = 0;
37.            BB(RTC->TAFCR, RTC_TAFCR_TAMP1E) = 1;
38.        }
39.    }
40. }
```

3, 4, 5) włączam zegar portu G (diody) i bloku PWR. Proszę zwrócić uwagę, że nie włączam żadnego zegara dla interfejsu RTC! Zegar bloku RTC włączymy za sekundę w innym rejestrze.

7, 8) konfiguracja portów diodowych (wyjście, push-pull, low speed)

11, 12) włączam wewnętrzny oscylator niskiej częstotliwości i czekam aż się rozburzy (szczegóły w rozdziale 17)

13) wybieram źródło taktowania RTC i włączam zegar bloku RTC

14) profilaktycznie, żeby RTC zdążył się rozkręcić

16) odblokowanie zapisu do rejestrów RTC

17) konfiguracja funkcji tamper: wejście PC13, czas podładowywania na maks., wyzwalane stanem niskim i jakieś tam filtrowanie. Dalej jest po staremu :)

Funkcja tamper w F429 wydaje się być trochę pokręcona w konfiguracji, ale za to działa bezbłędnie :) Jako ciekawostkę powiem, że przy ustawieniu niskiej częstotliwości próbkowania

¹⁴⁹ płytka STM32F429i-Disco niestety nie ma możliwości (bez kombinowania) podłączenia zewnętrznego źródła do V_{bat} więc nie mogę sprawdzić czy działa podtrzymanie baterijne

wejścia i „długiego” filtrowania, stan aktywny na wejściu tamper musi się utrzymywać aż kilka sekund żeby funkcja zadziałała :) Zdecydowanie wolę to rozwiązanie niż nad-aktywny tamper w F103.

Tyle w kwestii rejestrów backup. W F103 nie było dla nich specjalnej alternatywy, w F429 mogłyby ich właściwie nie być bo... patrz następny rozdział.

Co warto zapamiętać z tego rozdziału?

- jak to co? wszystko! nie po to się produkowałem żeby teraz jednym uchem wpadało a drugim wylatyszało :>

9.5. Backup SRAM (F429)

Backup SRAM to po prostu kawałek pamięci SRAM, której zawartość jest podtrzymywana baterijnie. W omawianym mikrokontrolerze, dostępne są aż 4kB pamięci backup SRAM. Pamięć ta dostępna jest od adresu 0x4002 4000. Sposób korzystania z tego kawałka pamięci jest uzależniony od posiadanego środowiska i jego konfiguracji. W wersji najprostszej można sobie latać po backup SRAMie „gołym” wskaźnikiem wedle uznania. Inna opcja (zdecydowanie wygodniejsza i bezpieczniejsza) to dopisanie nowej sekcji do skryptu linkera i korzystanie z backup SRAM jak z (prawie) zwykłej pamięci. Tak czy siak, skrypty linkera to już nie jest temat tego poradnika.

Zadanie domowe 9.4: napisać program, który po resecie sprawdza zawartość pamięci backup SRAM. Jeśli jest tam zapisany nasz super tajny numer to program zapala pierwszą diodę. W przeciwnym wypadku zapisuje w backup SRAM tajny numer i sygnalizuje to drugą diodą.

Przykładowe rozwiążanie (429¹⁵⁰, diody na PG13 i PG14):

```
1. int main(void) {
2.
3.     RCC->AHB1ENR |= RCC_AHB1ENR_GPIOGEN | RCC_AHB1ENR_BKPSRAMEN;
4.     RCC->APB1ENR = RCC_APB1ENR_PREN ;
5.     __DSB();
6.
7.     PWR->CR = PWR_CR_DBP;
8.     PWR->CSR = PWR_CSR_BRE;
9.     while (!( PWR->CSR & PWR_CSR_BRR ));
10.
11.    gpio_pin_cfg(GPIOG, PG13, gpio_mode_out_PP_LS);
12.    gpio_pin_cfg(GPIOG, PG14, gpio_mode_out_PP_LS);
13.
14.    volatile uint32_t zmienna __attribute__((section(".b kp")));
15.    volatile uint32_t * const wskaznik = (uint32_t *)0x40024004;
16.
17.    if ((zmienna == 152) && (*wskaznik == 348)) {
18.        BB(GPIOG->ODR, PG13) = 1;
19.    } else {
20.        zmienna = 152;
21.        *wskaznik = 348;
22.        BB(GPIOG->ODR, PG14) = 1;
23.    }
24.
25.    while (1);
26.
27. }
```

3, 4, 5) włączamy zegary portu G (diody), backup SRAMu i bloku PWR (w nim siedzi bit odpowiedzialny za blokadę zapisu do rejestrów domeny baterijnej)

7) odblokowanie możliwości zapisu rejestrów strefy baterijkowej

8, 9) włączenie regulatora napięcia podtrzymującego backup SRAM w trybie uśpienia *standby* i przy braku zasilania głównego. Domyślnie regulator jest wyłączony w celu oszczędzania energii, więc po zaniku V_{dd} (lub wejściu w uśpienie *standby*) *backup SRAM* traci zawartość nawet jeśli mamy zasilanie V_{bat}. Flaga BRR oznacza gotowość regulatora. Uwaga! Bit BRE nie jest resetowany przy resecie mikrokontrolera!

11, 12) konfiguracja portów - nuda

14) zmienna utworzona w pamięci backup SRAM, atrybut *section* (komplikator GCC) i nazwa sekcji (zdefiniowana wcześniej w skrypcie linkera) wynikają z używanych narzędzi i ich konfiguracji. Zmienną utworzoną w ten sposób posługuję się w programie jak zwykłą zmienną, co widać w dalszej części listingu.

W ramach sprawdzenia czy zmienna została poprawnie umieszczona w pamięci podtrzymywanej baterijnie, można sprawdzić jej adres. Sposób działania ponownie zależy od posiadanych narzędzi. Ja np. skorzystałem z programu *nm*. Tak czy siak należy się upewnić, że linker umieścił zmienną w przestrzeni backup SRAM. Najpewniej wylądowała na początku pamięci, czyli pod adresem 0x4002 4000.

¹⁵⁰ płytki STM32F429i-Disco niestety nie ma możliwości (bez kombinowania) podłączenia zewnętrznego źródła do V_{bat} więc nie mogę sprawdzić działania programu bez głównego zasilania

15) drugi sposób dostępu do backup SRAM to wskaźnik ustawiony gdzieś w obszarze tej pamięci. Tworzę więc stały wskaźnik i ustawiam go na adres 0x4002 4004. Czemu taki adres? Bo na początku pamięci wylądowała zmienna z 14 linii kodu, miała ona 32b czyli 4B. Wskaźnik postanowiłem ustawić „za” tą zmienną, stąd przesunięcie adresu o 4. W dalszej części programu posługuję się wskaźnikiem jak każdym innym.

Zadanie domowe 9.5: za pomocą debuggera odczytać pamięć spod adresów 0x4002 4000 i 0x4002 4004 i sprawdzić czy wszystko działa zgodnie z założeniami.

Zadanie domowe 9.6: sprawdzić czy po wyzerowaniu bitu DBP w PWR_CR możliwy jest zapis nowej wartości do pamięci backup SRAM przy wykorzystaniu debuggera

Pamięć backup SRAM nie jest oczywiście kasowana przy resecie procesora. Funkcja *tamper* (niestety) też jej nie dotyczy. Skasowanie tej pamięci można uzyskać tylko przez:

- wyłączenie zasilania głównego i podtrzymania baterijnego (przypominam o bicie BRE w PWR_CSR)
- jakieś kombinacje z bitami odpowiedzialnymi za zabezpieczenie pamięci przed odczytem (*Option Bytes*) (nie wiem, nie pytać!)

Co warto zapamiętać z tego rozdziału?

- *backup SRAM* to kawałek pamięci SRAM, który może być podtrzymywany z baterii
- regulator napięcia podtrzymującego SRAM jest domyślnie wyłączony!

9.6. RTC (F429)

Zegar czasu rzeczywistego w tym mikrokontrolerze ma tyleż wspólnego ze swoim bliźniakiem z F103 co radiobudzik z klepsydrą. Tutaj mamy całkowicie inne podejście, prawdziwy zegar z kalendarzem:

- czas (sekundy, minuty, godziny, AM/PM¹⁵¹) przechowywany jest w rejestrze: RTC_TR (*Time Register*)
- data (dzień, miesiąc, dzień tygodnia, rok) przechowywana jest w rejestrze: RTC_DR (*Data Register*)

151 oznaczenie czasu w formacie 12h: *AM* - przed południem, *PM* - po południu

Wszystkie wartości liczbowe zakodowane są w formacie BCD. Ze względu na to, że zegar i interfejs przez który się z nim komunikujemy pracują w różnych domenach zegarowych (problemy z synchronizacją) dostęp do rejestrów daty, czasu i „sub-sekund” (nie pytać!) jest realizowany poprzez *shadow registers*. Tzn. wartości z rejestrów RTC są kopiowane do rejestrów cieni, skąd możemy je sobie odczytywać w programie w dowolnym momencie. Kopiowanie następuje co 2 cykle zegara RTC i powoduje ustawienie flagi RSF w RTC_ISR. Jeśli skasujemy flagę i poczekamy aż się ustawi, to mamy pewność że w rejestrach cieniach jest najnowsza, poprawna wartość skopiowana z RTC. Przy resecie mikrokontrolera i wybudzaniu z uśpienia rejesty cenie są zerowane. Dodatkowo, aby zapewnić spójność danych, po dokonaniu odczytu rejestrów czasu (RTC_TR) (lub „sub-sekund” RTC_SSR), kopowanie wartości do rejestrów cieni zostaje zablokowane aż do odczytania rejestrów daty (RTC_DR). Jeśli chcemy odczytać dane bezpośrednio z rejestrów RTC (bez pośrednictwa rejestrów cieni), bo np. nie chcemy czekać na synchronizację, to możliwe wyłączyć cieniowanie poprzez bit BYPSHAD w rejestrze RTC_CR.

RTC może być taktowany z trzech źródeł (się rozjaśni w rozdziale 17):

- wewnętrznego oscylatora niskiej częstotliwości (LSI)
- zewnętrznego oscylatora niskiej częstotliwości (LSE)
- zewnętrznego oscylatora wysokiej częstotliwości (HSE) z dodatkowym preskalerem (taktowanie RTC nie może przekroczyć 4MHz)

Sygnał zegarowy podawany jest na dwa, połączone szeregowo, preskality - asynchronousny i synchronousny. Nie wiem, nie znam się, nie pytać! Podział ponoć pozwala zmniejszyć zużycie energii (wskażane jest ustawienie jak najwyższego stopnia podziału na pierwszym preskalerze). Z preskalerów powinien wychodzić sygnał 1Hz taktujący zegar. Wzór na częstotliwość sygnału wychodzącego z preskalerów:

$$ck_{spre} = \frac{RTCCLK}{(PREDIV_A + 1) \cdot (PREDIV_S + 1)}$$

gdzie:

- ck_{spre} - częstotliwość sygnału wychodzącego z preskalerów
- RTCCLK - częstotliwość sygnału taktującego blok RTC
- $PREDIV_A$ - nastawa preskalera asynchronousnego
- $PREDIV_S$ - nastawa preskalera synchronousznego

Preskalery mają różne długości (możliwe stopnie podziału) odsyłam do dokumentacji. Btw. proponuję odpalić sobie schemat blokowy RTC.

Zegar ma kilka opcji i mechanizmów pozwalających zwiększyć jego dokładność. Pierwsza opcja jest określona w RM jako **Synchronization**. Z tego co rozumiem (a nie bardzo rozumiem) całość polega na tym, że porównujemy okres zmierzony przez RTC (w rejestrze czasu mamy podany czas z rozdzielczością sekundy, dla zwiększenia rozdzielczości mamy dostęp do rejestru „sub-sekund” - RTC_SSR) z jakimś referencyjnym czasomierzem. Obliczamy poprawkę i wprowadzamy odpowiednią wartość do rejestru RTC_SHIFTER. Wzorki i opis można znaleźć w nocyce *AN3371 Using the hardware real-time clock (RTC) in STM32 F0, F2, F3, F4 and L1 series of MCUs*. Ja mało z tego rozumiem i nie będę udawał, że jest inaczej :)

Druga opcja (i ta mi się podoba niesłychanie) to użycie **zewnętrznego sygnału referencyjnego**. Działa to tak, że do wejścia RTC_REFIN zapodajemy przebieg o częstotliwości sieciowej (50 lub 60Hz)¹⁵². Zegar dalej jest taktowany ze swojego źródła, ale co sekundę, następuje porównanie zbocza sygnału taktującego zegar i zbocza sygnału referencyjnego. W razie wykrycia rozbieżności w następnym okresie dodawana jest (automatycznie) poprawka. Jeśli w określonym okienku czasowym nie pojawi się zbocze sygnału referencyjnego, to układ to olewa i nie wprowadza poprawki. Wydaje się być proste i nader wygodne. Uwaga! Korzystając z tej opcji należy:

- ustawić podział preskalerów na wartości domyślne (PREDIV_A = 0x007F, PREDIV_S = 0x00FF)
- nie korzystać z kalibracji zgrubnej (RTC_CALIBR = 0)

Trzecia opcja (czym to się właściwie różni od opcji 1?) to **cyfrowa kalibracja: zgrubna i dokładna**. Obu metod nie należy używać jednocześnie. Przy zgrubnej proponuję wyrzucić sygnał z RTC na pin, zmierzyć i obliczyć poprawkę. Nastawę kalibracji zgrubnej można zmienić tylko w trybie konfiguracji zegara (zaraz się wyjaśni), stąd nie nadaje się do dynamicznych zmian. Uwaga! kalibracja zgrubna nie zmienia częstotliwości sygnału wyjściowego (512Hz), gdyż blok kalibrujący znajduje się „za” miejscem z którego jest pobierany sygnał wyrzucany na nóżkę (patrz schemat blokowy). Kalibracja dokładna tymczasem, jest polecana do kompensowania wpływu temperatury (na oscylator) czy starzenia się oscylatora. Wybacz, to jest nudne, mam dość. Doczytaj we własnym zakresie, ja i tak nie rozumiem :)

Tyle w kwestii samego zegara/kalendarza, teraz bajery. W ramach bajarów mamy:

152 nie! nie bezpośrednio z gniazdka :]

- dwa alarmy, z możliwością odpalania przerwań, budzenia procka i wyprowadzenia sygnału alarmu na „zewnętrz” poprzez nóżkę
- układ cyklicznego budzenia procka, taki osobny licznik zliczający sygnał 1Hz (sygnał taktujący kalendarz) lub sygnał zegara RTCCLK¹⁵³ (sygnał taktujący blok RTC) i budzący procesor co ustalony okres czasu
- funkcję *timestamp*, która w reakcji na sygnał na wejściu odpowiedniego pinu (lub aktywację *tampera*) zapisuje aktualny stan zegara
- automatyczną „kompensację” roku przestępczego i miesięcy 30/31 dniowych

Procedura konfiguracji zegara:

- odblokować możliwość zapisu (bity PWR_CR_DBP, RTC_WPR)
- ustawić bit INIT w RTC_ISR (wejście w tryb konfiguracyjny)
- poczekać na ustawienie flagi RTC_ISR_INITF
- ustawić wartości preskalerów (zawsze muszą być dwa osobne zapisy, RTFM)
- ustawić wartości rejestrów czasu i daty
- skasować bit INIT

Inne uwagi i spostrzeżenia:

- wszystkie przerwania zegara RTC podciagnięte są pod linie EXTI
- za pomocą bitów RTC_CR_ADD1H i RTC_CR_SUB1H można łatwo przestawić czas między zimowym a letnim (dodać lub odjąć jedną godzinę), bez przeprowadzania całej procedury konfiguracyjnej zegara/kalendarza
- rejstry zegara są zabezpieczone przed zapisem:
 - bitem PWR_CR_DBP w bloku PWR
 - kluczem w RTC_WPR¹⁵⁴, aby odblokować zapis należy wpisać do tego rejestru wartości 0xCA i 0x53; zablokowanie następuje po wpisaniu jakiejkolwiek innej wartości lub resecie domeny baterijnej
- w celu sprawdzenia (np. po resecie) czy zegar jest skonfigurowany, można odczytać flagę RTC_ISR_INITS. Jeśli jest skasowana to znaczy, że zegar nie jest ustawiony. Flaga jest skasowana jeśli nastawa roku jest równa 0.

¹⁵³ dostępny jest osobny preskaler

¹⁵⁴ nie dotyczy rejestrów RTC_ISR[13:8], RTC_TAFCR, RTC_BKPxR

Zadanie domowe 9.7: samodzielne wymyślenie bardzo skomplikowanego zadania z wykorzystaniem RTC a następnie rozpracowanie i rozwiązanie problemu. Dla chętnych „na plusa”: podesłanie mi zadania z opracowaniem celem zamieszczenia w (ewentualnych) kolejnych wydaniach poradnika :}

Co warto zapamiętać z tego rozdziału?

- co chcesz ;)

9.7. Backup Registers i RTC (F334)

Śliczna biała zabaweczka ma 16 rejestrów podtrzymywanych baterijnie (*backup registers*). Ich obsługa nie różni się na oko niczym od tej zaprezentowanej w rozdziale dotyczącym F429. Podobnie ma się sprawa z zegarem RTC. Tzn. widzę, że w rejestrze konfiguracyjnym funkcji wykrywania sabotażu (*tamper*) jest coś innego (na oko jest więcej wejść *tamper*). I różnią się rejesty związanego z kalibracją zegara. Ale z grubsza na pewno działa to podobnie :] RTC jest nudne, więc na tym poprzestańmy. Doczytaj we własnym zakresie.

Co warto zapamiętać z tego rozdziału?

- za bardzo to chyba nie ma czego tu zapamiętywać

10. UKŁADY WATCHDOG („DUO CUM FACIUNT IDEM, NON EST IDEM”¹⁵⁵)

10.1. Watchdog niezależny IWDG

Parafrując definicję *konia* z pierwszej polskiej encyklopedii powszechniej: „*Watchdog jaki jest, każdy widzi*”. Watchdog niezależny¹⁵⁶ (*independent watchdog*) to prosty 12 bitowy dekrementator (*down counter*) taktowany z wewnętrznego oscylatora niskiej częstotliwości (LSI, coś między 30 a 60kHz, mało stabilne bydle). Jak IWDG zliczy do zera to resetuje procesor. Ot i cała filozofia.

Z układem IWDG łączą się następujące rejestryst:

- IWDG_RLR - wartość ładowana do licznika przy kasowaniu watchdoga (od tej wartości licznik zlicza w dół), po wpisaniu nowej wartości rozpoczyna się aktualizacja rejestru w domenie zegara RTC, o trwającej aktualizacji informuje bit IWDG_SR_RVU; rejestru nie należy modyfikować do czasu zakończenia trwającej aktualizacji (wyzerowanie się bitu IWDG_SR_RVU)
- IWDG_PR - nastawa preskalera licznika (możliwy podział przez 4, 8, 16, ..., 256), po wpisaniu nowej wartości rozpoczyna się aktualizacja rejestru w domenie zegara RTC, o trwającej aktualizacji informuje bit IWDG_SR_PVU; rejestru nie należy modyfikować do czasu zakończenia aktualizacji (wyzerowanie się bitu IWDG_SR_PVU)
- IWDG_SR - rejestr zawiera dwa bity które informują o trwającej właśnie aktualizacji wartości rejestrów IWDG_RLR i IWDG_PR (w czasie trwania aktualizacji nie należy tych rejestrów modyfikować ani odczytywać)
- IWDG_KR (*Key Register*) - to jest rejestr sterujący pracą układu IWDG, reaguje on tylko na magiczne, stałe wartości kluczowe:
 - 0xCCCC - uruchomienie licznika
 - 0xAAAA - skasowanie watchdoga (załadowanie rejestru licznika wartością z IWDG_RLR)
 - 0x5555 - odblokowanie możliwości zapisu do rejestrów IWDG_PR i IWDG_RLR, blokada jest ponownie aktywowana po wpisaniu do IWDG_KR jakiejkolwiek innej wartości (np. przy kasowaniu watchcata)

155 „Gdy dwóch robi to samo, to nie jest to samo.”

156 albo niezawisły :)

Ciekawostki na koniec:

- początkowa wartość rejestru przeładowania wynosi 0xFFFF, co odpowiada maksymalnemu okresowi zliczania
- w zależności od nastaw preskalera i wartości przeładowania, czas do resetu może wynosić od około 0,1ms do ponad 26s (wartości orientacyjne)
- raz włączonego watchdoga **nie da się wyłączyć aż do resetu mikrokontrolera**
- zachowanie watchdoga po zatrzymaniu rdzenia (przez debugger) zależy od bitu DBG_IWDG_STOP w rejestrze DBGMCU_CR (generalnie jak zatrzymujemy rdzeń, to nie będzie kasowania szczenięcia, więc jeśli watchdog nie zostanie zatrzymany to zresetuje mikrokontroler i zerwie połączenie z debuggerem)
- dla lubiących wzorki - czas do zadziałania piesa w funkcji częstotliwości LSI, nastawy preskalera, wartości rejestru przeładowania:

$$t_{timeout} = \frac{4 \cdot 2^{PR} \cdot RLR}{f_{LSI}} [s]$$

Zadanie domowe 10.1: IWDG ustawiony na 3s. Na sekundę przed zadziałaniem IWDG zapala się ostrzegawcza dioda (przypominająca o potrzebie przeładowania IWDG). Przeładowanie IWDG jest wywoływanie przyciskiem. Przycisk, ponadto, resetuje diodę przypominającą tak aby znowu zapaliła się na sekundę przed resetem mikrokontrolera. Opcja na piątkę: program na początku (po uruchomieniu) sprawdza przyczynę resetu i jeśli był wywołany przez IWDG to sygnalizuje to drugą diodą. Ha! Nader kompleksowy przykład mi się zmajstrował, nieprawdaż? No to sio do roboty! Cały czas obowiązuje nasza umowa o pracy samodzielnnej i minimum trzech dniach prób, pamiętasz!?

Przykładowe rozwiązanie (F429, diody na PG13 i PG14, przycisk na PA0):

```
1. volatile uint32_t reminder;
2.
3. int main(void){
4.
5.     RCC->AHB1ENR |= RCC_AHB1ENR_GPIOGEN | RCC_AHB1ENR_GPIOAEN;
6.     RCC->APB2ENR = RCC_APB2ENR_SYSCFGEN;
7.     __DSB();
8.
9.     gpio_pin_cfg(GPIOG, PG13, gpio_mode_out_PP_LS);
10.    gpio_pin_cfg(GPIOG, PG14, gpio_mode_out_PP_LS);
11.    gpio_pin_cfg(GPIOA, PA0, gpio_mode_in_floating);
12.
13.    if (RCC->CSR & RCC_CSR_WDGRSTF) {
14.        BB(RCC->CSR, RCC_CSR_RMVF) = 1;
15.        BB(GPIOG->ODR, PG14) = 1;
16.    }
17.
18.    BB(RCC->CSR, RCC_CSR_LSION) = 1;
19.    while ( BB(RCC->CSR, RCC_CSR_LSIIRDY) == 0 );
20.
21.    IWDG->KR = 0x5555;
22.    IWDG->PR = 4;
23.    IWDG->RLR = 1875;
24.    IWDG->KR = 0xaaaa;
25.    IWDG->KR = 0xcccc;
26.    __DSB();
27.
28.    while( BB(IWDG->SR, IWDG_SR_PVU) == 1 );
29.    IWDG->PR = 1;
30.    while( BB(IWDG->SR, IWDG_SR_RVU) == 1 );
31.    IWDG->RLR = 99;
32.
33.    SYSCFG->EXTICR[0] = SYSCFG_EXTICR1_EXTI0_PA;
34.    EXTI->RTSR = EXTI_RTSR_TR0;
35.    EXTI->IMR = EXTI_IMR_MR0;
36.
37.    NVIC_EnableIRQ(EXTI0_IRQn);
38.    SysTick_Config(160000);
39.
40.    while(1);
41. }
42.
43. void SysTick_Handler(void){
44.     reminder++;
45.     if (reminder == 200) BB(GPIOG->ODR, PG13) = 1;
46. }
47.
48. void EXTI0_IRQHandler(void) {
49.     if ( EXTI->PR & EXTI_PR_PR0 ) {
50.         EXTI->PR = EXTI_PR_PR0;
51.         IWDG->KR = 0xaaaa;
52.         BB(GPIOG->ODR, PG13) = 0;
53.         reminder = 0;
54.     }
55. }
```

Ogólna idea programu jest następująca: na początku po resetie sprawdzana jest flaga źródła resetu. Jeśli reset był wymuszony przez IWDG to zapalana jest dioda na PG14. Potem włączany jest układ IWDG, przerwanie od przycisku i SysTicka. SysTick inkrementuje zmienną *reminder* co 10ms. Jeśli jej wartość przekroczy 200 (czyli upłynął 2s) to zapalana jest dioda na PG13 (przypomnienie o konieczności skasowania hotdoga). W przerwaniu od przycisku następuje przeładowanie układu IWDG, zgaszenie diody przypominającej o watchdogu (PG13) i wyzerowanie zmiennej *reminder*. To teraz ciekawsze szczegóły:

5, 6) włączenie zegarów portów i bloku SYSCFG (przerwania zewnętrzne), zwróć uwagę, że IWDG nie wymaga włączenia zegara (tzn. wymaga, ale trochę inaczej → linia 18 kodu)

13) tu siedzi sprawdzanie flagi źródła resetu (wspomniałem o flagach mimochodem przy opisie wyjątku *reset*, w przypisie 88), szczegóły dotyczące bloku RCC zostaną omówione w rozdziale 17. Uwaga pułapka! Nie wiem czemu, ale w pliku nagłówkowym nie ma flagi układu IWDG takiej jak w RMie (IWDGRSTF), w pliku nagłówkowym ten bit jest nazwany inaczej: WDGRSTF... bywa.

14) wszystkie flagi dotyczące źródła resetu (w rejestrze RCC_CSR) są tylko do odczytu. Przypominam, że nie są one zerowane przy resecie mikrokontrolera. Czyli jeśli nie skasujemy takiej flagi (np. od watchdoga) to będzie ona ustawiona po kolejnym resecie, nawet jeśli będzie on wywołyany inną przyczyną (np. nóżką NRST). Skasowanie flag następuje po wykonaniu operacji zapisu do bitu RCC_CSR_RMVF.

Swoją drogą to jest chyba najdziwniejszy bit z jakim mieliśmy dotąd do czynienia. Zwróć uwagę na jego opis w dokumentacji: *rt_w*¹⁵⁷ - bit jest tylko do odczytu, a jakikolwiek zapis (zera lub jedynki, bez znaczenia) wywołuje jakieś zdarzenie (w przypadku bitu RMVF powoduje kasowanie flag źródła resetu) ale nie zmienia stanu samego bitu. Tak czy siak, wpisujemy jedynkę aby skasować flagi.

15) zapalamy diodę

18, 19) włączenie wewnętrznego źródła zegara niskiej częstotliwości (LSI) i oczekiwanie na jego rozbuchanie (szczegóły o źródłach sygnałów zegarowych do doczytania w mitycznym rozdziale 17)

21) zaczynamy konfigurować IWDG, wartość magiczna 0x5555 wpisana do rejestru IWDG_KR odblokowuje możliwość konfiguracji licznika (ustawienia preskalera i wartości przeładowania)

22, 23) ustawienie nastawy preskalera i wartości rejestru przeładowania. W RMie jest tabelka ułatwiająca dobrą preskalera (*Min/max IWDG timeout period at 32 kHz (LSI)*). Zwróć uwagę, że wartość wpisana do rejestru preskalera nie odpowiada stopniowi podziału częstotliwości. Założymy w przybliżeniu, że LSI ma 40kHz. Wartość PR = 4 odpowiada podziałowi częstotliwości przez 64. 40kHz przez 64 to 635Hz, jeden takt zegara trwa 1,6ms. Stąd 3s to będzie (3s/1,6ms) około 1875 taktów... lub jak ktoś woli to kilka akapitów temu był gotowy wzorek :)

24) przeładowanie licznika

25) uruchomienie licznika

26) profilaktycznie czekam na zakończenie operacji na pamięci (z powyższych linii) przed eksperymentem w liniach 28 - 31

28 - 31) zapisanie do rejestru IWDG_KR czegokolwiek innego niż 0x5555 powinno blokować możliwość zmiany rejestrów IWDG_RLR oraz IWDG_PR. Po operacjach z linii 24 i 25, blokada winna być aktywna. Sprawdzam to (w ramach edukacyjnej zabawy) poprzez wpisanie nowych

157 Read Only, Write Trigger

wartości rejestru preskalera i przeładowania (linie 29 i 31). Jeśli nowe wartości zadziałyają (mimo blokady), to IWDG będzie resetował mikrokontroler po około 20ms a nie 3s jak planowaliśmy... zdecydowanie zauważymy różnicę :)

Przed wykonaniem zapisu sprawdzam odpowiednie flagi w rejestrze IWDG_SR żeby się upewnić, że poprzedni zapis (z linii 22 i 23) się zakończył. Przy wcześniejszym zapisie (linie 22 i 23) nie sprawdzałem flag bo to był pierwszy zapis do tych rejestrów (IWDG_PR i IWDG_RLR), więc nie było żadnego „wcześniejszego” który mógłby się jeszcze nie zakończyć.

31 - 33) konfiguracja przerwania od przycisku

36) SysTick ustawiony na 10ms (F429 domyślnie działa na 16MHz)

41) przerwanie systemowe co 10ms, inkrementacja zmiennej i zapalenie diody po 2s

46) przerwanie zewnętrzne (przycisk)

49) przeładowanie licznika IWDG

Co warto zapamiętać z tego rozdziału?

- patrz rozdział 10.3

10.2. Watchdog okienkowy WWDG

Watchdog okienkowy również jest licznikiem zliczającym w dół i generującym resety. To co odróżnia go od zwykłego układu watchdoga to to, że przy watchdogu okienkowym przeładowanie licznika musi wystąpić w określonym przedziale czasowym (oknie). W zwykłym układzie watchdog liczyło się tylko to, aby przeładowanie nie nastąpiło zbyt późno. W układzie okienkowym skasowanie nie może nastąpić ani zbyt późno, ani zbyt wcześnie!

Z układem WWDG związane są następujące rejesty:

- WWDG_CR - rejestr kontrolny, zawiera bit włączający układ (WDGA) oraz zawartość licznika (pole bitowe T)
- WWDG_CFG - rejestr konfiguracyjny, zawiera bit włączający generowanie przerwań¹⁵⁸ (EWI), nastawę preskalera (bit WDGTB) oraz wartość wyznaczającą okno licznika (bita pola W)
- WWDG_SR - rejestr statusowy, zawiera tylko flagę przerwania

Żeby było śmieszniej działa to zgoła osobliwie. A dokładniej: licznik sobie zlicza w dół (bit T[6:0] w rejestrze WWDG_CR) i reset jest generowany w momencie wyzerowania szóstego

¹⁵⁸ generowanie przerwań można włączyć, ale nie można go już potem wyłączyć (oznaczenie *rs* przy opisie bitu w RM)

bitu pola T. Czyli gdy wartość z pól T[6:0] zmieni się z 0x40 na 0x3F. Czemu nie liczy do zera, tylko tak fikuśnie? Nie mam pojęcia. Trzeba pokochać i tyle. Podsumowując reset nastąpi gdy:

- wartość licznika (WWDG_CR_T) spadnie poniżej 0x40
- przeładowanie watchdoga nastąpi zbyt wcześnie, tj. gdy: WWDG_CR_T > WWDG_CFR_W

lub mówiąc inaczej¹⁵⁹: aby reset nie wystąpił przeładowanie licznika musi nastąpić, gdy:

$$0x40 < \text{WWDG_CR_T} < \text{WWDG_CFR_W}$$

Opóźnienie od przeładowania do wyjścia z okna czasowego (do resetu) wyraża się takim oto wzorem (lub bardzo podobnym)...:

$$t_{\text{WWDG_reset}} = T_{\text{PCLK1}} \cdot 4096 \cdot 2^{\text{WDGTB}[1:0]} \cdot (\text{T}[5:0] + 1)$$

Opóźnienie do wejścia w okno czasowe (od przeładowania do momentu, gdy będzie można już skasować licznik) wyraża się wzorem:

$$t_{\text{WWDG_window}} = T_{\text{PCLK1}} \cdot 4096 \cdot 2^{\text{WDGTB}[1:0]} \cdot (\text{T}[5:0] - \text{W}[5:0] + 1)$$

przykładowo jeśli:

- częstotliwość szyny APB1 wynosi 36MHz (wyjaśni się w rozdziale 17)
- nastawa preskalera (wartość z pola WDGTB, nie stopień podziału!) wynosi 0b10 = 2
- przeładowaliśmy rejestr WWDG_CR wartością 0xFF (bitы T[5:0] = 0x3F)
- wartość pola W[6:0] wynosi 0x65 (wartość bitów W[5:0] = 0x20)

to układ będzie można skasować po upływie minimum:

$$t_{\text{WWDG_window}} = \frac{1}{36e6} \cdot 4096 \cdot 2^2 \cdot (0x3F - 0x20 + 1) \approx \frac{0,52e6}{36e6} \approx 15 \text{ ms}$$

tudzież reset nastąpi po:

$$t_{\text{WWDG_reset}} = \frac{1}{36e6} \cdot 4096 \cdot 2^2 \cdot (0x3F + 1) \approx \frac{1,05e6}{36e6} \approx 30 \text{ ms}$$

¹⁵⁹ tak bardziej od dupy strony...

Pozostałe ciekawostki:

- licznik zlicza nawet gdy watchdog jest wyłączony - nie można więc nic założyć w kwestii początkowej wartości rejestru licznika (przy włączaniu)
 - wpisując nową wartość do rejestru WWDG_CR należy **zawsze** się upewnić, że szósty bit pola T[6:0] jest ustawiony... inaczej mamy gwarantowany natychmiastowy reset
 - skasowanie licznika jest wykonywane poprzez zapis nowej wartości do rejestru WWDG_CR, nowa wartość musi mieć ustawiony szósty bit pola T i (opcjonalnie) ustawiony bit odpowiedzialny za włączenie watchdoga (czyli powinna zawierać się w przedziale 0xC0 - 0xFF)
 - zachowanie watchdoga po zatrzymaniu rdzenia (przed debugger) zależy od bitu DBG_WWDG_STOP w rejestrze DBGMCU_CR
 - przerwanie EWI generowane jest tuż przed resetem, w momencie, gdy wartość licznika wyniesie 0x40 (można w nim oczywiście skasować watchdog aby zyskać trochę czasu)
-

Swoją drogą, tak mnie teraz uderzyła pewna myśl. W STMach hotdogi są zwyczajnymi układami peryferyjnymi mikrokontrolera. Zwróciłeś uwagę jak to było w AVRach? Tam przecież był osobny rozkaz asm do kasowania watchcata (*wdr*). Ciekawe, prawda?

Zadanie domowe 10.2: po uruchomieniu programu dioda ma mignąć 4 razy jeśli poprzedni reset był spowodowany układem WWDG; w przeciwnym wypadku 2 razy. Następnie ma być uruchamiany WWDG z czasami dobranymi tak aby wejście w „okno” następowało po 2s od przeładowania, zaś wyjście z okna (reset) po 4s od przeładowania. Program powinien również sterować dwiema diodami. Jedna ma się palić dopóki WWDG nie wejdzie w okno czasowe. Druga dioda ma się zapalać gdy jesteśmy w oknie czasowym (i można bezpiecznie przeładować WWDG). Przeładowanie WWDG przyciskiem. Na dokładkę proponuję jeszcze przerwanie od WWDG - niech zapala obie diody.

Przykładowe rozwiążanie (F429, diody na PG13 i PG14, przycisk na PA0):

```
1. volatile uint32_t reminder, delay, flaga;
2.
3. void delay_10ms(uint32_t cnt){
4.     delay = cnt;
5.     while(delay);
6. }
7.
8. void blink(void){
9.     BB(GPIOG->ODR, PG14) = 1;
10.    delay_10ms(20);
11.    BB(GPIOG->ODR, PG14) = 0;
12.    delay_10ms(20);
13. }
14.
15. int main(void){
16.
17.     RCC->CFGGR = RCC_CFGGR_PPREG1_DIV2 | RCC_CFGGR_HPRE_DIV16;
18.     RCC->AHB1ENR |= RCC_AHB1ENR_GPIOGEN | RCC_AHB1ENR_GPIOAEN;
19.     RCC->APB2ENR = RCC_APB2ENR_SYSCFGEN;
20.     RCC->APB1ENR = RCC_APB1ENR_WWDGEN;
21.     __DSB();
22.
23.     SysTick_Config(10000);
24.
25.     gpio_pin_cfg(GPIOG, PG13, gpio_mode_out_PP_LS);
26.     gpio_pin_cfg(GPIOG, PG14, gpio_mode_out_PP_LS);
27.     gpio_pin_cfg(GPIOA, PA0, gpio_mode_in_floating);
28.
29.     if (RCC->CSR & RCC_CSR_WWDGRSTF) {
30.         blink();
31.         blink();
32.         blink();
33.         blink();
34.     } else {
35.         blink();
36.         blink();
37.     }
38.     BB(RCC->CSR, RCC_CSR_RMVF) = 1;
39.
40.     delay_10ms(100);
41.     flaga = 1;
42.
43.     WWDG->CR = WWDG_CR_WDGA | WWDG_CR_T6 | 60;
44.     WWDG->CFR = WWDG_CFR_EWI | 3<<7 | WWDG_CFR_W6 | 30;
45.     WWDG->SR &= ~WWDG_SR_EWIF;
46.
47.     BB(GPIOG->ODR, PG14) = 1;
48.
49.     SYSCFG->EXTICR[0] = SYSCFG_EXTICR1 EXTI0_PA;
50.     EXTI->RTSR = EXTI_RTSR_TR0;
51.     EXTI->IMR = EXTI_IMR_MR0;
52.
53.     NVIC_ClearPendingIRQ(WWDG_IRQn);
54.     NVIC_EnableIRQ(WWDG_IRQn);
55.     NVIC_EnableIRQ(EXTI0_IRQn);
56.
57.     while(1);
58. }
59.
60. void SysTick_Handler(void){
61.     if (flaga) reminder++;
62.     if (reminder == 200) {
63.         BB(GPIOG->ODR, PG13) = 1;
64.         BB(GPIOG->ODR, PG14) = 0;
65.     }
66.
67.     if(delay) delay--;
68. }
69.
70. void EXTI0_IRQHandler(void) {
71.     if (EXTI->PR & EXTI_PR_PR0) {
72.         EXTI->PR = EXTI_PR_PR0;
73.         WWDG->CR = WWDG_CR_WDGA | WWDG_CR_T6 | 60;
74.         reminder = 0;
```

```

75.     BB(GPIOG->ODR, PG13) = 0;
76.     BB(GPIOG->ODR, PG14) = 1;
77. }
78. }
79.
80. void WWDG_IRQHandler(void) {
81.     WWDG->SR &= ~WWDG_SR_EWIF;
82.     WWDG->CR = WWDG_CR_WDGA | WWDG_CR_T6 | 5;
83.     BB(GPIOG->ODR, PG13) = 1;
84.     BB(GPIOG->ODR, PG14) = 1;
85.     while(1);
86. }

```

Troszkę za rozbudowany wyszedł ten przykład... ale za to jakiś kompleksowy. Nie myśl, że WWDG jest jakiś skomplikowany, 75% kodu to realizacja tych wszystkich migania, diodek i innych fanaberii które sobie wymyśliłem :)

17) aby uzyskać takie długie (rzędu sekund) czasy w układzie WWDG musiałem troszkę obniżyć częstotliwość pracy mikrokontrolera... ale rozdział o zegarach (17) jest dopiero przed nami, więc głupio wyszło... także ten... generalnie SysTick będzie chodził teraz na 1MHz a WWDG będzie taktowany z częstotliwością 500kHz, obiecuję że wszystko się później wyjaśni - na razie przyjmij na wiarę i zapomnij o tej linijce!

20) układu IWDG nie trzeba był włączać (bo on przecież taki niezależny ma być...), WWDG natomiast jak najbardziej się włącza

23) przerwania systemowe co 10ms (przypominam, że przez czary-mary z linijki 17, SysTick chodzi na 1MHz)

29 - 38) sprawdzenie flagi resetu, jeśli był wywołany przez WWDG to cztery mignięcia ledem, w przeciwnym wypadku dwa mignięcia, na koniec skasowanie flag (funkcja *delay* działa w oparciu o przerwania SysTicka)

40) odczekanie chwili po tych kodach migowych, żeby operator się przygotował :)

41) zmienna *flaga* jest sprawdzana w przerwaniu SysTicka, od momentu kiedy zostaje ustawiona (czyli od teraz) w przerwaniu SysTicka zliczany jest czas (zmienna *reminder*) do wejścia WWDG w „okno czasowe” (wtedy zostanie zapalona dioda że już można przeładować WWDG)

43) włączenie watchdoga i ustawienie czasu do resetu (wartość 60 - na podstawie wcześniej podanego wzorku) oraz szóstego bitu pola *T*

44) włączenie przerwania, ustawienie preskalera WWDG (3), konfiguracja okna czasowego (30 - na podstawie wcześniej podanej formułki). Uwaga! Przy wielkości okna czasowego, również należy zawsze ustawić szósty bit pola *W*!

45) wspomniałem wcześniej o tym, że licznik WWDG działa cały czas (wspomniałem?), nawet przed jego konfiguracją i ustawieniem bitu WWDG_CR_WDGA. Powoduje to, że po uruchomieniu mikrokontrolera, licznik zdążył już kilka razy zliczyć do „zera¹⁶⁰”. W związku z tym flaga

160 w cudzysłowie bo licznik WWDG generalnie nie zlicza do zera...

przerwania na bank jest ustawiona. Jeżeli beztrosko włączymy przerwanie w NVICu to zostanie ono od razu obsłużone! A my tego nie chcemy. Toteż należy wykonać dwa kroki:

- skasować flagę przerwania w układzie peryferyjnym - linijka 45
- skasować „oczekiwanie” przerwania w NVICu - linijka 53

Dopiero po tym można je bezpiecznie włączyć (linijka 54) w kontrolerze NVIC.

Uwaga! Od momentu skasowania flagi w peryferialu (linijka 45) musi minąć trochę czasu zanim będzie można skasować *pending* w NVICu. Chodzi o to, że układ peryferyjny potrzebuje chwili na zdobycie zgłoszenia przerwania (szczególnie, że jest taktowany z niższą częstotliwością niż NVIC i rdzeń - bo linijka 17...). W pierwszej wersji tego programu kasowanie flagi i pendingu miałem „tuż po sobie”. Efekt był taki, że przerwanie uruchamiało się natychmiast po włączeniu, bo peryferial nie wyrabiał się ze zdobyciem zgłoszenia przerwania i NVIC łapał nowy „pending” od razu po skasowaniu. Oczywiście jak realizowałem program krokowo debuggerem, to skubaniec zdążył zdobyć zgłoszenie przerwania i wszystko było ok... Pomogło dopiero „rozsuniecie” tych operacji w programie :) (ew. można dodać jakiś zapychacz typu instrukcja barierowa)

73) przeładowanie WWDG w przerwaniu od przycisku. WWDG nie ma niestety wygodnego sposobu przeładowywania, za każdym razem trzeba od nowa wpisywać całą zawartość rejestru WWDG_CR.

80 - 86) przerwanie od WWDG. Najpierw kasuję flagę przerwania (jak zawsze), potem przeładowuję WWDG żeby zyskać trochę czasu, zapalam obie diody i wchodzę w nieskończoną pętlę DUŚ¹⁶¹. Zwróć uwagę na to że w tym wypadku, przy przeładowywaniu, zapodaję mniejszą wartość rejestru licznika. Cały ten cyrk w tym przerwaniu ma na celu drobne odroczenie resetu mikrokontrolera, tak aby dało się zauważać zapalenie obu diod.

Co warto zapamiętać z tego rozdziału?

- patrz rozdział 10.3

10.3. Porównanie układów watchdog

¹⁶¹ Do Usianej Śmierci

Mnogość narzędzi, zawsze rodzi pytanie o to, którego użyć. Spróbujmy porównać oba układy nadzorcze:

Tabela 10.1 Porównanie układów *watchdog*

Właściwość	IWDG	WWDG
reset występuje jeśli licznik zostanie skasowany	<i>zbyt późno</i>	<i>zbyt późno lub zbyt wcześnie</i>
źródło taktowania	<i>wewnętrzne (niezależne) - LSI</i>	<i>szyna APB1</i>
pewność działania	<i>duża (niezależne źródło zegarowe)</i>	<i>mała (brak niezależnego źródła taktowania)</i>
dokładność odmierzenego czasu	<i>niska (mała stabilność źródła)</i>	<i>duża (stabilne źródło sygnału zegarowego)</i>
możliwość wymuszenia sprzętowego włączania układu w czasie POR ¹⁶²	<i>tak (za pomocą Option Bytes)</i>	<i>nie</i>
działanie w trybach uśpienia (stop i standby)	<i>tak</i>	<i>nie</i>
możliwość generowania przerwania tuż przed resetem	<i>nie</i>	<i>tak</i>

Co warto zapamiętać z tych rozdziałów?

- tabelkę 10.1 :)
- wiedzieć, gdzie znaleźć wzory opisujące działanie liczników nadzorujących (w dwóch poprzednich podrozdziałach)

10.4. Układy *watchdog* w mikrokontrolerze F334 (F334)

Ten mikrokontroler wszystko musi mieć inaczej niż dwa pozostałe :) Do watchdoga niezależnego (IWDG) dodana została opcja „okna”, działająca na tej samej zasadzie jak w watchdogu okienkowym (WWDG). Za konfigurację okna odpowiada rejestr IWDG_WINR. Licznik można bezpiecznie (bez wywoływania resetu mikrokontrolera) przeładować tylko wtedy, gdy aktualna wartość licznika jest większa od zera i mniejsza od wartości z rejestru okienkowego (IWDG_WINR).

Korzystanie z opcji okienkowej nie jest konieczne. Domyślna wartość rejestru WINR jest równa maksymalnej wartości początkowej licznika. Czyli wartość licznika nigdy nie jest większa od wartości WINR, więc funkcja okienkowa jest „nieaktywna”.

¹⁶² Power on Reset

Drugiemu układowi nadzorującemu (WWDG) nic się, na szczęście, nie zmieniło. Tyle w temacie.

Co warto zapamiętać z tego rozdziału?

- w F334 do licznika IWDG dodano opcję „okna”

11. RESET, ZASILANIE I TRYBY OSZCZĘDZANIA ENERGII („*DIFFICILIS IN OTIO QUIES*”¹⁶³)

11.1. Reset

Taki temat z zupełnie innej beczki, żeby odpocząć od liczników. Mamy trzy rodzaje resetów:

- System Reset
- Power Reset
- Backup Domain Reset

System reset powoduje, że wszystkie rejesty (prawie, patrz niżej) przyjmują swoje wartości domyślne a procesor „startuje od zera” (odczytuje z tablicy wektorów adres stosu i odpala wyjątek *reset*). Ten typ resetu występuje w następujących sytuacjach:

- przy wymuszeniu sprzętowym (*external reset*) - nóżka NRST
- przy wymuszeniu programowym (*software reset*) - patrz bit SYSRESETREQ w SCB_AIRCR lub funkcja *NVIC_SystemReset()*
- jeśli zadziała któryś z układów watchdog (WWDG, IWDG)
- w przypadku próby uśpienia mikrokontrolera, jeśli skasowany jest odpowiedni bit konfiguracyjny (patrz bity: nRST_STDBY, nRST_STOP w dokumencie *STM32F10xxx Flash programming manual*) - występuje wtedy *Low Power Management Reset*¹⁶⁴

Zawsze musi być jakieś ale... tu też jest. Reset systemowy nie przywraca domyślnych wartości następujących rejestrów:

- RCC_CSR - to dosyć logiczne, bo ten rejestr pozwala ustalić źródło resetu, więc musi być zachowany
- rejestrów domeny baterijnej (wszelkie RTC, backup, tamper, ...)

Warto zwrócić uwagę na to, że nóżka *NRST* mikrokontrolera, nie działa tylko jako *wejście*. Każde źródło resetu systemowego (np. reset wywołany programowo) powoduje pojawienie się na niej stanu niskiego. Z tego względu podłączenie nóżki na stałe do *VCC* (co było możliwe w AVR), jest kiepskim pomysłem. Wyprowadzenie *NRST* można wykorzystać jako wyjście sygnału resetu

¹⁶³ „Bezczynność nie daje odpoczynku.”

¹⁶⁴ nie bardzo potrafię wyobrazić sobie sens stosowania tego mechanizmu... a Ty?

dla innych układów cyfrowych w systemie. Odsyłam do schematu *Simplified diagram of the reset circuit* w RMie.

Power reset działa tak samo jak *system reset*. Wywoływany jest przy:

- zadziałaniu układu monitorującego napięcie zasilania mikrokontrolera (POR, PDR lub BOR¹⁶⁵)
- wyjściu z trybu uśpienia Standby

Backup domain reset jest jedynym, który resetuje wszystkie rejestrów domeny backup. Wywoływany jest:

- programowo - bit BDRST w rejestrze RCC_BDCR
- po podaniu zasilania na V_{DD} lub V_{BAT}, jeśli wcześniej oba napięcia były nieobecne

Ustalenie źródła resetu jest możliwe dzięki flagom w rejestrze RCC_CSR. Korzystaliśmy już z tego mechanizmu przy okazji omawiania układów liczników nadzorujących watchdog (zadanie 10.1 oraz 10.2). Do dyspozycji mamy następujące flagi:

- *PWR_CSR_SBF* wskazuje, że mikrokontroler został wybudzony z trybu uśpienia *standby*
- *RCC_CSR_LPWRSTF* wskazuje na *Low Power Management Reset*
- *RCC_CSR_WWDGRSTF* wskazuje na reset wywołany układem okienkowego watchdoga
- *RCC_CSR_IWDGRSTF*¹⁶⁶ wskazuje na reset wywołany układem zwykłego (niezależnego) watchdoga
- *RCC_CSR_SFTRSTF* wskazuje na *Software Reset*
- *RCC_CSR_PORRSTF* wskazuje na reset wywołany układem POR lub PDR
- *RCC_CSR_PINRSTF* wskazuje na reset zewnętrzny (nóżka NRST)
- *RCC_CSR_BORRSTF* (nie dotyczy F103) wskazuje na reset wywołany układem BOR, POR lub PDR

Uwaga! Jedno źródło resetu może spowodować ustalenie kilku flag jednocześnie. No i flagi nie są kasowane przy resecie. Więc jeśli „na bieżąco” nie będziemy ich kasować to zidentyfikowanie źródła resetu nie będzie możliwe. Flagi kasowane są poprzez bit RCC_CSR_RMVF. Teraz żeby było śmieszniej w RM mikrokontrolera F103 flagi oznaczone są

¹⁶⁵ układ BOR nie występuje w mikrokontrolerze F103

¹⁶⁶ przypominam, że w pliku nagłówkowym ten bit nazywa się inaczej: RCC_CSR_WDGRSTF (nie wiem czemu)

jako *rw* (read/write). Co w takim razie robi ten *write* skoro flagi kasuje się innym bitem? W F429 już jest logicznie: flagi są tylko do odczytu.

Co warto zapamiętać z tego rozdziału?

- źródło resetu mikrokontrolera można odczytać z flag rejestru RCC_CSR
- funkcja *NVIC_SystemReset()* pozwala wymusić programowy reset mikrokontrolera
- domena baterystyczna nie jest zerowana przy resetie
- każde źródło *resetu systemowego*, powoduje ściągnięcie nóżki *NRST* do masy

11.2. Zasilanie (F103)

Kilka wybranych ciekawostek, żeby mieć ogólne pojęcie o temacie. Mikrokontroler zasilany może być napięciem z przedziału od 2 do 3,6V. Maksymalny pobór prądu wynosi coś koło 100mA. Dodatkowym źródłem zasilania jest bateria (1,8–3,6V). Zasilanie baterystyczne podtrzymuje pracę RTC i pamięć BKP. Jeżeli nie korzysta się z podtrzymywania baterystycznego to wskazane jest połączenie nóżki V_{BAT} z V_{DD} . Jest możliwe, że wyprowadzenie V_{BAT} zacznie wypluwać prąd. Jeżeli źródło (bateria) nie może go przyjąć, to zaleca się zabezpieczyć je diodą. Pobór prądu z baterii nie przekracza 1,5 μ A.

Nóżka V_{DDA} to osobne zasilanie części analogowej (V_{SSA} – masa analogowa). Jeśli w mikrokontrolerze występuje nóżka V_{ref^-} (ujemny biegum napięcia odniesienia dla ADC i DAC) to należy podłączyć ją do masy analogowej (V_{SSA}). Nie używane V_{SSA} i V_{DDA} należy podłączyć odpowiednio do masy i zasilania części cyfrowej. Nóżki $V_{ref^{+/-}}$ (plus i minus napięcia odniesienia) są dostępne tylko w obudowach 100 i 144 pinowych. W obudowie 64 pin są na stałe podpięte do zasilania części analogowej. Napięcie odniesienia dla bloków analogowych musi zawierać się w przedziale od 2,4V do V_{DDA} . Pobór prądu z wejścia V_{ref^+} to niecałe 200 μ A.

Mikrokontroler posiada wbudowany układ kontroli napięcia zasilającego, który odpowiada za utrzymanie układu w stanie resetu jeśli napięcie jest zbyt niskie (POR i PDR). Na jego pracę nie mamy wpływu. Szczegóły można sobie doczytać w datasheetcie. Ponadto jest na pokładzie PVD (*programmable voltage detector*), który pozwala porównywać V_{DD} z progiem ustalonym programowo. Event (zdarzenie) komparatora PVD jest sprzętowo połączony z przerwaniem EXTI16, czyli może np. odpalić przerwanie jeśli napięcie przekroczy zadany próg. Szczegóły w dokumentacji.

Co warto zapamiętać z tego rozdziału?

- a bo ja wiem...

11.3. Zasilanie (F429)

Tak, żeby mieć ogólne pojęcie o temacie. Mikrokontroler zasilany może być napięciem z przedziału 1,7V do 3,6V. Przy czym przy niższych napięciach wprowadzone są dodatkowe obostrzenia, np. zmniejszeniu ulega maksymalna częstotliwość pracy przetwornika ADC, częstotliwości taktowania szyn, pracy pamięci Flash (szczegóły w datasheet¹⁶⁷). Maksymalny pobór prądu wynosi coś koło 150mA. Dodatkowym źródłem zasilania jest bateria (1,65 - 3,6V). Zasilanie bateryjne podtrzymuje pracę RTC i pamięć BKP. Maksymalny pobór prądu z baterii, nie przekracza 10µA. Reszta opisu bez zmian w stosunku do STM32F103.

Nowością jest to, że oprócz układów (takich jak w F103) POR, PDR oraz PVD, STM32F429 posiada programowalny układ BOR. Układ BOR utrzymuje mikrokontroler w stanie resetu dopóki napięcie nie wzrośnie powyżej ustawionego progu. Różnica między układem BOR a POR/PDR polega właśnie na możliwości programowania progu zadziałania. Włączenie i konfiguracja układu jest możliwe poprzez *Option Bytes* (patrz rozdział 16.4).

Co warto zapamiętać z tego rozdziału?

- to samo co z poprzedniego podrozdziału

11.4. Zasilanie (F334)

Mikrokontroler może być zasilany napięciem z przedziału 2 - 3,6V. Maksymalny pobór prądu wynosi około 150mA. Typowy jest o wiele mniejszy :) Bateria podtrzymująca zasilanie układów RTC, BKP powinna mieć napięcie z zakresu 1,65 - 3,6V. Pobór prądu z baterii to około 1,5µA. Układ F334 nie ma wyprowadzonych, osobno, pinów związanych z napięciem odniesienia części analogowej (V_{ref+} , V_{ref-}). Napięcie odniesienia jest równe napięciu zasilania części analogowej (2,4 - 3,6V). Za monitorowanie zasilania odpowiadają układy POR, PDR (może zostać wyłączony) oraz programowalny PVD. Reszta opisu bez zmian w stosunku do poprzedniego rozdziału.

11.5. Debugowanie a tryby uśpienia

Pewien problem pojawia się przy próbie połączenia debugowania i trybów obniżonego poboru mocy (uśpienia). Przekonałem się o tym dość boleśnie przy próbie uruchomienia gotowego

¹⁶⁷ tabela *Limitations depending on the operating power supply range*

przykładu (chyba od ST) wykorzystującego jeden z tych trybów. Procesor w trybie uśpienia, zrywa połączenie z debuggerem! Miałem przez to problem z wgraniem innego programu, gdyż ten przykładowy kod od razu usypiał procka na starcie. Tzn. wtedy myślałem, że to chodzi o zrywanie połączenia ze względu na uśpienie, teraz już nie jestem tego taki pewien - patrz uwaga pod kolejnym akapitem.

Przypuszczałem, że zrywanie połączenia wynikało z tego, że tryb uśpienia wyłącza zegar dla układów peryferyjnych. Podpadał pod to również jakiś blok komunikujący się z debuggerem. Jest na to na szczęście rada :) Ustawienie bitów DBGMCU_CR_DBG_SLEEP, DBGMCU_CR_DBG_STOP i DBGMCU_CR_DBG_STANDBY powinno spowodować utrzymanie połączenia. Powodują one, że w trybach *sleep*, *stop* i *standby* jakiś tam sygnał zegarowy jest podtrzymywany i komunikacja nie ulega zerwaniu. Żeby nie było tak wesoło, w erracie jest kilka informacji związanych z powyższym:

- rejestr DBGMCU_CR nie jest dostępny z poziomu zwykłego programu, dobrać się do niego można tylko poprzez debugger (dotyczy tylko STM32F103)
- jeżeli będzie ustawiony bit DBG_STOP, to przerwania od SysTicka będą budzić mikrokontroler mimo że nie powinny
- jeżeli będzie ustawiony bit DBG_STOP i uśpienie zostanie wywołane instrukcją *wfe* to procesor może zgubić jedną instrukcję za *wfe!* zalecany *nop* tuż po *wfe*

Jako ciekawostkę powiem że OpenOCD ma zaszyte, w pliku konfiguracyjnym, ustawianie powyższych bitów.

Uwaga (dotyczy - prawdopodobnie - tylko oprogramowania OpenOCD i F103)! Jeśli program wykorzystujący tryby uśpienia zostanie uruchomiony bez podłączonego debuggera (tak, żeby się w pełni uśpił) a potem spróbujemy się podłączyć z JTAGiem, to coś się chrzani! Nie jest wówczas możliwe ponowne nawiązanie połączenia nawet jeśli się uruchomi firmowy bootloader (patrz rozdział 16.2). Wygląda to tak, jakby coś w mikrokontrolerze się blokowało. Pomaga dopiero całkowite odłączenie zasilania mikrokontrolera i uruchomienie „na czysto” w trybie bootloadera. Także w razie czego proszę się nie bać, procesor działa tylko wymaga odrobiny czułości. Gdzieś w Internecie znalazłem kiedyś informację z opisem tego problemu i podobnymi wnioskami (czyli nie tylko ja tak mam), ale jak na złość nie mogę się teraz dokopać do tego...

Co warto zapamiętać z tego rozdziału?

- uśpienie mikrokontrolera „domyślnie” zerwie połączenie z debuggerem

- za pomocą rejestru DBGMCU_CR można utrzymać połączenie po uśpieniu, jednak jest to okupione kilkoma błędami
- jeżeli po uśpieniu mikrokontrolera wystąpi problem z nawiązaniem połączenia to może pomóc całkowite wyłączenie zasilania

11.6. Tryby obniżonego poboru mocy (F103)

Po resecie mikrokontroler pracuje w trybie *run*¹⁶⁸. STMik ma trzy stopnie obniżonego poboru mocy:

- *Sleep Mode*: wyłączony zegar CPU, cała reszta mikrokontrolera pracuje (szybkie wybudzenie)
- *Stop Mode*: wszystkie zegary peryferii i CPU wyłączone
- *Standby Mode*: wyłącza się wewnętrzny regulator 1,8V; utrata zawartości SRAMu i rejestrów konfiguracyjnych (przypominam: wybudzeniu z trybu *standby* towarzyszy *power reset* mikrokontrolera, patrz rozdział 11.1)

Ponadto energo-żarłoczność można zmniejszyć obniżając prędkości taktowania gdzie tylko się da oraz wyłączając zegary nieużywanych bloków. Nauczymy się tego wszystkie w rozdziale 17.

Tryb *sleep mode* polega jedynie na zatrzymaniu zegara rdzenia. Wszystkie peryferia mikrokontrolera pracują normalnie. Porty I/O zachowują swój stan na czas uśpienia. Wejście do trybu jest możliwe poprzez instrukcje *wfi* (*wait for interrupt*) oraz *wfe* (*wait for event*). W CMSIS są dostępne funkcje wywołujące te rozkazy. Zachowanie procesora po ich wywołaniu zależne jest od bitu SLEEPONEXIT w rejestrze SCB_SCR. Jeśli bit jest skasowany to procesor od razu zasypia, jeśli bit jest ustalony to procesor zasypia gdy tylko opuści ISR wyjątku o najniższym priorytecie, czyli gdy powróci do *thread mode* lub mówiąc inaczej: gdy skończy z przerwaniami i wróci do *main*, było o tym w rozdziale 5.1

Opuszczenie *sleep mode* wywołanego przez *wfi* następuje po zgłoszeniu przerwania przez jakieś peryferial. Jeśli zaśnięcie było wywołane instrukcją *wfe* to procesor budzi się z okazji *wakeup-event*, czyli:

- jeśli pojawi się przerwanie włączone w peryferialu. Nie musi być włączone w NVICu pod warunkiem że jest ustalony bit SEVONPEND w SCB_SCR. Po wybudzeniu należy ręcznie wyczyścić flagę przerwania w peryferialu i „pendingu” w NVICu

168 Run Forrest Run!

- jeśli pojawi się zewnętrzne przerwanie (EXTI) i będzie ono skonfigurowane w trybie *Event*¹⁶⁹, po wybudzeniu nie trzeba czyścić flag przerwań bo nie są ustawiane przy konfiguracji EXTI Event

Tryb *stop mode* opiera się na trybie *Deep Sleep* rdzenia Cortex oraz wyłączeniu zegara dla peryferiów mikrokontrolera. Dodatkowo można skonfigurować wewnętrzny stabilizator 1,8V tak, aby przeszedł w tryb uśpienia *low-power* (bit PWR_CR_LPDS). Uśpienie regulatora zmniejsza pobór prądu kosztem wydłużenia czasu wybudzania. SRAM jest zachowany. Stany pinów I/O też. W *stop mode* mogą pracować jedynie następujące bloki:

- *RTC* - w zależności od stanu bitu RCC_BDCR_RTCEN
- *IDWG* - (*watchdog niezawisły*) gdyż raz włączonego nie da się wyłączyć!
- *LSI* - (wewnętrzny oscylator małej częstotliwości) w zależności od stanu bitu LSION w rejestrze RCC_CSR (swoją drogą - co się stanie z IWDG jeśli wyłączę LSI?)
- *LSE* - (zewnętrzny oscylator małej częstotliwości) w zależności od stanu bitu LSEON w rejestrze RCC_BDCR

Ponadto układy ADC i DAC mogą zużywać energię jeśli nie zostały ręcznie wyłączone (poprzez bity ADC_CR2_ADON i DAC_CR_ENx) przed uśpieniem procesora. Przy opuszczaniu trybu *stop mode* zegar systemowy zostaje przestawiony automatycznie na wewnętrzny (HSI)!

Uśpienie procesora uzyskuje się za pomocą tych samych instrukcji co poprzednio (*wfi* lub *wfe*) przy czym:

- musi być ustawiony bit SCB_SCR_SLEEPDEEP
- musi być skasowany bit PWR_CR_PDDS
- muszą być skasowane wszystkie flagi oczekujących przerwań peryferiów, w przeciwnym wypadku instrukcja uśpienia zostanie zignorowana!

Opuszczenie trybu jest możliwe zależnie od sposobu wejścia:

- *wfi* – wybudzenie następuje po jakimkolwiek przerwaniu EXTI włączonym w NVICu
- *wfe* – tylko poprzez EXTI skonfigurowane jako *Event* (przypominam, że pod EXTI łapie się też np. *Alarm RTC*)

¹⁶⁹ patrz rejestr EXTI_EMR

Tryb **standby mode** to tryb najgłębszego uśpienia. Tak mi się jakoś nieodparcie kojarzy z napisem „*Można teraz bezpiecznie wyłączyć komputer.*” Wyłączone zostaje wszystko łącznie z wewnętrznym regulatorem 1,8V. Tracone są dane z rejestrów konfiguracyjnych i SRAMu (poza pamięcią backup i RCC_CSR). W tym trybie działać mogą układy:

- IWDG
- RTC
- LSI
- LSE

Uśpienie mikrokontrolera w tym trybie następuje (jak zawsze) poprzez rozkazy *wfi* i *wfe* jeśli:

- ustawiony jest bit SCB_SCR_SLEEPDEEP
- ustawiony jest bit PWR_CR_PDDS
- skasowany jest bit PWR_CSR_WUF

Wyjście z tego trybu uśpienia jest możliwe poprzez:

- *Wakeup Pin* (rosnące zbocze na pinie WKUP)
- Alarm RTC (EXTI17)
- reset zewnętrzny (nóżka NRST)
- reset wymuszony przez IWDG

Po wyjściu program zachowuje się tak jakby dopiero co został uruchomiony (power reset)!

W tym trybie uśpienia wszystkie piny przechodzą w stan HiZ (stan wysokiej impedancji) poza:

- resetem
- *tamper pinem* (jeśli jest włączona funkcja tamper lub wyjście zegarowe układu RTC)
- *WKUP pinem* (jeśli funkcja jest włączony)

Podsumujmy tryby oszczędzania energii w formie macierzowej:

Tabela 11.1 Podsumowanie trybów uśpienia

Tryb uśpienia		Sleep Mode	Stop Mode	Standby Mode
Konfiguracja bitów przy usypianiu		SLEEPDEEP = 0 SLEEPONEXIT = 0/1	SLEEPDEEP = 1 PDDS = 0 (wszystkie flagi przerwań muszą być skasowane)	SLEEDPEED = 1 PDDS = 1 WUF = 0
Zachowanie wybranych układów w stanie uśpienia	GPIO	zachowują stan	zachowują stan	HiZ ¹⁷⁰
	opóźnienie wybudzenia	1,8µs	LPDS = 1: 3,6µs LPDS = 0: 5,4µs	50µs
	regulator 1,8V	pracuje	zależnie od LPDS w PWR_CR	wyłączony
	peryferia	pracują	wyłączone (z wyjątkiem IWDG, RTC, LSI, LSE; energię pobierać może także ADC i DAC)	
Pobudka	wfi	przerwanie włączone w NVIcu	przerwanie od EXTI	<ul style="list-style-type: none"> -reset zewnętrzny -reset IWDG -RTC alarm (EXTI17) -WKUP pin (zbocze rosnące)
	wfe	przerwanie + SEVONPEND lub zdarzenie od EXTI	zdarzenie od EXTI	
Orientacyjny pobór prądu (85°C; 3,3V; zewnętrzny oscylator):		peryferia wyłączone: 1,1mA @1MHz, 6,4mA @72MHz peryferia włączone: 1,5mA @8MHz, 29,5mA@72MHz	regulator: - włączony: 35µA - w trybie low power: 25µA	watchdog: - wyłączone: 2µA - włączony: 4µA
Uwagi:		-	po wybudzeniu jako źródło zegara zostaje ustawiony HSI	utrata zawartości SRAM i rejestrów (poza BKP i RCC_SCR), Flaga PWR_CSR_SBF

Zadanie domowe 11.1: napisać prosty program z dwoma przerwaniami zegarowymi: od SysTicka i od zwykłego licznika. W procedurach obsługi przerwań migać dwiema różnymi diodami. W głównej pętli programu umieścić instrukcję *wfi*. Przetestować działanie różnych trybów uśpienia i bitów opisanych w tym rozdziale.

170 oprócz: resetu, *tamper* (jeśli używany), *WKUP* (jeśli używany)

Przykładowe rozwiązanie (F103, diody na PB0 i PB1):

```
1. int main(void) {
2.
3.     RCC->APB2ENR = RCC_APB2ENR_IOPBEN | RCC_APB2ENR_TIM1EN;
4.     gpio_pin_cfg(GPIOB, PB0, gpio_mode_output_PP_2MHz);
5.     gpio_pin_cfg(GPIOB, PB1, gpio_mode_output_PP_2MHz);
6.
7.     TIM1->PSC = 10000;
8.     TIM1->ARR = 500;
9.     TIM1->DIER = TIM_DIER_UIE;
10.    TIM1->CR1 = TIM_CR1_CEN;
11.
12.    NVIC_EnableIRQ(TIM1_UP_IRQn);
13.
14.    SysTick_Config(800000);
15.
16.    SCB->SCR |= SCB_SCR_SLEEPDEEP;
17.
18.    while (1){
19.        __WFI();
20.    }
21.
22. } /* main */
23.
24. __attribute__((interrupt)) void TIM1_UP_IRQHandler(void){
25.     if (TIM1->SR & TIM_SR UIF){
26.         TIM1->SR = ~TIM_SR UIF;
27.         BB(GPIOB->ODR, PB0) ^= 1;
28.     }
29. }
30.
31. __attribute__((interrupt)) void SysTick_Handler(void){
32.     BB(GPIOB->ODR, PB1) ^= 1;
33. }
```

Program na początku konfiguruje piny obsługującego diody, licznik TIM1 tak aby generował przerwania zegarowe i włącza przerwania od SysTicka. Linijkę **16** proszę sobie na razie zakomentować we własnym zakresie. W procedurach obsługi przerwań są migania diodami. Nic nowego.

19) w pętli głównej programu umieszczono rozkaz *Wait for Interrupt*. Powoduje on przejście do trybu uśpienia (*sleep mode*). Wybudzenie następuje po pojawienniu się przerwania. W naszym programie są dwa przerwania (SysTick i TIM1) i oba powinny działać. To jest dobry moment, żeby przetestować sobie działanie debugera w trybach obniżonego poboru mocy (czy nie zrywa się połączenie).

16) po odkomentowaniu tej linijki procesor będzie przechodził do trybu uśpienia *stop mode*. Z tego trybu wybudzić może go tylko przerwanie EXTI. Czyli w naszym przypadku, żadna dioda nie powinna migać. Ze względu na błąd (opisany [tu](#)) przerwanie SysTick jednak wybudza procesor. W związku z tym dioda na PB1 miga. Mimo, że nie powinna! Na szczęście po odłączeniu debugera, wszystko działa tak jak powinno (przypominam: w OpenOCD debugger sam ustawia sobie bit DBG_STOP, jeśli debugger nie będzie podłączony to bit nie będzie ustawiany). Tyle.

Trochę informacji, przemyśleń i przykładów dotyczących również F103 znajduje się w rozdziale opisującym *eco-driving* w F429 (rozdział 11.7). Zdecydowanie proszę go przeczytać nawet jeśli ktoś nie zamierza korzystać z mikrokontrolera innego niż STM32F103 :)

Co warto zapamiętać z tego rozdziału?

- patrz rozdział 11.7

11.7. Tryby obniżonego poboru mocy (F429)

Podstawowa zasada, czyli im mniej jest włączone i wolniej działa tym mniej pobieramy energii, pozostaje niezmienna. Poza tym przybyło trochę nowości w stosunku do F103. Przede wszystkim mamy możliwość sterowania wewnętrznym regulatorem napięcia 1,2V. To napięcie zasila rdzeń, pamięci i peryferia cyfrowe mikrokontrolera. Im niższe napięcie tym mniejszy pobór prądu. Niestety kosztem maksymalnej częstotliwości pracy! Po szczegóły elektryczne odsyłam do datasheetu. W skrócie sprawa wygląda tak, że do wyboru są trzy poziomy (*scale*) napięcia:

- *scale 3* - najniższe napięcie (1,14V), maksymalna częstotliwość zegara szyny AHB¹⁷¹ wynosi 120MHz; regulator **zawsze** pracuje na tym poziomie jeśli wyłączona jest pętla PLL¹⁷¹
- *scale 2* - typowo 1,26V, maksymalna częstotliwość zegara szyny AHB wynosi 144MHz (lub 168MHz w trybie *over-drive*)
- *scale 1* - najwyższe napięcie (1,32V), maksymalna częstotliwość zegara szyny AHB wynosi 168MHz (lub 180MHz w trybie *over-drive*)

Zmiany poziomu napięcia można dokonać tylko, gdy układ PLL¹⁷¹ jest wyłączony a procesor taktowany jest wewnętrznym lub zewnętrznym oscylatorem dużej częstotliwości. Nowo ustawiony poziom aktywuje się, gdy zostanie włączony układ PLL¹⁷¹. Proste? Proste. Do tego mamy tryb extra: *over-drive mode*, w którym mikrokontroler może pracować z wyższą częstotliwością niż wynika z ograniczeń dla danego poziomu napięcia. W RMie jest opisana procedura uruchamiania tego bajeru. Niestety nie wiem, gdzie tu jest „haczyk” :)

W *sleep mode* nowością jest mechanizm, który pozwala na automatyczne wyłączanie zegarów układów peryferyjnych przy usypianiu mikrokontrolera. Bardzo użyteczne rozwiązanie - podoba mi się. Po przejściu do *sleep mode* (co usypia tylko CPU) zostaje wyłączony zegar dla peryferiów wskazanych w rejestrach RCC_APBxLPENR i RCC_AHBxLPENR. Skasowanie bitu związanego z danym peryferialem powoduje, że jego zegar zostaje wyłączony po

¹⁷¹ przyjmij na wiarę, szczegóły później... w rozdziale 17

uśpieniu mikrokontrolera. RM nie jest zbyt wylewny przy opisie tych rejestrów, ale zakładam że po wybudzeniu procesora, zegary zostaną przywrócone do stanu sprzed uśpienia! Jak coś jasno nie wynika z dokumentacji to zawsze można sobie na szybko napisać programik sprawdzający (patrz zadanie 11.2). F103 nie oferował takiego mechanizmu, więc trzeba było ręcznie wyłączać sygnały zegarowe niepotrzebnych bloków przed uśpieniem, a przywracać po wybudzeniu.

Stop mode jest pełen nowości. Tzn. ogólna idea się nie zmienia, ale przybyło kilka bitów konfiguracyjnych. Oczywiście im więcej elementów zostanie uśpionych (i im głębsze będzie to uśpienie) tym dłużej będzie trwało później rozburzanie mikrokontrolera. W trybie *stop mode* wewnętrzny regulator napięcia może pracować w jednym z dwóch trybów:

- *normal mode* - nic specjalnego, ale mamy dwie dodatkowe opcje:
 - uśpienie pamięci Flash: bit FPDS w rejestrze PWR_CR
 - przełączenie wewnętrznego regulatora napięcia w tryb *low-power*: bit LPDS w PWR_CR
- *under-drive mode* - jakiś tryb obniżonego poboru mocy (wiadomo - mniejszy pobór, ale dłuższe wybudzanie), włączany bitami UDEN w PWR_CR

W trybie *under-drive* Flash jest uśpiony z automatu. Za to dalej możemy włączyć tryb *low-power* regulatora. Myślę, że w tabelce będzie lepiej widać. Zwróć uwagę na czasy wybudzania (wartości typowe, źródło: datasheet) mikrokontrolera z poszczególnych trybów (bo z praktycznego punktu widzenia tylko tym się różnią... no i poborem energii):

Tabela 11.2 Opcje konfiguracji trybu *stop mode* w STM32F429

opcje	UDEN	MRUDS	LPUDS	LPDS	FPDS	narzut przy rozburzaniu
<i>normal mode</i>	-	-	0	-	0	-
	<i>FPD</i> ¹⁷²	-	0	-	0	<i>flash</i>
	<i>LP</i> ¹⁷³	-	0	0	1	<i>regulator z LP</i>
	<i>LP + FPD</i>	-	-	0	1	<i>flash i regulator z LP</i>
<i>under-drive mode</i>	<i>FPD</i>	3	1	-	0	<i>flash, regulator z UD¹⁷⁴, logika rdzenia</i>
	<i>LP + FPD</i>	3	-	1	1	<i>flash, regulator z LP i UD, logika rdzenia</i>

172 *Flash Power Down* - uśpienie pamięci Flash,

173 *Low Power* - tryb obniżonego poboru mocy regulatora napięcia

174 *Under Drive Mode*

Dla porównania, wybudzenie ze stanu:

- *sleep mode* trwa 6 cykli zegara CPU
- *stop mode* trwa 318μs

Zadanie domowe 11.2: sprawdzić działanie „automatycznego wyłącznika zegarów”. Niech dwa liczniki generują przerwania i migają diodami. I niech jeden z liczników będzie wyłączany po uśpieniu (rejestr RCC_APB1LPENR). Sprawdź generalnie czy i jak to działa :)

Przykładowe rozwiązań (F429, diody na PG13 i PG14):

```
1. int main(void) {
2.
3.     RCC->AHB1ENR |= RCC_AHB1ENR_GPIOGEN;
4.     RCC->APB1ENR |= RCC_APB1ENR_TIM2EN;
5.     RCC->APB1ENR |= RCC_APB1ENR_TIM3EN;
6.     __DSB();
7.
8.     //RCC->APB1LPENR &= ~(RCC_APB1LPENR_TIM2LPEN);
9.
10.    gpio_pin_cfg(GPIOG, PG13, gpio_mode_out_PP_LS);
11.    gpio_pin_cfg(GPIOG, PG14, gpio_mode_out_PP_LS);
12.
13.    TIM2->PSC = 8000-1;
14.    TIM2->ARR = 250-1;
15.    TIM2->DIER = TIM_DIER_UIE;
16.    TIM2->CR1 = TIM_CR1_CEN;
17.
18.    TIM3->PSC = 8000-1;
19.    TIM3->ARR = 250-1;
20.    TIM3->DIER = TIM_DIER_UIE;
21.    TIM3->CR1 = TIM_CR1_CEN;
22.
23.    NVIC_EnableIRQ(TIM2_IRQn);
24.    NVIC_EnableIRQ(TIM3_IRQn);
25.
26.    while(1){
27.        __WFI();
28.    }
29.
30. }
31.
32. void TIM2_IRQHandler(void){
33.     TIM2->SR = 0;
34.     BB(GPIOG->ODR, PG13) ^= 1;
35. }
36.
37. void TIM3_IRQHandler(void){
38.     TIM3->SR = 0;
39.     BB(GPIOG->ODR, PG14) ^= 1;
40. }
```

Ogólna idea programu nie powinna budzić wątpliwości, jeśli budzi to proponuję zacząć czytać od początku (Poradnik, nie rozdział :) Dwa liczniki, przerwania od przepełnienia, miganie diodami:

- licznik TIM2 migaj PG13 z częstotliwością ca. 4Hz
- licznik TIM3 migaj PB14 z częstotliwością ca. 4Hz

W głównej pętli programu procesor jest usypiany (*sleep mode*), liczniki zliczają, przerwania wybudzają mikrokontroler i obie diody migają. Se proszę przetestować czy wszystko działa to se pójdziemy dalej z tą robotą.

Cały cymes siedzi w linijce 8. Ta linijka sprawia, że po uśpieniu mikrokontrolera wyłączony zostaje sygnał zegarowy licznika TIM2. Czyli: po uśpieniu mikrokontrolera licznik TIM2 nie będzie zliczał → nie będzie generował przerwań → nie wybudzi procka → dioda nie będzie migać. TIM3 będzie działał bez zmian i migał diodą. Zgoda? No to czas odkomentować linijkę. Ale wcześniej, przed skompilowaniem i wgraniem programu, wprowadźmy jeszcze dwie (małe i nieznaczące) modyfikacje w konfiguracji licznika TIM2 (zaraz się wyjaśni po co):

- `TIM2->PSC = 10`
- `TIM2->ARR = 10`

Zmiany, zapis, komplikacja, flash... krew, pot, łzy i niedowierzanie. Migają obie diody, prawda? A mówiłem przed chwilą, że TIM2 nie będzie działał... No trochę oszukałem... żeby sprawdzić Twoją czujność rzecz jasna (i urozmaicić przykład) :)

Zadanie domowe 11.3: przemyśleć sprawę i odpowiedzieć na pytanie - czemu dioda PG13 (z zadanie 11.2) miga, mimo że TIM2 jest wyłączały na czas uśpienia?

I jak? Odpowiedź jest prosta jak wszystko w STM32. Konfiguracja z ósmej linijki kodu powoduje, że zegar licznika TIM2 jest wyłączały wtedy kiedy procesor jest uśpiony. Ale w naszym przykładzie TIM3 regularnie wybudza procek na czas obsługi swojego przerwania. I właśnie w czasie obsługi tego ISR, kiedy procek nie śpi, licznik TIM2 cichaczem sobie zlicza. Oczywiście obsługa ISR jest „krótko i rzadko”. Z tego względu zmniejszyliśmy nastawę preskalera i rejestru przeładowania licznika TIM2, aby móc zaobserwować miganie. Bez tego musielibyśmy czekać znacznie dłużej na zmianę stanu diody. Jak łatwo policzyć, gdyby licznik TIM2 pracował cały czas, to przerwania występowaliby z częstotliwością trochę ponad 66kHz. A patrząc na diody widać, że PG13 migają nawet wolniej niż PG14 (4Hz).

Zadanie domowe 11.4: napisać program z licznikiem generującym przerwanie zegarowe migające diodą. W funkcji *main*, w nieskończonej pętli, umieścić:

- rozkaz usypiający procesor (*wfi*, tryb *sleep mode*)
- rozkaz przełączający inną diodę świecącą

Sprawdzić działanie programu. Następnie przetestować następujące przypadki:

- ustawiony bit SLEEPONEXIT w SCB_SCR
- globalnie wyłączone przerwania
- przerwanie od licznika wyłączone w kontrolerze NVIC

porównać działanie programu i wyciągnąć błyskotliwe wnioski :)

Przykładowe rozwiązań (F429, diody na PG13 i PG14):

```
1. int main(void) {
2.
3.     RCC->AHB1ENR |= RCC_AHB1ENR_GPIOGEN;
4.     RCC->APB1ENR |= RCC_APB1ENR_TIM2EN;
5.     __DSB();
6.
7.     gpio_pin_cfg(GPIOG, PG13, gpio_mode_out_PP_LS);
8.     gpio_pin_cfg(GPIOG, PG14, gpio_mode_out_PP_LS);
9.
10.    TIM2->PSC = 8000-1;
11.    TIM2->ARR = 250-1;
12.    TIM2->DIER = TIM_DIER_UIE;
13.    TIM2->CR1 = TIM_CR1_CEN;
14.
15.    NVIC_EnableIRQ(TIM2_IRQn);
16.
17. //SCB->SCR |= SCB_SCR_SLEEPONEXIT_Msk;
18.
19.    while(1){
20.        //__disable_irq();
21.        __WFI();
22.        BB(GPIOG->ODR, PG13) ^= 1;
23.        //__enable_irq();
24.        //TIM2->SR = 0;
25.        //NVIC_ClearPendingIRQ(TIM2_IRQn);
26.    }
27.
28. }
29.
30. void TIM2_IRQHandler(void){
31.     TIM2->SR = 0;
32.     BB(GPIOG->ODR, PG14) ^= 1;
33. }
```

Rozpoczynamy od wersji z zakomentowanymi linijkami jak wyżej. Odpalamy i... obie diody migają razem. Wiemy czemu? Program główny konfiguruje licznik i zasypia w linii 21. Pojawia się przerwanie od licznika, które miga pierwszą diodą w ISR (linijka 32). Przerwanie wybudziło procesor, więc zaczyna on wykonywać kolejne instrukcje za *wfi*. Linia 22 to miganie drugą diodą. Pętla główna się zapętla i proceek zasypia aż do kolejnego przerwania. Ta dam! Podglądając przebiegi na analizatorze (czy też na oscylowizorze) można zauważyc przesunięcie między sygnałami sterującymi jedną i drugą diodą. Stan diody w przerwaniu zmienia się szybciej o jakieś 1,125 μ s (czyli 18 cykli zegara CPU¹⁷⁵). Brzmi całkiem sensownie.

175 STM32F429 domyślnie chodzi na 16MHz, szczegóły w rozdziale 17

Dorzućmy wyłączenie przerwań w linii 20. Po uruchomieniu programu dioda PG14 milczy, zaś PG13 nie migra tylko świeci. Uważny obserwator zauważa jednakże, że nie świeci ona pełnym blaskiem :) Analizator (ewentualnie oscyloskop jak ktoś jest "pro") potwierdza, dioda PG13 migra, ino chyba. Zastanówmy się czemu? Otóż procesor w ogóle nie zasypia! Cytat z *Cortex Generic User Guide*:

„WFI is a hint instruction that suspends execution until one of the following events occurs: [...] an interrupt masked by PRIMASK becomes pending”

Czyli dokładnie tak jak u nas. Pojawia się przerwanie, ale nie może być obsłużone gdyż przerwania są wyłączone (w rejestrze specjalnym PRIMASK, linia 20). W związku z tym przechodzi w stan *pending* (w NVICu) i nie pozwala uspić procka. Pętla główna cały czas się kręci (PG13 migra bardzo szybko).

No to w ramach zabawy dorzućmy linijkę 25, czyli kasowanie stanu *pending* w NVICu. Odpalamy i widzimy, że nie ma żadnej różnicy. Chwilka zastanowienia i dochodzimy do wniosku, że nie kasujemy flagi przerwania w układzie peryferyjnym (patrz analiza zadania 7.1), więc NVIC odczytuje to jako kolejne przerwanie i na nowo ustawia stan oczekiwania. Dopusujemy szybko linijkę 24. No! Teraz jest miód malina. Dioda PG13 migra! PG14 natomiast milczy, bo przerwania są wyłączone. Nadążasz? Przerwania cyklicznie budzą procesor, ale są wyłączone toteż ISR nie może się wykonać.

Cofnijmy się trochę i zakomentujmy nazad linijki 24, 25. Zamiast nich dorzućmy linijkę 23. Uruchamiamy program i widzimy, że znowu obie diody migają tak samo jak wcześniej. Czy na pewno? Otóż niet! Analizator logiczny prawdę Ci powie. Owszem, obie diody migają, ale teraz stan PG13 zmienia się wcześniej niż PG14. Różnica wynosi około 28 cykli procesora. Dlaczego tak? Procesor jest usypanym z zablokowanymi przerwaniami. Pojawia się przerwanie od licznika, które wybudza procesor. Przerwania dalej są zablokowane, więc procesor nie może obsłużyć tego ISR. Zamiast tego wykonuje kolejne instrukcje pętli głównej. Skok do ISR następuje dopiero po włączeniu przerwań w linii 23. I to jest wbrew pozorom ważny przypadek! Dzięki temu mechanizmowi możemy wymusić wykonanie jakichś działań tuż po wybudzeniu procesora a przed obsługą przerwania, które procesor wybudziło¹⁷⁶.

Teraz dla odmiany wyrzucamy linie 20, 23, 24, 25. Dorzucamy za to linię 17 i ponawiamy obserwacje. Tym razem migra tylko dioda z przerwania! Tzn. że procesor w ogóle nie wykonuje linii 22! Nigdy, bezapelacyjnie, do samego końca¹⁷⁷...! SLEEPONEXIT powoduje, że gdy tylko procesor

176 żeby nie było, sam na to nie wpadłem :) jest to opisane w *Generic User Guide* rdzenia :)

177 no chyba, że skasujemy bit SLEEPONEXIT :)

opusci ISR - natychmiast zasypia. Pętla w ogóle nie jest potrzebna. Program mógłby się kończyć na rozkazie uśpienia. Nie pójdzie dalej dopóki jest ustawione SLEEPONEXIT.

Zadanie domowe 11.5: w rozwiązaniu zadania 11.4 podmienić `wfi` na `wfe` i przeprowadzić analogiczne zabawy edukacyjne. Wyciągnąć zacne wnioski.

Przykładowe rozwiązanie (tylko zmieniony fragment kodu z rozwiązania do zadania 11.4):

```
1. //__disable_irq();
2. NVIC_EnableIRQ(TIM2 IRQn);
3. //SCB->SCR |= SCB_SCR_SEVONPEND_Msk;
4.
5. while(1){
6.     WFE();
7.     // WFE();
8.     NOP();
9.
10.    //TIM2->SR = 0;
11.    //NVIC_ClearPendingIRQ(TIM2 IRQn);
12.    BB(GPIOG->ODR, PG13) ^= 1;
13. }
```

Reszta kodu bez zmian w stosunku do poprzedniego przykładu. Co do linijki 8 - odsyłam do *erraty* (wspominałem o tym też [tu](#)). Miało być lekko i łatwo a tu niespodzianka! Po zamianie `wfi` na `wfe` dzieją się cuda. Dioda w przerwaniu migła, a PG13 milczy (kod z zakomentowanymi linijkami jw.). Po oględzinach analizatorem okazuje się, że PG13 zapala się na niecałą mikro sekundę po każdym zboczu sygnału diody PG14. Zaprawdę powiadam Ci zdurnialem. Ale już się odnalazłem (chyba). Cały myk polega na tym, że instrukcja `wfe` oczekuje zdarzenia a nie przerwania. A w przykładowym kodzie dostaje aż dwa zdarzenia naraz. Pierwsze zdarzenie jest związane z ustawieniem flagi przerwania w liczniku, drugie z obsługą tego przerwania w NVICu. Przynajmniej tak to sobie wytlumaczyłem... Czyli:

- usypiamy procesor instrukcją `wfe`
- przychodzi pierwsze zdarzenie (licznik zgłasza przerwanie) i wybudza procesor
- to daje jeden obieg pętli głównej i np. zapalenie diody PG13
- procesor znowu zatrzymuje się na `wfe`
- przychodzi drugie zdarzenie¹⁷⁸ (z NVICa)
- drugi obieg pętli głównej gasi diodę
- procesor zasypia

Ok. Wiem, że ta teoria jest trochę naciągnięta, ale idealnie pasuje do objawów. Na próbę proszę odkomentować drugą instrukcję `wfe`. Zgodnie z moją teorią powinno to naprawić sytuację,

¹⁷⁸ oj boję się że błędę...

bo program będzie dwa razy czekał na zdarzenie. Bingo! Po zmianie, obie diody migają razem, tak jak powinny :) Wydaje mi się, że to potwierdza moją teorię o dwóch zdarzeniach... przynajmniej odrobinę. Próbowałem też kasować flagę przerwania w peryferialu i *pending* w NVICu (linijki 10 i 11) ale nic to nie zmieniało. Poszukałem trochę w nocy i nie tylko ja na tym polu zbłądziłem. Polecam odszukać na forum ST dwa tematy: *WFE - Exiting stop mode* oraz *STM32F3 stop mode problem (WFE)*. Dotyczą one dokładnie takiej sytuacji jaką ja zaobserwowałem przed chwilą. Joseph Yiu¹⁷⁹ sugeruje, aby zamiast pojedynczej instrukcji *wfe* używać zawsze sekwencji rozkazów *sev + wfe + wfe*. Po szczegóły odsyłam do wspomnianych tematów, dokumentacji lub serii książek *Definitive Guide to ARM Cortex Mx*.

Tak czy siak, w tych zabawach zapomniałem o sednie zagadnienia. Instrukcja *wfe* nie jest stworzona z myślą o pracy z przerwaniami (po to jest *wfi*). Wyłączmy więc przerwanie (w NVICu lub globalnie), włączmy bit SEVONPEND, wywalmy drugie *wfe*, dodajmy kasowanie flagi przerwania i *pendingu*¹⁸⁰. I sprawdźmy co się dzieje. Tym razem wszystko działa tak jak się spodziewałem. Dioda z przerwania nie migła (przerwanie jest przecież wyłączone), dioda z pętli migła. Przynajmniej tyle... Przy czym i tak sugerowałbym stosowanie tria *sev+wfe+wfe*.

Co warto zapamiętać z tego rozdziału?

- im mniej bloków ma włączony sygnał zegarowy i im wolniej pracują, tym mniej energii zużywa mikrokontroler
- są trzy tryby uśpienia:
 - *sleep mode* - zatrzymany tylko rdzeń
 - *stop mode* - zatrzymany rdzeń i peryferia
 - *standby mode* - wszystko zatrzymane
- im głębiej uśpimy mikrokontroler tym mniej energii będzie pobierał, ale wydłuży się czas wybudzania
- procesor może automatycznie zasypiać jeśli nie ma żadnego przerwania do obsłużenia (bit SLEEPONEXIT)
- w przypadku korzystania z instrukcji *wfe* mądrzy ludzie zalecają użycie sekwencji *sev wfe wfe*
- oczekujące przerwanie (bo np. jest wyłączone w NVICu) uniemożliwi uśpienie mikrokontrolera
- w F429 jest fajna funkcja automatycznie wyłączająca zegar wybranych peryferiali po uśpieniu rdzenia

179 autor serii książek *Definitive Guide to ARM Cortex-Mx*

180 wyłączyliśmy przerwania, więc flagi same się nie skasują

11.8. Tryby obniżonego poboru mocy (F334)

Dobra wiadomość dla tych, którzy nie lubią nowości. Pod względem obniżania poboru mocy układ F334 zachowuje się praktycznie tak samo jak F103. Najważniejszą różnicą jest to, że mikrokontroler może być wybudzony z trybu *Stop Mode* przed interfejsy komunikacyjne (USART, I2C) poprzez linie EXTI. Szczegóły zostaną omówione w rozdziałach poświęconych tym układom.

Tabela 11.3 Podsumowanie trybów uśpienia

Tryb uśpienia		Sleep Mode	Stop Mode	Standby Mode
Konfiguracja bitów przy usypianiu		SLEEPDEEP = 0 SLEEPONEXIT = 0/1	SLEEPDEEP = 1 PDDS = 0 LPDS = 0/1 (opcjonalne uśpienie regulatora napięcia) (wszystkie flagi przerwań muszą być skasowane)	SLEEDPEED = 1 PDDS = 1 WUF = 0
Zachowanie wybranych układów w stanie uśpienia	GPIO	zachowują stan	zachowują stan	HiZ ¹⁸¹
	wybudzanie	6 cykli zegara	LPDS = 1: 5,5μs LPDS = 0: 3,8μs	54μs
	regulator 1,8V	pracuje	zależnie od LPDS w PWR_CR	wyłączony
	peryferia	pracują	wyłączone (z wyjątkiem IWDG, RTC, LSI, LSE; energię pobierać może także ADC i DAC)	
Pobudka	wfi	przerwanie włączone w NVIcu	przerwanie od EXTI	-reset zewnętrzny -reset IWDG -RTC alarm (EXTI17) -WKUP pin (zbocze rosnące)
	wfe	przerwanie + SEVONPEND lub zdarzenie od EXTI	zdarzenie od EXTI	
Orientacyjny pobór prądu (85°C; 3,3V; zewnętrzny oscylator):		peryferia wyłączone: 0,76mA @1MHz, 6,3mA @72MHz peryferia włączone: 1,55mA @8MHz, 51,8mA@72MHz	regulator: - włączony: 18,6μA - w trybie low power: 7μA	watchdog: - wyłączone: 0,93μA - włączony: 1,28μA
Uwagi:		-	po wybudzeniu jako źródło zegara zostaje ustalony HSI	utrata zawartości SRAM i rejestrów (poza BKP i RCC_SCR), Flaga PWR_CSR_SBF

181 oprócz: resetu, *tamper* (jeśli używany), *WKUP* (jeśli używany)

12. MECHANIZM DMA („*ANNUNTIO VOBIS GAUDIUM MAGNUM: HABEMUS DMA*”¹⁸²)

12.1. Z czym to się je?

Przy przesiadaniu się z AVR na STM32, DMA (*Direct Memory Access*) stanowiło dla mnie jakąś niepojętą mroczną zagadkę. Podchodziłem do tego jak pies do jeża. A prawa jest prosta – jak wszystko w STM¹⁸³. DMA to taki twór do przesyłania danych między pamięcią i periferiami w sposób sprzętowy. Jak to rozumieć?

Wyobraźmy sobie, że napełniamy wiadro wodą za pomocą szklanki i jednocześnie czytamy książkę. Podstawiamy więc szklankę pod kran, odkręcamy wodę i mamy wolne do momentu jak szklanka się napełni – w tym czasie możemy sobie czytać. Gdy szklanka się napełni przerywamy czytanie i przelewamy wodę do wiadra, po czym zaczynamy od nowa. Jest to taka moja nieudolna analogia do działania procesora bez DMA. Na przykładzie AVR: uruchamiamy przetwornik ADC (ustawienie szklanki i puszczenie wody). Dopóki trwa pomiar, procesor może zająć się czymkolwiek innym (czytamy książkę). Gdy pomiar się zakończy (pełna szklanka) zostaje zgłoszone przerwanie (woda cieknie nam do rękawa). AVRek przerywa cokolwiek robił (czytanie) i np. zapisuje wynik z ADC do jakieś tablicy (przelanie wody do wiadra).

Wyobraźmy sobie teraz, że wpadliśmy na szczwany pomysł i zamiast metody szklankowej podstawiliśmy pod kran rurkę. Rurka kończy się w wiadrze (nie jest to idealna analogia ale lepszej na razie nie mam). Teraz po ustawnieniu rurki (konfiguracji sprzętu) możemy odkroić kran i woda sama będzie lała się do wiaderka, a my w tym czasie czytamy do woli. Pi razy drzwi tak wyobrażam sobie DMA.

Po skonfigurowaniu połączenia np. między ADC a jakiś obszarem pamięci (buforem/tablicą), dane są przesyłane „sprzętowo” i nie musimy się tym zajmować. Odpalamy i zapominamy. Inne przykłady wykorzystania DMA (tak żeby sobie wyrobić pogląd):

- przesyłanie danych obrazu z bufora w SRAM do sterownika wyświetlacza (np. przez SPI)
- przesyłanie próbek z bufora w SRAM do DAC aby wygenerować żądany przebieg
- wspomniane w przykładzie zapisywanie wyników z ADC do bufora SRAM
- kopiowanie bloków pamięci w SRAMie (coś jak funkcja *memcpy*)
- przesyłanie danych z jednego interfejsu do innego (np. *usart echo*)
- i wiele innych...

182 „Ogłaszam wam radość wielką: mamy DMA” ;)

183 tylko czasem ciężko rozkminić :)

Oczywiście bez DMA (z małymi wyjątkami) da się żyć. Dane można kopiować za pomocą prostej pętli lub funkcji typu *memcpy*, wyniki z ADC odbierać w przerwaniu... Tyle że to angażuje procesor – a nam zależy na tym, żeby jak najwięcej rzeczy działało się „samo”.

Co warto zapamiętać z tego rozdziału?

- DMA to mechanizm umożliwiający sprzętowe przesyłanie danych w obrębie pamięci i rejestrów układów peryferyjnych

12.2. DMA (F103)

W mikrokontrolerze są dwa kontrolery DMA i 17 kanałów DMA. Kanał DMA to coś jak ta rurka z przypowieści o napełnianiu wiadra wodą. Każdy kanał (rurkę) możemy oddziennie skonfigurować. DMA najczęściej współpracuje z jakimś układem peryferyjnym, który wysyła żądania DMA. Żądanie jest jak odblokowanie rurki, powoduje że DMA wykonuje jedną transakcję, czyli przesyła porcję danych o zaprogramowanej wielkości (8, 16, 32 bity) i czeka na kolejne żądanie. Np. przetwornik ADC może wysyłać żądanie po każdej skończonej konwersji, a DMA będzie przesyłać wynik konwersji z rejestru przetwornika do pamięci SRAM.

W praktyce, użycie DMA, wygląda mniej więcej tak:

- w rejestrach DMA_CPAR¹⁸⁴ i DMA_CMAR¹⁸⁵ ustawiamy adresy pomiędzy którymi mają być przesyłane dane (kierunek przesyłu skonfigurujemy za chwilę)
- w DMA_CNDTR ustawiamy ile ma być pojedynczych transferów (transakcji) DMA, czyli ile danych (o ustalonym rozmiarze) ma być w sumie przesłanych zanim (w uproszczeniu) kanał się wyłączy; rejestr jest 16 bitowy więc maksymalnie możemy ustawić 65 535 transferów
- w rejestrze DMA_CCR konfigurujemy bajery wedle uznania:
 - MEM2MEM - w tym trybie przesyłanie startuje od razu po włączeniu kanału DMA, nie są wymagane żądania od peryferiala. Tryb używany w szczególności przy przesyłaniu danych między dwoma obszarami pamięci SRAM (stąd jego nazwa *memory to memory*).
 - PL - priorytety - jeśli używamy kilku kanałów DMA to możemy je sobie „popriorytować” wedle uznania
 - PSIZE i MSIZE - wybieramy jaka jest wielkość pojedynczej „porcji danych” (8, 16, 32 bit) po stronie źródłowej i docelowej

184 Channel Peripheral Address Register

185 Channel Memory Address Register

- MINC, PINC - inkrementacja adresów - włączenie tej funkcji powoduje, że po każdym transferze adres (docelowy lub źródłowy) zwiększy się o wartość wynikającą z wielkości przesyłanych danych (+1 dla danych 8b; +2 dla 16b; +4 dla 32b)
 - CIRC - tryb kołowy - po wyzerowaniu licznika transferów (CNDTR) kontroler DMA przywróci mu początkową wartość i zacznie transferować od nowa (od adresów bazowych jeśli była włączona inkrementacja)
 - DIR – kierunek przepływu danych
 - TEIE – włączenie przerwania od błędu transmisji danych
 - HTIE – włączenie przerwania wywoływanego po przesłaniu połowy danych
 - TCIE – włączenie przerwania od zakończenia przesyłania (wyzerowanie CNDTR)
 - EN – włączenie kanału DMA
- rejestr DMA_ISR zawiera flagi przerwań DMA (tylko do odczytu)
 - rejestr DMA_IFCR służy do kasowania flag z poprzedniego rejestru (wpisanie jedynki kasuje flagę)

I to właściwie cała magia. No to siup:

Zadanie domowe 12.1: napisać program, który za pomocą DMA kopiuje blok pamięci (np. tablicę znaków) w inne miejsce pamięci SRAM. Po zakończeniu kopiowania oba bloki są porównywane i w zależności od wyniku porównania zapalana jest jedna lub druga dioda świecąca.

Przykładowe rozwiążanie (F103, diody na PB0 i PB1):

```

1. int main(void) {
2.
3.     static char bufor1[] = "Ala ma kota, a sierotka ma rysia.";
4.     static volatile char bufor2[50];
5.     size_t size = sizeof(bufor1);
6.
7.     RCC->APB2ENR = RCC_APB2ENR_IOPBEN;
8.     gpio_pin_cfg(GPIOB, PB0, gpio_mode_output_PP_2MHz);
9.     gpio_pin_cfg(GPIOB, PB1, gpio_mode_output_PP_2MHz);
10.
11.    //memcpy(bufor2, bufor1, size);
12.
13.    RCC->AHBENR |= RCC_AHBENR_DMA1EN;
14.    DMA1_Channel1->CMAR = (uint32_t)bufor1;
15.    DMA1_Channel1->CPAR = (uint32_t)bufor2;
16.    DMA1_Channel1->CNDTR = size;
17.    DMA1_Channel1->CCR = DMA_CCR1_MEM2MEM | DMA_CCR1_MINC | DMA_CCR1_PINC | DMA_CCR1_DIR;
18.    DMA1_Channel1->CCR |= DMA_CCR1_EN;
19.
20.    while( (DMA1->ISR & DMA_ISR_TCIF1) == 0 );
21.
22.    if ( memcmp(bufor1, bufor2, size) ) BB(GPIOB->ODR, PB0) = 1;
23.    else BB(GPIOB->ODR, PB1) = 1;
24.
25.    while (1);
26.
27. } /* main */

```

- 3)** to jest nasz bufor źródłowy, z którego będziemy kopiowali dane
- 4)** bufor docelowy (pewnie za duży, ale nie chciało mi się dokładnie liczyć literek). Zwróć uwagę na modyfikator *volatile*, w tym programiku nie jest może super potrzebny. Tym niemniej DMA powoduje sprzętową zmianę zawartości pamięci. Kompilator nie jest tego świadom, bo ta zmiana nie wynika jawnie z programu. Trzeba więc go ostrzec, że dane modyfikowane przez DMA mogą się po cichu zmieniać. Stąd *volatile*.
- 5)** zmienna pomocnicza przechowująca ilość danych do przesłania, wprowadzona dla wygody (w sumie powinna być *const*, ale nie chce mi się już zmieniać)
- 11)** tak by wyglądało kopiowanie za pomocą funkcji *memcpy*, niestety zajmuje ono CPU a nam chodzi o to, żeby dane kopiowały się „same”
- 13)** włączenie zegara dla kontrolera DMA1
- 14, 15)** ustawienie adresu docelowego i źródłowego. Nazwy tablic są w C wskaźnikami na początek zajmowanego obszaru pamięci, więc nie potrzeba operatora „&”. Rzutowanie jest po to, żeby kompilator się nie czepiał. Zwróć uwagę na sposób dostępu do rejestru, np:
- `DMA1_Channel1->CMAR`
- To jest rejestr CMAR dla pierwszego kanału kontrolera DMA1. Każdy kanał DMA ma swoje rejesty CMAR, CPAR, CNDTR, CCR. Stąd taki zapis. Z drugiej strony rejesty ISR i IFCR są wspólne dla całego kontrolera DMA (patrz linijka 20).
- 16)** ustawiam ilość transferów DMA. Jeden transfer to pojedyncze przesłanie danych o ustalonym (za chwilę go ustawimy) rozmiarze. W przykładzie przesyłam dane o rozmiarze 1B.
- 17)** rejestr konfiguracyjny DMA. Ustawiam:
- tryb *mem2mem* - bez tego kontroler DMA wymaga wyzwalania przez układ peryferyjny (poprzez żądania DMA). W trybie mem2mem DMA rusza od razu po włączeniu kanału; w tym przykładzie o żadnym wyzwalaniu przez peryferial nie może być mowy bo przesyłamy dane z pamięci do pamięci :)
 - inkrementację adresów (źródłowego i docelowego) - do rejestrów adresowych (CMAR, CPAR) wpisałem adresy początków tablic. Gdyby nie było inkrementacji to DMA cały czas odczytywałoby zerowy element tablicy *bufor1* i zapisywało pod adresem zerowego elementu tablicy *bufor2*. Inkrementacja powoduje, że po każdym transferze adres jest zwiększany, czyli będą odczytywane (i zapisywane) kolejne elementy tablic.
 - kierunek przepływu danych od CMAR do CPAR

18) włączam kanał DMA. W tym momencie rozpoczyna się przesyłanie danych (bo wybrałem tryb *mem2mem*).

20) czekam na ustawienie flagi końca transferów żeby mieć pewność że wszystko się przesłało. Zwrót uwagę na nazwę bitu: DMA_ISR_TCIF1. Ta jedynka na końcu oznacza numer kanału!

22, 23) porównanie bloków pamięci i zapalenie odpowiedniej diody

Kilka uwag:

- zapis do CNDTR, CMAR i CPAR jest możliwy tylko gdy kanał jest wyłączony
- po włączeniu DMA rejestr CNDTR jest tylko do odczytu i wskazuje ile jeszcze transferów pozostało do wykonania (dekrementuje się po każdym transferze)
- jak CNDTR dojdzie do zera to przesyłanie zostaje przerwane lub rozpoczyna się od nowa jeśli wybrany jest tryb kołowy (CIRC)
- jeśli przesyłanie się zakończyło ($CNDTR = 0$ i nie jest włączony tryb kołowy) to bit włączający kanał (DMAx_CCR_EN) pozostaje ustawiony! Przypominam, że nie jest możliwa modyfikacja rejestrów CPAR, CMAR, CNDTR dopóki kanał jest włączony. Czyli jeśli chcemy taki kanał włączyć jeszcze raz, to musimy:
 - skasować bit EN
 - skonfigurować kanał (przynajmniej ustawić nową wartość CNDTR)
 - ponownie włączyć bit EN
- jeżeli po stronie źródłowej i docelowej będą ustawione różne rozmiary danych to sprawa się nieco komplikuje – odsyłam do tabelki *Programmable data width & endian behavior (when bits PINC = MINC = 1)*. Przykładowo (w ramach ćwiczeń ogarniania tabelki) rozważmy taki motyw, że źródło ustawimy na 8b zaś cel na 32b, włączymy inkrementację i wykonamy cztery transfery:
 - pierwszy transfer ze źródła odczyta jeden bajt (o umownej wartości B0) i zapisze go pod adresem docelowym jako wartość 32b, czyli 0x0000 00B0
 - nastąpi inkrementacja adresów: po stronie źródłowej o 8b → 1B → czyli adres wzrośnie o jeden; po stronie docelowej o 32b → 4B → adres wzrośnie o 4
 - drugi transfer odczyta ze źródła jeden bajt (o umownej wartości B1) i po stronie wtórnej zapisze w postaci 32b: 0x0000 00B1
 - nastąpi inkrementacja adresów: po stronie źródłowej o 8b → 1B → czyli adres wzrośnie o jeden; po stronie docelowej o 32b → 4B → adres wzrośnie o 4
 - itd...

jeżeli rozmiar źródłowy będzie większy niż docelowy to dane zostaną ucięte (zostanie tylko młodsza część). Odsyłam do tabelki.

- inkrementacja adresów umożliwia zapisanie bloku w pamięci SRAM (tablicy) np. jeśli ustawimy adres docelowy na początek tablicy w SRAM to po zapisaniu każdej wartości kontroler DMA sam sobie będzie przesuwał „wskaźnik” zapisu – czyli będzie zapisywał pod kolejnymi pozycjami w tablicy (tak w skrócie)
- poza „szczegółowymi” flagami przerwań (np. połowa transmisji czy koniec transmisji) dostępna jest również taka ogólna flaga (DMA_ISR_GIFx) - jest ona ustawiana jeśli została ustawiona jakakolwiek „szczegółowa” flaga DMA; po skasowaniu wszystkich flag szczegółowych można skasować tą ogólną; flaga ogólna nie wyzwala przerwania
- do dyspozycji mamy dwa kontrolery DMA1 (ma 7 kanałów) i DMA2 (ma 5 kanałów)
- przy włączonej inkrementacji rejestr CMAR, CPAR cały czas zawierają wartość początkową, program nie ma dostępu do „aktualnych” adresów wynikających z działania inkrementacji
- tryby *circular* i *mem2mem* się nie łączą
- rejestr CCR, CMAR, CPAR zachowują swoją wartość po zakończeniu transmisji lub wyłączeniu kanału DMA
- błędy w transmisji powstają np. przy próbie odczytu/zapisu niewłaściwych adresów (zapis do pamięci Flash itp...), pojawienie się błędu automatycznie wyłącza kanał (zeruje bit EN)

Zadanie domowe 12.2: program wypełniający (za pomocą DMA) tablicę w pamięci, stałą wartością równą np. 32.

Zadanie domowe 12.3: jeden licznik coś sobie liczy (dla wygody będzie to zwykły timer). Drugi timer co 250ms odpala DMA (generuje żądanie), które zapisuje aktualną wartość pierwszego licznika w buforze w pamięci SRAM. W sumie mają być cztery takie zapisy (transfery DMA). Do kodu! Timery start!

Przykładowe rozwiązań (F103):

```
1. int main(void) {
2.
3.     static volatile uint32_t wyniki[4];
4.
5.     RCC->APB2ENR = RCC_APB2ENR_IOPBEN;
6.     RCC->APB1ENR = RCC_APB1ENR_TIM2EN | RCC_APB1ENR_TIM3EN;
7.     RCC->AHBENR |= RCC_AHBENR_DMA1EN;
8.
9.     gpio_pin_cfg(GPIOB, PB0, gpio_mode_output_PP_2MHz);
10.    gpio_pin_cfg(GPIOB, PB1, gpio_mode_output_PP_2MHz);
11.
12.    TIM2->PSC = 8000-1;
13.    TIM2->ARR = UINT16_MAX;
14.    TIM2->EGR = TIM_EGR_UG;
15.
16.    TIM3->PSC = 8000-1;
17.    TIM3->ARR = 250-1;
18.    TIM3->DIER = TIM_DIER_UDE;
19.    TIM3->EGR = TIM_EGR_UG;
20.
21.    DMA1_Channel3->CMAR = (uint32_t)wyniki;
22.    DMA1_Channel3->CPAR = (uint32_t)&TIM2->CNT;
23.    DMA1_Channel3->CNDTR = 4;
24.    DMA1_Channel3->CCR = DMA_CCR3_MSIZE_1 | DMA_CCR3_PSIZE_0 | DMA_CCR3_MINC | DMA_CCR3_EN;
25.
26.    TIM2->CR1 = TIM_CR1_CEN;
27.    TIM3->CR1 = TIM_CR1_CEN;
28.
29.    while (1);
30.
31. } /* main */
```

3) tablica na wartości licznika przesłane przez DMA, typ 32 bitowy żeby było bardziej edukacyjnie (rejestr licznika jest 16 bitowy)

5, 6, 7) zegary dla dwóch liczników i DMA; port B i konfiguracja pinów z linii 9 i 10 zapłatała się przez przypadek :>

12, 13) konfiguracja pierwszego licznika, timer zlicza z częstotliwością 1kHz

14) rejestr preskalera jest buforowany (nowa wartość zostaje wpisana przy UEV, patrz rozdział 8.3), wymuszam programowy UEV żeby nowa wartość została wpisana do rejestru od razu

16 - 19) konfiguracja licznika wyzwalającego DMA, włączono generowanie żądań transferów DMA przy UEV, policzenie okresu UEV pozostawiam Tobie :)

21) podanie adresu docelowego w przestrzeni pamięci

22) podanie adresu źródłowego, adresu rejestru TIM2->CNT. W poprzednim przykładzie nie miało znaczenia w którym rejestrze (CMAR lub CPAR) zapiszemy adres źródłowy/docelowy, gdyż ponieważ, oba adresy były w przestrzeni SRAM. Tutaj jest inaczej, jeden z adresów (adres rejestru TIM2->CNT) leży w przestrzeni układów peryferyjnych i musimy umieścić go w rejestrze CPAR (*Channel Peripheral Address Register*). Inaczej nie będzie działać¹⁸⁶ :) Kierunek przesyłu oczywiście wybieramy sobie bitem DIR.

23, 24) reszta konfiguracji i włączenie kanału DMA.

26, 27) włączamy oba liczniki

186 tak, oczywiście że sprawdziłem :)

Dwie uwagi:

- specjalnie wybrałem różne rozmiary danych na wejściu i wyjściu aby urozmaicić przykład - proszę docenić inwencję :)
- przesył nie startuje od razu po włączeniu kanału (tak jak w poprzednim przykładzie) bo nie włączliśmy trybu *mem2mem*, toteż kontroler DMA czeka na żądanie od układu peryferyjnego

No dobra... ale skąd kontroler DMA wie, który układ ma go wyzwałać? Inaczej mówiąc: jak połączyć konkretny peryferial (generujący żądania DMA) z konkretnym kanałem DMA? Wybornie, że o to pytasz :) Możliwe połączenia są z góry ustalone. Odpalamy rozdział *DMA request mapping* i zjeżdżamy do diagramu *DMA1 request mapping*. Pokazuje on co może wyzwałać poszczególne kanały kontrolera DMA1. Np kanał 1 może być wyzwalany przez:

- przetwornik ADC1
- trzeci kanał licznika TIM2
- pierwszy kanał licznika TIM4

Wszystkie te sygnały są sumowane, więc nie ma możliwości wykrycia (przez kontroler DMA), który z nich wyzwolił kanał. Kanał powinien współpracować z tylko jednym źródłem wyzwalania jednocześnie. Żeby nie męczyć wzroku, proponuję zjechać niżej do tabeli *Summary of DMA1 requests for each channel*. Chcemy wyzwałać DMA update-eventami licznika TIM3. Znajdujemy więc sygnał TIM3_UP (nazwę trzeba sobie rozkminić samemu) i już wiemy, że może on wyzwałać kanał trzeci kontrolera DMA1. Jak zjedziemy jeszcze niżej to znajdziemy analogiczny diagram i tabelkę dla kontrolera DMA2.

Czego się spodziewamy po „zakończeniu programu”? Licznik TIM2 zlicza milisekundy. TIM3 co 250ms żąda, aby DMA zapisło stan TIM2 w tablicy wyniki. W sumie mamy cztery zapisy, czyli spodziewamy się, że kolejne wartości tablicy to będzie: 250, 500, 750, 1000. Ewentualnie jeden impuls w tę czy we w tę :) Dla niedowiarków screen-fociszon z Eclipse'a:

Name	Type	Value
wyniki	volatile uint32_t [4]	0x20000400 <wyniki>
(*)= wyniki[0]	volatile uint32_t	250
(*)= wyniki[1]	volatile uint32_t	500
(*)= wyniki[2]	volatile uint32_t	750
(*)= wyniki[3]	volatile uint32_t	1000

Rys. 12.1 Wartości zmiennej *wyniki* odczytane w debuggerze

Tyle. Bolało? No bo jednak kodzimy bez biblioteki, na rejestrach... a to jest przecież trudne! Dokładniejsze (niż w RM) informacje o DMA w STM32F1 można znaleźć w nacie: AN2548 *Using the STM32F1x and STM32L1x DMA controller*.

Co warto zapamiętać z tego rozdziału?

- zapamiętuj co chcesz, bylebyś bez trwogi potrafił czerpać plony z dobrodziejstwa jakim jest DMA!
- najczęściej transfery DMA są wyzwalane żądaniami z układu periferyjnego
- dane modyfikowane przez DMA powinny być oznaczone jako ulotne (*volatile*)
- po zakończeniu transferów (wyzerowaniu CNDTR) kanał się wyłącza, ale bit włączający (EN) pozostaje ustawiony
- zmiana konfiguracji kanału jest możliwa tylko jeśli bit EN jest skasowany

12.3. DMA (F429)

To mój pierwszy raz z DMA w F429. Zabieram się do lektury RMa. Tobie też to sugeruję. Spróbuj jak najwięcej zrozumieć sam (nic nie ryzykujesz), potem porównamy wnioski.

Jest trochę inaczej. Przede wszystkim zmienia się nazewnictwo. W F429 są dwa kontrolery DMA mające po osiem strumieni (strumień to odpowiednik kanału z F103). Każdy strumień może mieć do ośmiu kanałów, czyli źródeł wyzwalania (IMHO myląca nazwa). Źródła wyzwalania opisane są w RMie w tabelkach: *DMA1 request mapping*, *DMA2 request mapping*. Przykładowo DMA1 strumień 1 może być wyzwalany przez jeden z pięciu kanałów:

- kanał nr 3: licznik TIM2 (UEV lub CH3)
- kanał nr 4: USART3_RX
- ...

Za wybór źródła wyzwalania (czyli kanału dla strumienia) odpowiadają bity rejestru konfiguracyjnego DMA_SxCR_CHSEL (x to numer strumienia).

Druga nowość to kolejka FIFO (*first in, first out*). Każdy strumień ma swoją kolejkę o długości 4 słów. W F103 było tak, że dana odczytana ze źródła od razu była zapisywana w miejscu docelowym. Ilość odczytów i zapisów była zawsze równa. Jeśli wielkości danych się nie równały, to wartość była obcinana lub uzupełniania zerami. W F429 tak nie jest ze względu na obecność FIFO. FIFO to taki bufor pomiędzy odczytem ze źródła a zapisem do celu. Przynosi trzy korzyści:

- redukuje ilość operacji zapisu/odczytu (ilość dostępów do pamięci)
- umożliwia łączenie/dzielenie danych
- umożliwia realizację trybu *burst*

Załóżmy, że po stronie źródła odczytujemy dane wielkości 1B a docelowo chcemy zapisać 4B. W F103 każdy odczyt 1B wiążałby się z zapisem wielkości 4B (składającej się z odczytanego bajtu uzupełnionego zerami). W F429 działa to tak, że kolejne odczytane bajty są zapisywane w kolejce FIFO. Gdy uzbiera się z tego słowo (4B) to dopiero zostanie ono wysłane z FIFO do miejsca docelowego¹⁸⁷. Nie ma przy tym żadnego uzupełniania zerami! Cztery bajty składają się razem na całe słowo. Mechanizm działa też w drugą stronę. Tzn. można odczytać słowo ze źródła (jeden odczyt) i zapisywać po bajcie do celu (4 zapisy).

O tym kiedy dane zgromadzone w buforze zaczyną być zapisywane do miejsca docelowego, decyduje konfigurowalny próg (*threshold*, bity DMA_SxFCR_FTH). Do wyboru są cztery progi, określające przy jakim stopniu zapełnienia kolejki, ma nastepować zapis do celu ($\frac{1}{4}$, $\frac{1}{2}$, $\frac{3}{4}$, full).

Domyślnie FIFO jest wyłączone i DMA pracuje w trybie bezpośrednim - *direct mode* (patrz bit DMA_SxFCR_DMDIS). DMA zachowuje się wtedy w sposób zbliżony do wersji z F103, każdemu odczytowi odpowiada jeden zapis. *Direct mode* pociąga za sobą trzy ograniczenia:

- w *direct mode* rozmiary danych źródłowych i docelowych muszą być równe (brany jest pod uwagę rozmiar podany w polu DMA_SxCR_PSIZE)
- *direct mode* nie łączy się z trybem przesyłania danych z pamięci do pamięci
- *direct mode* nie łączy się z *burst mode*

Obecność bufora FIFO powoduje, że ilość operacji odczytu nie musi być równa ilości operacji zapisu. I teraz ważna sprawa! Jednym z rejestrów konfiguracyjnych DMA jest rejestr DMA_SxNDTR określający ilość transakcji (analogia do CNDTR w F103). Pojawia się więc pytanie: ilość których operacji określa rejestr DMA_SxNDTR (odczytu czy zapisu)? Śpieszę z odpowiedzią: określa ilość operacji wykonanych po stronie układu peryferyjnego.

Uwaga! Rozmiary danych odczytywanych, zapisywanych i ilość transferów należy dobierać z głową. Np. jeśli spłodzimy coś takiego:

- rozmiar odczytywany z peryferiala: 1B
- rozmiar zapisywany do pamięci: 4B
- ilość transferów z peryferiala: 6

¹⁸⁷ do tego dochodzi jeszcze konfigurowalny próg, ale o tym za chwilkę

to wyjdzie bzdura. Zostaną wykonane 4 odczyty po 1B, z których złoży się jedno słowo do zapisania. Następnie zostaną wykonane 2 pozostałe odczyty po 1B i nijak się z tego nie ulepi całego słowa! Patrz tabelka w RM: *Restriction on NDT versus PSIZE and MSIZE*.

Dostępne są trzy tryby pracy DMA (wybierane poprzez DMA_SxCR_DIR):

- *Peripheral to Memory*: żądanie z peryferiala powoduje odczytywanie danych ze źródła (peryferiala) do FIFO¹⁸⁸, po przekroczeniu wybranego progu (*threshold*) następuje opróżnienie FIFO do miejsca docelowego
- *Memory to Peripheral*: po włączeniu strumienia następuje (od razu) odczyt pamięci do całkowitego zapełnienia FIFO¹⁸⁸. Po wystąpieniu żądania ze strony układu peryferyjnego, zawartość FIFO jest wysyłana do celu. Gdy stopień zapełnienia FIFO zjedzie do poziomu ustawionego *thresholdu* (lub poniżej) kolejka jest dopełniana nowymi danymi z pamięci.
- *Memory to Memory*: włączenie kanału powoduje zapełnianie bufora FIFO. Po przekroczeniu progu zawartość jest wysyłana do celu. Tryb nie współpracuje z trybami *circular* oraz *direct mode*, ponadto jest dostępny **jedynie w kontrolerze DMA2!**

Transmisja się kończy gdy zaistnieje jeden z warunków:

- wyzeruje się rejestr SMA_SxNDTR (inaczej niż w F103, automatycznie następuje wtedy skasowanie bitu włączającego strumień: DMA_SxCR_EN)
- bit DMA_SxCR_EN zostanie skasowany programowo
- jeśli wybrano opcję *Peripheral Flow Control* (DMA_SxCR_PFCTRL) i układ peryferyjny zakończy transmisję *Memory-to-Peripheral* lub *Peripheral-to-Memory* (tylko układ SDIO to potrafi)

Uwaga! Jeżeli po wyłączeniu strumienia (wyzerowanie bitu EN przez program) w buforze FIFO znajdują jakieś dane, to strumień pozostanie włączony aż do całkowitego opróżnienia kolejki FIFO. Na czas opróżniania kolejki, bit włączający strumień pozostanie ustawiony. Po opróżnieniu kolejki bit zostanie skasowany automatycznie. Jeżeli w programie musimy mieć pewność, że strumień zakończył pracę, to po skasowaniu bitu EN, należy poczekać na faktyczne wyzerowanie tego bitu.

Drugi bajer (po FIFO) to podwójne buforowanie (*double buffer*, DMA_SxCR_DBM). Podwójne buforowanie polega na tym, że jeden strumień DMA ma możliwość pracy na dwóch buforach w pamięci (adresy podane w rejestrach DMA_SxM0AR i DMA_SxM1AR). I teraz: po uruchomieniu strumienia pracuje on na buforze wybranym za pomocą bitu DMA_SxCR_CT.

188 oczywiście jeśli nie jest włączony tryb *bezpośredni* (*direct mode*)

Załóżmy, że np. zapisuje próbki z ADC do pierwszego bufora. Gdy ten bufor się zapełni, następuje automatyczna zmiana bufora (zmienia się również stan bitu DMA_SxCR_CT). Teraz próbki lądują w drugim buforze. Program główny może modyfikować zawartość i adres (w konfiguracji DMA) nieużywanego aktualnie bufora bez wyłączania strumienia. Genialne w swojej prostocie. Włączenie podwójnego buforowania automatycznie wymusza tryb kołowy.

Trzeci bajer to *burst mode* (patrz DMA_SxCR_MBURST i DMA_SxCR_PBURST). Uprzedzam: nie czuję się pewnie w tym temacie... *Burst mode*¹⁸⁹ to zapis/odczyt kilku danych od razu. Tzn. że jedno żądanie DMA nie wymusza jednej transakcji tylko całą serię. Do wyboru mamy 4, 8 lub 16 transakcji. Np. jeśli skonfigurujemy tryb seryjny po stronie peryferiala:

- DMA_SxCR_PSIZE = 2B
- DMA_SxCR_PBURST = 8

To po wyzwoleniu transferu, DMA od razu (seryjnie) wykona 8 przesyłów danych 2B z/do FIFO. Przypuszczam, że główną zaletą jest tu szybkość i niepodzielność takiego zapisu. Jeśli w trybie *burst* do zapisania są np. 4 słowa to kontroler DMA nie zwolni magistrali danych (np. jeśli potrzebuje jej CPU) do czasu zakończenia całej transmisji *burst*. Z drugiej strony trzeba się pilnować i konfigurować to wszystko z głową:

- jeśli seryjnie zapisujemy dane z FIFO to musimy się postarać, aby tych danych w FIFO wystarczyło na całą serię
- jeśli seryjnie odczytujemy dane do FIFO, to musimy się postarać aby się pomieściły

Np. taka konfiguracja (zapisywanie danych z FIFO do pamięci):

- próg FIFO ½
- zapis danych wielkości 4B
- tryb *burst* 4

jest bez sensu! Bo w kolejce są 2 słowa (½ FIFO). A *burst* będzie próbował zapisać naraz 4 „serie” po 4B (4 słowa). W buforze (kolejce) nie będzie tylu danych! Ale jeśli zmienimy założenia - np. rozmiar zapisywanych danych ustawimy na pół-słowa (2B) to już będzie ok. W buforze będą 2 słowa, do przesłania będą 4*½ słowa, czyli też dwa słowa. Ładnie podsumowuje to tabelka: *FIFO threshold configurations*. Ograniczeń przy korzystaniu z trybu *burst* jest niestety trochę więcej, podsumujmy w skrócie najważniejsze:

189 jak przetłumaczyć *burst mode*? Może być *tryb seryjny*?

- zapis w trybie *burst* nie może przekroczyć 1kB address boundary¹⁹⁰
- liczba serii *burst* pomnożona przez wielkość danej nie może być większa od pojemności FIFO (to akurat jest logiczne i poniekąd już było omówione)

Ponadto, jeśli:

- NDTR nie będzie wielokrotnością iloczynu liczby serii i rozmiaru danej, lub
- zawartość FIFO nie będzie wielokrotnością iloczynu liczby serii i rozmiaru danej

to ostatnie transfery zostaną wykonane w trybie *single (nie-burst)* mimo konfiguracji strumienia do pracy w trybie *burst*.

Tryb kołowy (DMA_SxCR_CIRC) działa podobnie jak w poprzednim omawianym mikrokontrolerze. Powoduje automatyczne odtworzenie zawartości rejestru NDTR i pierwotnych adresów jeśli była włączona inkrementacja (DMA_SxCR_MINC, DMA_SxCR_PINC). W trybie kołowym, jeśli korzystamy z *burst mode*, spełnione muszą być dwa dodatkowe warunki:

$$NDTR = \text{wielokrotność } (MBURST \cdot \frac{MSIZE}{PSIZE})$$

$$NDTR = \text{wielokrotność } (PBURST \cdot PSIZE)$$

Na koniec zostały flagi przerwań i obsługa błędów. Flagi podzielone są na dwa rejesty (tylko do odczytu):

- dolny rejestr flag DMA_LISR - flagi dla strumieni 0 - 3
- górny rejestr flag DMA_HISR - flagi dla strumieni 4 - 7

Do kasowania służą analogiczne rejesty (tylko do zapisu):

- dolny rejestr kasowania flag DMA_LIFCR - kasowanie flag dla strumieni 0 - 3
- górny rejestr kasowania flag DMA_HIFCR - kasowanie flag dla strumieni 4 - 7

190 przepraszam, ale nie wiem jak to zgrabnie przetłumaczyć...

A co do samych flag (każda może generować przerwanie):

- flaga zakończenia transmisji¹⁹¹ DMA_xISR_TCIF
- flaga połowy transmisji DMA_xISR_HTIF
- flaga błędu transferu DMA_xISR_TEIF, ustawiana gdy wystąpi:
 - błęd szyny danych¹⁹²
 - próba modyfikacji rejestru adresu aktualnie używanego bufora w trybie *double buffered*
- flaga błędu trybu *direct mode* DMA_xISR_DMEIF, ustawiana:
 - tylko w *direct mode, peripheral to memory*, bez inkrementacji adresu w pamięci - flaga jest ustawiona jeśli pojawi się żądanie z peryferiala a poprzednia „dana” nie została jeszcze przesłana
- flaga błędu kolejki FIFO DMA_xISR_FEI, ustawiana gdy wystąpi:
 - FIFO *underrun* (zabrakło danych np. w trybie *burst*)
 - FIFO *overrun* (FIFO się zapchało)
 - włączenie kanału jeśli próg FIFO nie współgra z rozmiarem *bursta* (patrz tabela: *FIFO threshold configurations*)¹⁹²

Pozostałe uwagi nie godne osobnych akapitów:

- podobnie jak poprzednio dostępna jest automatyczna inkrementacja adresów
- bit DMA_SxCR_PINCOS pozwala wymusić inkrementację adresu o 4 bez względu na zaprogramowany rozmiar przesyłanych danych (dotyczy tylko strony peryferyjnej - DMA_SxPAR)
- każdy strumień ma konfigurowalny priorytet (DMA_SxCR_PL)
- aktualny stan zapełnienia bufora FIFO można sprawdzić poprzez bity DMA_SxFIFO_FS
- zmiana adresów buforów w trybie podwójnego buforowania, powinna być wykonana zaraz po ustawieniu się flagi końca transmisji (TCIF), chodzi o to żeby zdążyć ze zmianą adresu przed kolejną zmianą bufora
- gdy DMA zakończy działanie (NDTR zjedzie do zera) to do ponownego wystartowania z tymi samymi ustawieniami wystarczy ponownie ustawić bit włączający strumień (EN)
- najpierw należy włączyć DMA, a potem peryferial z nim współpracujący (wyzwalający)

191 w trybie *double buffered* jest ustawiana co zmianę buforu

192 powoduje automatyczne wyłączenie strumienia (skasowanie DMA_SxCR_EN)

- najpierw należy wyłączyć DMA (i poczekać aż EN=0), potem wyłączyć współpracujący z nim periferial
- przed włączeniem strumienia należy się upewnić, że wszystkie flagi przerwań są skasowane (albo profilaktycznie je skasować)
- dodatkowe informacje, w tym obliczenia czasów oczekiwania i przesyłania danych są dostępne w nacie: AN4031 *Using the STM32F2 and STM32F4 DMA controller*

Zadanie domowe 12.4: napisać program, który za pomocą DMA kopiuje blok pamięci (np. tablicę znaków) w inne miejsce pamięci SRAM. Po zakończeniu kopiowania oba bloki są porównywane i w zależności od wyniku porównania zapalana jest jedna lub druga dioda świecąca.

Przykładowe rozwiązanie (F429, diody na PG13 i PG14):

```

1. #define dma_en_bb BB(DMA2_Stream0->CR, DMA_SxCR_EN)
2.
3. int main(void) {
4.
5.     static char bufor1[] = "Ala ma kota, a sierotka Ma-ryśia.";
6.     static char bufor2[50];
7.     size_t size = sizeof(bufor1);
8.
9.     RCC->AHB1ENR |= RCC_AHB1ENR_DMA2EN | RCC_AHB1ENR_GPIOGEN;
10.    __DSB();
11.
12.    gpio_pin_cfg(GPIOG, PG13, gpio_mode_out_PP_LS);
13.    gpio_pin_cfg(GPIOG, PG14, gpio_mode_out_PP_LS);
14.
15.    DMA2_Stream0->PAR = (uint32_t)bufor1;
16.    DMA2_Stream0->M0AR = (uint32_t)bufor2;
17.    DMA2_Stream0->NDTR = size;
18.    DMA2_Stream0->CR = DMA_SxCR_MINC | DMA_SxCR_PINC | DMA_SxCR_DIR_1;
19.    dma_en_bb = 1;
20.
21.    while ( ! (DMA2->LISR & DMA_LISR_TCIF0) );
22.
23.    if ( memcmp(bufor1, bufor2, size) ) BB(GPIOG->ODR, PG14) = 1;
24.    else BB(GPIOG->ODR, PG13) = 1;
25.
26.    while (1);
27.
28. } /* main */

```

1) to tak dla urozmaicenia, żeby potem było mniej pisania (patrz linijka 19)

15 - 18) konfiguracja DMA: ustawione adresy buforów, liczba transferów, inkrementacje adresów i kierunek... praktycznie jak samo jak w zadaniu 12.1. Swoją drogą w dokumentacji jest informacja, że nie łączy się *memory to memory* z *direct mode*. Tryb bezpośredni jest domyślnie włączony, ja go nie wyłączyłem... a i tak jakoś to działa.

Zadanie domowe 12.5: jeden licznik coś sobie liczy (dla wygody będzie to zwykły timer), drugi timer co 250ms odpala DMA, które zapisuje aktualną wartość pierwszego licznika w buforze w pamięci SRAM. I tak cztery razy. Timery start! (tak samo jak w zadaniu 12.3, tylko mikrokontroler inny)

Przykładowe rozwiązanie (F429):

```
1. int main(void) {
2.
3.     static volatile uint16_t wyniki[4];
4.
5.     RCC->AHB1ENR |= RCC_AHB1ENR_DMA1EN;
6.     RCC->APB1ENR = RCC_APB1ENR_TIM2EN | RCC_APB1ENR_TIM3EN;
7.     __DSB();
8.
9.     TIM2->PSC = 16000-1;
10.    TIM2->ARR = UINT16_MAX;
11.    TIM2->EGR = TIM_EGR_UG;
12.
13.    TIM3->PSC = 16000-1;
14.    TIM3->ARR = 250-1;
15.    TIM3->EGR = TIM_EGR_UG;
16.    __DSB();
17.    TIM3->DIER = TIM_DIER_UDE;
18.
19.    DMA1_Stream2->PAR = (uint32_t)&TIM2->CNT;
20.    DMA1_Stream2->M0AR = (uint32_t)wyniki;
21.    DMA1_Stream2->NDTR = 4;
22.    DMA1_Stream2->FCR = DMA_SxFCR_DMDIS | DMA_SxFCR_FTH_0;
23.    DMA1_Stream2->CR = DMA_SxCR_CHSEL_2 | DMA_SxCR_CHSEL_0 | DMA_SxCR_MBURST_0 |
24.        DMA_SxCR_MSIZE_0 | DMA_SxCR_PSIZE_0 | DMA_SxCR_MINC | DMA_SxCR_EN;
25.
26.    TIM2->CR1 = TIM_CR1_CEN;
27.    TIM3->CR1 = TIM_CR1_CEN;
28.
29.    while (1);
30.
31. } /* main */
```

Program niewiele różni się w porównaniu z wersją dla F103:

3) tablica na wyniki przesłane przez DMA

5 - 7) włączenie zegarów (DMA, TIM2, TIM3)

9 - 11) konfiguracja timera TIM2: domyślna częstotliwość z jaką działa mikrokontroler to 16MHz, nastawa preskalera powoduje, że licznik zlicza z częstotliwością 1ms. Wymuszenie UEV powoduje wpisanie nowej wartości preskalera z rejestru buforowego.

13 - 17) konfiguracja drugiego licznika (wyzwalającego DMA), tak aby generował żądania co 250ms. Uwaga! Po programowym wymuszeniu UEV, a przed włączeniem generowania żądań DMA, dodano krótkie opóźnienie (zrealizowane poprzez instrukcję barierową). Bez tego UEV wymuszony bitem UG (linia 15) powodował wygenerowanie pierwszego żądania DMA już na etapie konfiguracji licznika! W efekcie pierwszy przesył DMA następował od razu po włączeniu strumienia i pierwsza wartość zapisana w tablicy wynosiła 0 (a kolejne 250, 500, 750). Dodanie tego opóźnienia rozwiązało problem.

19 - 24) konfiguracja DMA:

- przesyły z rejestru TIM2_CNT do tablicy w pamięci SRAM
- liczba transakcji: 4
- wyłączony tryb bezpośredni (włączone FIFO, dla urozmaicenia)
- adresy danych po stronie źródłowej i docelowej ustawione na 16b
- próg FIFO ustawiony na $\frac{1}{2}$ (czyli dopiero po zapełnieniu połowy kolejki danymi z licznika, zostanie wykonany zapis do pamięci SRAM)
- kanał strumienia wybrany na podstawie tabeli *DMA1 request mapping* (TIM3_UP)
- włączony tryb *burst* (4 serie)
- włączona inkrementacja adresu w pamięci (aby kolejne wyniki były zapisywane na kolejnych pozycjach tablicy)

Jeszcze w kwestii kolejki FIFO, trybu *burst* i ilości przesyłów. Nie ma „potrzeby” korzystania z tych funkcji. W przykładzie zostały wykorzystane dla urozmaicenia. Wspominałem o tym, że wszystko należy konfigurować z głową, rozpatrzmy ten przykładowy kod:

- DMA na każde żądanie z licznika TIM3 odczytuje zawartość rejestru TIM2_CNT (2B)
- wyniki są zapisywane do kolejki FIFO
- w sumie mają być cztery przesyły ($4*2B = 8B$)
- pojemność FIFO wynosi 4 słowa (słowo ma 4B), czyli $4*4B = 16B$
- próg FIFO jest ustawiony na $\frac{1}{2}$ (8B)
- próg FIFO zostanie osiągnięty po czterech odczytach z TIM2_CNT (w kolejce będzie wtedy 8B danych)
- tryb *burst* ustawiony jest na cztery serie po 2B
- po osiągnięciu progu FIFO, zostanie uruchomione przesyłanie danych do pamięci SRAM
- cztery serie (*burst*), po 2B (rozmiar danych po stronie pamięci) to w sumie 8B danych, które **muszą** być dostępne
- w FIFO jest 8B, przesłane będzie 8B... wszystko się zgadza!

Co warto zapamiętać z tego rozdziału?

- w F103 były kanały DMA; w F429 są strumienie DMA, zaś termin *kanał* odnosi się do źródła wyzwalania strumienia
- w celu ponownego uruchomienia strumienia, bez zmian ustawień, wystarczy ponownie ustawić bit EN
- w F429, gdy rozmiary danych źródłowych i docelowych są różne, nie występuje obcinanie lub uzupełnianie zerami jak w F103
- DMA Twoim przyjacielem!

12.4. DMA (F334)

Mikrokontroler F334 posiada tylko jeden kontroler DMA obsługujący do siedmiu kanałów. Moduł DMA, pod względem użytkowym, jest taki sam jak w mikrokontrolerze F103. Oczywiście, z uwagi na inne peryferia mikrokontrolera, różnią się źródła wyzwalania kanałów. No ale to chyba nie budzi wątpliwości. Nie ma co przedłużać.

Zadanie domowe 12.6: timer przepelnia się co 1ms. Każde przepelenie powoduje inkrementację zmiennej (np. 8-mio bitowej) oraz wyzwala przesłanie wartości tej zmiennej na port I/O (do rejestru wyjściowego). Suma summarum na wyjściu ma powstać licznik binarny.

Przykładowe rozwiązanie (F334):

```
1. static volatile uint8_t value;
2.
3. int main(void){
4.
5.     RCC->AHBENR = RCC_AHBENR_GPIOCEN | RCC_AHBENR_DMA1EN;
6.     RCC->APB2ENR = RCC_APB2ENR_TIM17EN | RCC_APB2ENR_SYSCFGEN;
7.
8.     gpio_pin_cfg(GPIOC, PC0, gpio_mode_out_PP_LS);
9.     gpio_pin_cfg(GPIOC, PC1, gpio_mode_out_PP_LS);
10.    gpio_pin_cfg(GPIOC, PC2, gpio_mode_out_PP_LS);
11.    gpio_pin_cfg(GPIOC, PC3, gpio_mode_out_PP_LS);
12.    gpio_pin_cfg(GPIOC, PC4, gpio_mode_out_PP_LS);
13.    gpio_pin_cfg(GPIOC, PC5, gpio_mode_out_PP_LS);
14.    gpio_pin_cfg(GPIOC, PC6, gpio_mode_out_PP_LS);
15.    gpio_pin_cfg(GPIOC, PC7, gpio_mode_out_PP_LS);
16.
17.    TIM17->PSC = 400-1;
18.    TIM17->EGR = TIM_EGR_UG;
19.    TIM17->ARR = 20-1;
20.    TIM17->DIER = TIM_DIER_UDE | TIM_DIER_UIE;
21.    TIM17->CR1 = TIM_CR1_CEN;
22.
23.    DMA1_Channel7->CMAR = (uint32_t)&value;
24.    DMA1_Channel7->CPAR = (uint32_t)&GPIOC->ODR;
25.    DMA1_Channel7->CNDTR = 1;
26.    DMA1_Channel7->CCR = DMA_CCR_CIRC | DMA_CCR_DIR | DMA_CCR_EN;
27.
28.    SYSCFG->CFG_R1 = SYSCFG_CFG_R1_TIM17_DMA_RMP;
29.
30.    NVIC_EnableIRQ(TIM17_IRQn);
31.    while(1);
32.
33. }
34.
35. void TIM17_IRQHandler(void){
36.     if(TIM17->SR & TIM_SR UIF){
37.         TIM17->SR &= ~TIM_SR UIF;
38.         value++;
39.     }
40. }
```

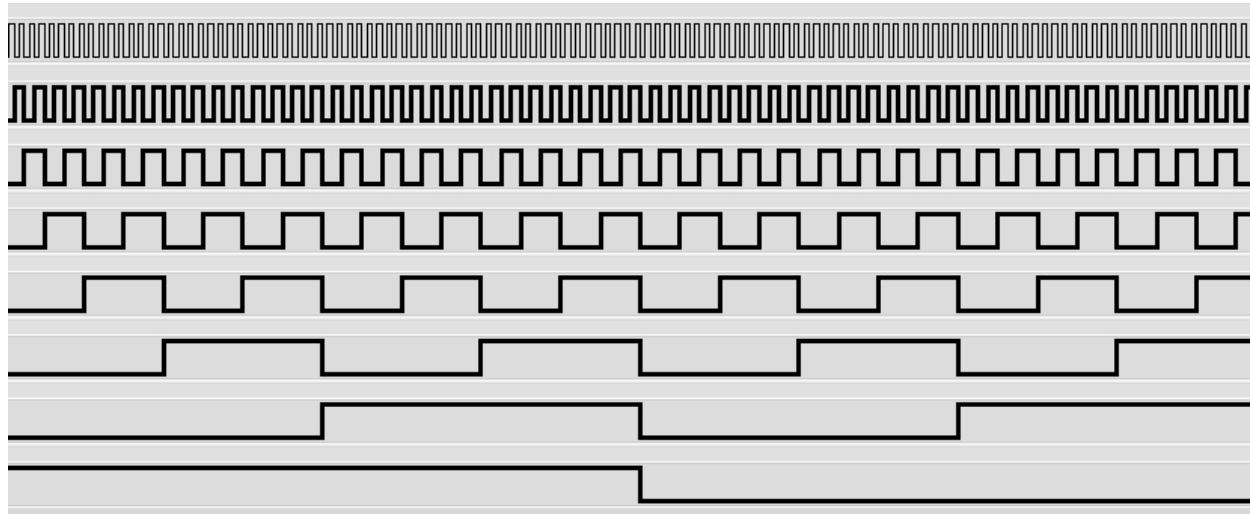
Programik jest prościutki. Licznik TIM17 przepętlia się co 1ms (konfiguracja z linijk 17-21). Każde przepelenie generuje przerwanie i żądanie DMA. W 18 linijce wymuszono *update event* aby od razu załadowana została nowa wartość rejestru preskalera (rejestr jest buforowany, patrz rozdział 8.3). W przerwaniu licznika (linijki 35-40) inkrementowana jest zmienna o nazwie *value*. Prościzna.

Konfiguracja *DMA* też nie zawiera niczego odkrywczego. Dane są przesyłane ze zmiennej *value* do rejestru wyjściowego portu C. Włączony jest tryb kołowy, adresy nie są inkrementowane, więc wartość CNDTR nie ma większego znaczenia. Byleby nie zero, bo wtedy kanał *dma* się wyłącza :)

Nie bez powodu wybrałem akurat licznik 17 i kanał 7. Niektóre źródła wyzwalania przesyłów *DMA* wymagają dodatkowej konfiguracji w rejestrze *SYSCFG_CFGRx*. Przykładem jest właśnie np. para TIM17 i DMA kanał 7. Domyślnie, bez ustawionego bitu *SYSCFG_CFG_R1_TIM17_DMA_RMP*, licznik 17 wyzwala kanał pierwszy kontrolera DMA. Jeżeli chcemy, aby licznik współpracował z kanałem 7, to należy wspomniany bit ustawić. Informacje na

temat wymaganych konfiguracji różnych źródeł żądań DMA można znaleźć w rozdziale *DMA request mapping* w RM. Źródła wymagające „specjalnej troski” ozdobione są przypisem dolnym odsyłającym do rejestrów *SYSCFG_CFGRx*.

A dla wzrokowców, zrzucik z działania mojego rozwiązania:



Rys. 12.2 Licznik binarny (*od góry*: PC0, PC1, PC2, ..., PC6, PC7)

Co warto zapamiętać z tego rozdziału?

- DMA w F334 działa tak samo jak w F103
- niektóre źródła żądań DMA wymagają dodatkowej konfiguracji w SYSCFG_CFGRx

13. PRZETWORNIK ADC („*SUPERFLUA NON NOCENT*”¹⁹³)

13.1. ADC wstęp (F103)

Tego tematu boję się prawie jak liczników :) Tutaj również jest sporo trybów, z czego liczna część dosyć egzotyczna. Dodatkowo mam wrażenie, że osoby piszące rozdział RMa dotyczący ADC założyły się z podobnym zespołem opisującym DAC o to, kto wymyśli więcej dziwacznych nazw na trywialne rzeczy. Nie wiem czy to wynika z moich braków i jakiś błędów w *podejściu* ale rozdział o ADC (w RMie) znajduję jako bardzo nieprzyjazny.

Zacznijmy od szczypty teorii. Podstawowe *marketingowe* parametry przetwornika ADC, na podstawie RMa:

- przetworniki o rozdzielczości 12 bit
- do 18-stu multipleksowanych kanałów (w tym dwa zarezerwowane na czujnik temperatury i wewnętrzne źródło napięcia odniesienia)
- „analogowy” *watchdog*¹⁹⁴ (AWD)
- częstotliwość pracy przetworników ADC do 14MHz
- minimalny czas konwersji $1\mu\text{s}$
- funkcja automatycznej samo-kalibracji
- możliwość wspólnej pracy kilku przetworników (*dual mode*)

W datasheetie w tabeli *device overview* dowiadujemy się ponadto, że nasz ulubiony mikrokontroler posiada dokładnie 3 przetworniki ADC i 16 multipleksowanych kanałów zewnętrznych (+ te dwa zarezerwowane na czujnik temperatury i napięcie odniesienia). Czyli inaczej mówiąc: STM ma trzy odrębne przetworniki analogowo cyfrowe. Przetworniki mierzą napięcie na wybranych wejściach analogowych (kanałach). Kanałów zewnętrznych (nóżek, które mogą być wejściami analogowymi) jest w sumie 16. Teraz to wszystko wydaje mi się proste i nie warte wyjaśnień, ale nie tak dawno temu różnica między ilością kanałów a przetworników nie była dla mnie taka oczywista... o_O

Zatrzymajmy się tutaj na chwilę i zobaczymy jak są oznaczane nóżki mikrokontrolera związane z przetwornikiem ADC. A więc: datasheet → tabela *Pin definitions* i znajdujemy takie oto trzy kwiatki:

- ADC123_IN10
- ADC3_IN4
- ADC12_IN15

193 „*Nadmiar nie szkodzi.*”

194 przy czym dla mnie to on jest cyfrowy... ale co ja tam wiem, sam sobie ocenisz

Zasada oznaczania jest prosta: cyfry po „ADC” oznaczają numery przetworników, z którymi dana nóżka może współpracować. Liczba po „IN” oznacza numer kanału. Czyli np. pierwsza kropka to kanał dziesiąty przetworników ADC1, ADC2 i ADC3. Druga kropka to kanał 4 i działa tylko z przetwornikiem ADC3. Proste... teraz.

Dla odprężenia odrobina danych elektrycznych z datasheetu STM32F10x (poglądowo):

- napięcie na wejściu analogowym (V_{ain}) powinno zawierać się w przedziale:

$$V_{ref-} \leq V_{ain} \leq V_{ref+}$$

- ujemne napięcie odniesienia (V_{ref-})¹⁹⁵ powinno być połączone z masą analogową (V_{ssa})
- dodatnie napięcie odniesienia (V_{ref+})¹⁹⁵ powinno zawierać się w przedziale:

$$2,4 \text{ V} \leq V_{ref+} \leq V_{dda}$$

- prąd pobierany z wejścia dodatniego napięcia odniesienia wynosi maksymalnie: $220\mu\text{A}$
- prąd upływu pinu wejściowego: do $\pm 1\mu\text{A}$
- maksymalna rezystancja źródła mierzonego sygnału: $50\text{k}\Omega$

Nóżka V_{ref+} jest dostępna tylko w mikrokontrolerach w obudowach $>100\text{pin}$, w pozostałych przypadkach dodatnim napięciem odniesienia jest V_{dda} .

Wprowadźmy sobie nowe pojęcie - **grupę kanałów**. Grupa kanałów jest to zbiór wybranych (podczas konfiguracji) kanałów, które mają podlegać konwersji oraz kolejność tych konwersji. Mówiąc najprościej jak się da: wybieramy sobie które kanały po kolei mają być mierzone i ustawiamy je w rejestrach konfiguracyjnych grupy. Kanały mogą się mieszać, powtarzać itd. Przykładowo:

- 3 konwersje: IN1, IN2, IN3
- 4 konwersje: IN1, IN8, IN2, IN0
- 1 konwersja: IN8
- 6 konwersji: IN0, IN0, IN3, IN1, IN12, IN0

Kanały można konfigurować w ramach dwóch grup:

- grupa regularna¹⁹⁶ (*Regular Group*)
- grupa wstrzykiwana¹⁹⁷ (*Injected Group*)

¹⁹⁵ jeśli jest wyprowadzone w danej obudowie

¹⁹⁶ to tłumaczenie nie do końca oddaje sens nazwy... ale przynajmniej łatwo zapamiętać

¹⁹⁷ lub wtryskiwana... lub pieszczołwię strzykawkowa :)

I oczywiście obie grupy odrobinę się różnią. W ramach grupy regularnej można skonfigurować maksymalnie do 16 konwersji. Po każdej konwersji wynik ląduje we **wspólnym** rejestrze danych ADCx_DR. Jeśli program nie wyrobi się z odczytywaniem wyników, to kolejna konwersja nadpisze wynik poprzedniej! I nic nas o tym nie ostrzeże¹⁹⁸! Oczywiście można się wspomagać DMA i przerwaniami co całkowicie rozwiązuje problem :) O właśnie! Zakończenie konwersji kanału z grupy regularnej może generować żądanie DMA. Druga grupa (wstrzykiwana) nie ma takiej możliwości.

Grupa strzykawkowa może obejmować maksymalnie tylko 4 konwersje. W zamian za to, wynik każdej konwersji ląduje w osobnym rejestrze danych ADCx_JDRx. Ponadto grupa ta ma wyższy priorytet niż grupa regularna i jest raczej przeznaczona do wstrzykiwania niż do pracy ciągłej. Tzn. jeśli trwa konwersja w ramach grupy regularnej to można ją przerwać uruchamiając grupę strzykawkową¹⁹⁹. Efekt będzie taki, że konwersja kanałów grupy regularnej zostanie wstrzymana, przetwornik zajmie się kanałami grupy wstrzykiwanej a jak skończy to wróci do grupy zwyczajnej (czy tam regularnej jak ktoś woli). Ponadto można ustawić *offset* odejmowany automatycznie od wyników konwersji tej grupy (patrz rozdział 13.5).

Mały przykład żeby nie było zbyt abstrakcyjnie: mamy układ który stale wykonuje jakieś pomiary a raz na ruski rok (np. na żądanie operatora) mierzy napięcie baterii która go zasila. Aż się prosi aby stałe pomiary były w grupie regularnej, a pomiar baterii we wstrzykiowanej. Jak ktoś będzie chciał sprawdzić baterię to wystarczy aktywować pomiar grupy wstrzykiwanej. Przetwornik przerwie pomiary regularne, zmierzy napięcie baterii i wróci do swojej normalnej pracy. Bez żadnych zabaw w zmianę konfiguracji, zmienianie kanałów itd.. Łatwo, łatwo, prosto i przyjemnie :)

Jak prawie wszystko w STM, przetworniki ADC mogą generować żądania DMA i przerwania. Występują jednak przy tym drobne ograniczenia. W rejestrze statusowym ADCx_SR są następujące flagi:

- STRT - flaga rozpoczęcia konwersji kanałów z grupy regularnej
- JSTRT - flaga rozpoczęcia konwersji kanałów z grupy wstrzykiwanej
- JEOC - flaga zakończenia konwersji kanałów z grupy wstrzykiwanej
- EOC - flaga zakończenia konwersji kanałów z jednej z grup
- AWD - flaga „analogowego” watchdoga

198 mogli dorzucić jakąś flagę nadpisania wyników, nie?

199 pojawi się jeszcze wiele głupich nazw... wybacz - muszę mieć jakąś rozrywkę żeby nie usnąć

Uwaga pułapka! Flaga EOC jest z automatu kasowana podczas odczytu rejestru z wynikiem konwersji grupy regularnej (rejestru danych, ADCx_DR). Nawet jeśli ten odczyt jest dokonywany np. w formie podglądu w debuggerze!

Przerwania mogą generować tylko trzy ostatnie flagi (JEOC, EOC, AWD). Żądania DMA generowane są po zakończeniu konwersji w ramach grupy regularnej, przy czym generować je może tylko ADC1 i ADC3. ADC2 nie ma takiej możliwości, ale rekompensuje to pracą w trybie *dual mode*.

Jeszcze taka uwaga na koniec: wszystkie bity i rejesty związane z grupą wstrzykiwaną (*Injected*) mają w prefiksie wielkie Jot (nie Iii). Chodzi o to, aby nie myliło się z graficznie podobnymi znakami (np. z jedynką czy eLem)... przynajmniej tak to sobie tłumaczę.

Co warto zapamiętać z tego rozdziału?

- mikrokontroler posiada trzy 12-bitowe przetworniki ADC i 16 multipleksowanych kanałów wejściowych
- konwersje kanałów są konfigurowane w ramach grup: regularnej i wstrzykiwanej
- grupa regularna:
 - do 16 konwersji
 - wspólny rejestr danych
 - generowanie żądań DMA
- grupa wstrzykiwana:
 - do 4 konwersji
 - osobne rejesty danych
 - wyższy priorytet (przerywanie grupy regularnej)

13.2. Tryby pracy pojedynczego przetwornika ADC (F103)

Najprostszym trybem jaki można sobie wyobrazić jest taki w którym próbujemy tylko jeden kanał. Możemy przeprowadzić jedną konwersję lub włączyć ADC żeby sobie mielił w tle (bit ADCx_CR2_CONT) bez przerwy. Prawda że proste i logiczne? No to mamy z głowy już dwa tryby pracy przetwornika ADC:

- *Single Conversation Mode*
- *Continuous Conversation Mode*

To samo było w AVR. Jedyne urozmaicenie to to, że nasz kanał może być ustawiony w grupie regularnej lub truskawkowej (strzykawkowej). W zależności od grupy:

- wynik zapisze się w innym rejestrze (ADCx_DR lub ADCx_JDRy)
- *Single Conversation Mode* dla grupy regularnej może być uruchamiany programowo (bit ADCx_CR2_ADON) lub z trygierza zewnętrznego (licznik lub EXTI)
- grupa strzykawkowa może być uruchamiana tylko z trygierza (notabene jednym z trygierzy jest bit JSWSTART w rejestrze ADC_CR2... czyli da się odpalić programowo ustawiając ten bit)
- po zakończeniu konwersji wywoływane są różne zdarzenia (EOC, JEOC)

Oto i cała filozofia. Tych trybów można użyć np. do pomiaru napięcia baterii (ciągłego lub na zwołanie). No to wiadomo co nas czeka:

Zadanie domowe 13.1: pomiar napięcia na wybranym wejściu ADC w trybie *Single Conversation Mode*. Kanał należy skonfigurować jako regularny (*regular group*) i odpalić konwersję programowo.

Przykładowe rozwiążanie (F103, wejście analogowe PC0):

```
1. int main(void) {  
2.  
3.     RCC->APB2ENR = RCC_APB2ENR_IOPCEN | RCC_APB2ENR_ADC1EN;  
4.     gpio_pin_cfg(GPIOC, PC0, gpio_mode_input_analog);  
5.  
6.     ADC1->CR2 = ADC_CR2_ADON;  
7.     for(volatile uint32_t delay = 100000; delay; delay--);  
8.  
9.     ADC1->SQR3 = 10;  
10.    ADC1->CR2 = ADC_CR2_ADON;  
11.  
12.    while( !(ADC1->SR & ADC_SR_EOC) );  
13.        __BKPT();  
14.  
15. } /* main */
```

Włączenie zegarów i konfiguracja pinu, mam nadzieję, nie budzą Twoich wątpliwości. Jako wejście wybrałem sobie nóżkę PC0 (ADC123_IN10).

6) to jest ważna linijka! Po uruchomieniu mikrokontrolera, przetwornik ADC jest wyłączony (*power down mode*) a sygnalizuje to skasowany bit ADCx_CR2_ADON. Ustawienie bitu ADON po raz pierwszy (gdy wcześniej był skasowany) powoduje wybudzenie przetwornika. Przetwornik po obudzeniu potrzebuje chwilki aby być w pełni przytomnym²⁰⁰. Stąd w następnej linijce jest jakieś prymitywne opóźnienie. W każdej chwili można uśpić przetwornik poprzez skasowanie bitu ADON. Warto o tym pamiętać przed uśpieniem całego STMa, bo ADC pobiera energię jeśli jest wybudzony :)

200 według datasheeta „chwilka” (t_{stab}) trwa maksymalnie 1μs

9) w tej linijce następuje konfiguracja konwersji grupy regularnej. Konfiguracja grupy regularnej obejmuje trzy rejestrady ADCx_SQR1..3. W tych rejestrach jest w sumie szesnaście pól oznaczonych jako SQ1..SQ16. Każde pole SQn pozwala skonfigurować numer kanału²⁰¹, który będzie mierzony w n-tej konwersji sekwencji. Dodatkowo w rejestrze ADCx_SQR1 znajduje się czterobitowe pole, o wdzięcznej nazwie L, w którym należy ustawić sumaryczną liczbę konwersji w grupie. Czyli np. taka konfiguracja:

- L = 5
- SQ1 = 10
- SQ2 = 7
- SQ3 = 12
- SQ4 = 0
- SQ5 = 7
- SQ6 = 3
- SQ7 = 10
- ...

spowoduje, że przetwornik (pod warunkiem włączenia trybu wielokanałowego, patrz przykład 13.3) przeprowadzi 5 konwersji: IN10, IN7, IN12, IN0, IN7.

W omawianym przykładzie chcemy dokonać pomiaru tylko jednego kanału i nie korzystamy z trybu wielokanałowego (*scan mode*). Z tego względu konfiguracja ilości konwersji w grupie nie jest brana pod uwagę. Bez względu na zawartość pola ADCx_SQR1_L, wykonany będzie jeden pomiar kanału ustawionego w polu SQ1 (rejestr ADCx_SQR3). Ustawiamy kanał 10 i jedziemy dalej.

10) właściwie to już dojechaliśmy do końca. W linii 6. ustawiliśmy bit ADON po to, aby wybudzić przetwornik z trybu uśpienia (*power down*). Po wybudzeniu przetwornika, bit ADON pozostaje ustawiony. W każdej chwili można odczytać bit ADON aby sprawdzić czy przetwornik jest wybudzony lub skasować ten bit aby przetwornik uśpić.

Ponowne ustawienie (ustawionego już wcześniej) bitu ADON, powoduje programowe rozpoczęcie konwersji kanałów grupy regularnej. Mówiąc inaczej: rozpoczęcie konwersji następuje po ustawieniu bitu ADON gdy przetwornik jest wybudzony z trybu *power down*. Jest tylko jeden warunek: nie można w tej samej operacji zmieniać również innych bitów rejestrów ADCx_CR2. Chodzi o to, żeby nie uruchomić konwersji przez przypadek. To znaczy:

201 żeby nie było wątpliwości, np. ADC123_IN10 to kanał nr 10

- te linijki mogą uruchomić konwersję (ustawiają tylko bit ADON):
 - `ADC1->CR2 = ADC_CR2_ADON;`
 - `ADC1->CR2 |= ADC_CR2_ADON;`
 - `BB(ADC1->CR2, ADC_CR2_ADON) = 1;`
- takie linijki nie spowodują uruchomienia konwersji (modyfikują inne bity rejestru poza ADON):
 - `ADC1->CR2 = ADC_CR2_ADON | ADC_CR2_JEXTTRIG;`
 - `ADC1->CR2 |= ADC_CR2_ADON | ADC_CR2_DMA;`

No to właśnie uruchomiliśmy konwersję grupy regularnej :)

12) oczekiwanie na flagę EOC (*end of conversion*). Flaga jest ustawiana po zakończeniu **wszystkich** konwersji grupy regularnej (w naszym przykładzie jest tylko jedna konwersja) lub grupy strzykawkowej. Dodatkowo w przypadku tej drugiej, ustawiana jest flaga JEOC (*injected end of conversion*).

13) a to taka miła instrukcja (*breakpoint*), która powoduje zatrzymanie rdzenia jeśli jest podłączony debugger. Czyli program po dojściu do tego miejsca sam robi sobie Halt (tak jakbym nacisnął *pause* w Eclipse), a ja odczytuję wynik konwersji przez debugger (odczytałem: 0x7AC). Uwaga! Jeśli nie będzie podłączonego debugera to *breakpoint...* a sam sprawdź :) albo doczytaj (ale uprzedzam: trzeba się naszukać trochę... jak zawsze w STMach).

Do wejścia podłączyłem mniejszej więcej połowę napięcia zasilania. Przetwornik jest 12 bitowy. Połowa z 2^{12} to 2048. Szesnastkowo 0x800. Ja odczytałem 0x7AC. Działa²⁰²!

Zadanie domowe 13.2: pomiar napięcia na wybranym wejściu ADC w trybie *single conversation mode*. Kanał należy skonfigurować jako strzykawkowy. Konwersja ma zostać uruchomiona jednokrotnie wybranym licznikiem.

²⁰² przykłady mają być łatwe dla Ciebie i szybkie dla mnie, o poprawie dokładności porozmawiamy sobie innym razem

Przykładowe rozwiązanie (F103, wejście analogowe PC0):

```
1. int main(void) {
2.
3.     RCC->APB2ENR = RCC_APB2ENR_IOPCEN | RCC_APB2ENR_ADC1EN;
4.     RCC->APB1ENR = RCC_APB1ENR_TIM2EN;
5.     gpio_pin_cfg(GPIOC, PC0, gpio_mode_input_analog);
6.
7.     TIM2->PSC = 8000-1;
8.     TIM2->ARR = 1000-1;
9.     TIM2->EGR = TIM_EGR_UG;
10.    TIM2->CR2 = TIM_CR2_MMS_1;
11.
12.    ADC1->CR2 = ADC_CR2_ADON | ADC_CR2_JEXTTRIG | ADC_CR2_JEXTSEL_1;
13.    for(volatile uint32_t delay = 100000; delay; delay--);
14.    ADC1->JSQR = 10<<15;
15.
16.    TIM2->CR1 = TIM_CR1_CEN | TIM_CR1_OPM;
17.
18.    while( !(ADC1->SR & ADC_SR_JE0C) );
19.    __BKPT();
20.
21. } /* main */
```

Początek nie może budzić wątpliwości :) Jak coś to zalecam lekturę Poradnika od początku :)] Włączenie taktowania kilku bloków, konfiguracja wejścia IN10 i licznika. Licznik działa w trybie *master*. Po sekundzie ma się przekręcić i wysłać sygnał TRGO do przetwornika.

12) wybudzenie przetwornika z *power down* mamy już rozkminione. Drugi bit to włączenie wyzwalania zewnętrznym trygierzem. Ostatni bit to wybór trygierza. W tej materii udajemy się do RMa i odszukujemy rozdział *Conversion on external trigger*. Zwrót proszę uwagę na to, że różne przetworniki i różne grupy kanałów mają różne trygiery. Tak czy siak, odszukujemy nasz sygnał (TIM2_TRGO) i odczytujemy wartość JEXTSEL. Oczywiście zanim sobie założymy, że licznik TIMx będzie nam wzywał przetwornik ADCy jakimś tam sygnałem, dobrze jest upewnić się czy taka konfiguracja jest możliwa. Jak czegoś nie ma w tabelce to wiadomo - się nie da :)

13) o opóźnieniu po budzeniu ADC już pisałem

14) konfiguracja kanałów grupy wstrzykiwanej. Grupa strzykiwana to maksymalnie cztery konwersje. Cała konfiguracja mieści się w jednym rejestrze (ADCx_JSQR). Analogicznie jak poprzednio w grupie regularnej, mamy cztery pola na numery kanałów (JSQ1...JSQ4) i pole na sumaryczną ilość konwersji (JL). Uwaga! Sposób konfiguracji jest nieco fikušny. Otóż konwersje są przeprowadzane od pola SQx o numerze x = 4-JL w „góre”. Czyli:

Tabela 13.1 Kolejność konwersji grupy *wstrzykiwanej*

JL	ilość konwersji	kolejność konwersji
0	1	JSQ4
1	2	JSQ3 → JSQ4
2	3	JSQ2 → JSQ3 → JSQ4
3	4	JSQ1 → JSQ2 → JSQ3 → JSQ4

W naszym przykładzie, podobnie jak poprzednio, przeprowadzamy konwersję pojedynczego kanału, więc te ustawienia nas mało interesują... ale to już niedługo :)

16) włączam licznik, dodatkowo ustawiam opcję *one pulse mode* żeby się wyłączył po pierwszym UEV. Zwróć uwagę, że tym razem nie ma drugiego ustawiania bitu ADC_CR2_ADON. Tym razem pomiar wyzwalany jest zewnętrznym sygnałem (z licznika).

18) czekamy zakończenie konwersji i zatrzymujemy program celem odczytania wyniku. Odczytałem: 0x7AD. Na wejściu podane było to samo napięcie co w przykładzie z zadania 13.1.

To było dosyć proste prawda? A co jeśli mamy kilka kanałów do zmierzenia? Wtedy musimy:

- skonfigurować wybrane kanały w ramach wybranej grupy
- włączyć tryb wielokanałowy (o mylącej nazwie *scan mode*)

Scan mode powoduje, że wykonywane są wszystkie konwersje ustawione w konfiguracji grupy. Bez ustawionego bitu ADCx_CR1_SCAN konwertowany jest tylko jeden kanał bez względu na to ile konwersji ustawiono w rejestrze konfiguracyjnym grupy. Po zakończeniu konwersji wszystkich kanałów grupy, ustawiana jest odpowiednia flaga (EOC dla grupy regularnej lub EOC i JEOC dla grupy tryskawkowej). Jeśli dodatkowo włączony jest bit ADCx_CR2_CONT (*continuous*) to konwersja grupy regularnej się zapętli - taki *free running mode* znany z AVR. Przypominam, że wyniki konwersji kanałów grupy regularnej lądują we wspólnym rejestrze (ADCx_DR) i trzeba je na bieżąco odczytywać. Inaczej kolejne konwersje nadpiszą poprzedników. Używanie *scan mode* dla grupy regularnej praktycznie wymusza użycie DMA do odbierania wyników.

Zadanie domowe 13.3: jednokrotna konwersja dwóch kanałów grupy regularnej. Wyniki przesyłane do SRAMu (do dwuelementowej tablicy) z wykorzystaniem naszego ulubionego DMA.

Przykładowe rozwiązanie (F103, pomiar na wejściach PC0 i PC1):

```
1. int main(void) {
2.
3.     static volatile uint16_t wyniki[2];
4.
5.     RCC->APB2ENR = RCC_APB2ENR_IOPCEN | RCC_APB2ENR_ADC1EN;
6.     RCC->AHBENR |= RCC_AHBENR_DMA1EN;
7.     gpio_pin_cfg(GPIOC, PC0, gpio_mode_input_analog);
8.     gpio_pin_cfg(GPIOC, PC1, gpio_mode_input_analog);
9.
10.    DMA1_Channel1->CPAR = (uint32_t)&ADC1->DR;
11.    DMA1_Channel1->CMAR = (uint32_t)wyniki;
12.    DMA1_Channel1->CNDTR = 2;
13.    DMA1_Channel1->CCR = DMA_CCR1_MSIZE_0 | DMA_CCR1_PSIZE_1 | DMA_CCR1_MINC | DMA_CCR1_EN;
14.
15.    ADC1->CR2 = ADC_CR2_ADON | ADC_CR2_DMA;
16.    for(volatile uint32_t delay = 100000; delay; delay--);
17.    ADC1->CR1 = ADC_CR1_SCAN;
18.    ADC1->SQR3 = 10 | 11<<5;
19.    ADC1->SQR1 = ADC_SQR1_L_0;
20.
21.    BB(ADC1->CR2, ADC_CR2_ADON) = 1;
22.    while( !(ADC1->SR & ADC_SR_EOC) );
23.    __BKPT();
24.
25. } /* main */
```

3) tablica na wyniki dwóch konwersji, zwróć uwagę na typ danych!

5 - 8) nuda...

10 - 12) konfiguracja DMA. Dane przesyłamy z rejestru ADC1_DR do tablicy w SRAMie. Przesyły będą 2 bo wykonamy dwie konwersje (dwa kanały, jedna konwersja na kanał)

13) tutaj jest trochę magii z rozmiarami danych (dla urozmaicenia przykładu): po stronie peryferiala (ADC) 32b, po stronie pamięci 16b. Jeśli nie pamiętasz jak to działa to odsyłam do tabeli *Programmable data width & endian behavior* w RM. W skrócie: górną połową z 32b będzie wywalona i zostanie tylko dolne 16b. Górnny kawałek i tak nie jest nam potrzebny bo ADC jest 12bitowe i cały wynik mieści się w dolnych 16bitach... i jeszcze ma luz :) Po stronie pamięci włączam inkrementację. Inkrementacja po stronie peryferiala byłaby bez sensu, chcemy cały czas odczytywać dane z rejestru ADC1_DR. Na koniec włączam kanał. Od tej chwili jest on gotowy do działania, gdy tylko pojawi się żądanie DMA od układu ADC.

Zapewne już rozgryzłeś, mój pilny Czytelniku, dlaczego wybrałem akurat kanał 1 kontrolera DMA1? Podpowiedź: *Summary of DMA1 requests for each channel*.

15) wybudzenie ADC i włączenie trybu DMA (generowania żądań), potem małe opóźnienie

17) włączam tryb wielokanałowy (*scan*), aby przetwornik przeprowadził wszystkie konwersje ustalone w konfiguracji grupy

18, 19) konfiguracja grupy regularnej: w sumie będą dwie konwersje (kanały IN10, IN11)

21) włączam konwersję grupy regularnej poprzez ponowne ustawienie bitu ADON, dla urozmaicenia wykorzystuję bit banding. Na koniec czekam na zakończenie konwersji i przerwywam program²⁰³. W debuggerze odczytuję zawartość tablicy:

- wynik[0] = 1964
- wynik[1] = 2988

możesz wierzyć lub nie, ale z grubsza się zgadza (na IN10 jest około 1,6V; IN11 około 2,4V).

Na początku rozdziału wspominałem o tym, że grupa wstrzykiwana ma wyższy priorytet i może przerwać konwersję grupy regularnej. Przećwiczmy więc i taki scenariusz:

Zadanie domowe 13.4: niech przetwornik wykonuje (w kółko) konwersje dwóch kanałów w grupie regularnej. Wyniki niech będą wysyłane przez DMA do SRAMu, do tablicy dwuelementowej (właściwie trzy elementowej - zaraz się wyjaśni czemu). Ponadto niech co sekundę wykonywany będzie pomiar na innym kanale w ramach grupy wstrzykiwanej. Żeby dodatkowo urozmaicić zabawę, dodajmy... a co tam: wynik konwersji grupy wstrzykiwanej (po zakończeniu konwersji) ma być przepisywany z rejestru ADCx_JDRy do trzeciego elementu tablicy w SRAMie (tej samej tablicy, w której zapisywane są wyniki pomiarów grupy regularnej). I jak szaleć to szaleć - niech mikrokontroler miga diodą w rytm konwersji grupy strzykawkowej... i... niech kod wsadem się stanie!

203 „Oddaję głos do studia!” - jakoś tak mi się skojarzyło...

Przykładowe rozwiązań (F103, wejścia analogowe PC0, PC1, PC2, diod na PB0):

```
1. volatile uint16_t wyniki[3];
2.
3. int main(void) {
4.
5.     RCC->APB2ENR = RCC_APB2ENR_IOPCEN | RCC_APB2ENR_IOPBEN | RCC_APB2ENR_ADC1EN;
6.     RCC->AHBENR |= RCC_AHBENR_DMA1EN;
7.     RCC->APB1ENR = RCC_APB1ENR_TIM2EN;
8.
9.     gpio_pin_cfg(GPIOC, PC0, gpio_mode_input_analog);
10.    gpio_pin_cfg(GPIOC, PC1, gpio_mode_input_analog);
11.    gpio_pin_cfg(GPIOC, PC2, gpio_mode_input_analog);
12.    gpio_pin_cfg(GPIOB, PB0, gpio_mode_output_PP_2MHz);
13.
14.    TIM2->PSC = 8000-1;
15.    TIM2->ARR = 1000-1;
16.    TIM2->EGR = TIM_EGR_UG;
17.    TIM2->CR2 = TIM_CR2_MMS_1;
18.
19.    DMA1_Channel1->CPAR = (uint32_t)&ADC1->DR;
20.    DMA1_Channel1->CMAR = (uint32_t)wyniki;
21.    DMA1_Channel1->CNDTR = 2;
22.    DMA1_Channel1->CCR = DMA_CCR1_MSIZE_0 | DMA_CCR1_PSIZE_1 | DMA_CCR1_MINC | DMA_CCR1_EN | DMA_CCR1_CIRC;
23.
24.    ADC1->CR2 = ADC_CR2_ADON | ADC_CR2_DMA | ADC_CR2_JEXTTRIG | ADC_CR2_JEXTSEL_1 | ADC_CR2_CONT;
25.    ADC1->CR1 = ADC_CR1_SCAN | ADC_CR1_JEOCIE;
26.    for(volatile uint32_t delay = 100000; delay; delay--);
27.    ADC1->SQR3 = 10 | 11<<5;
28.    ADC1->SQR1 = ADC_SQR1_L_0;
29.    ADC1->JSQR = 12<<15;
30.
31.    NVIC_EnableIRQ(ADC1_2_IRQn);
32.    BB(ADC1->CR2, ADC_CR2_ADON) = 1;
33.    BB(TIM2->CR1, TIM_CR1_CEN) = 1;
34.
35.    while (1);
36.
37. } /* main */
38.
39. __attribute__ ((interrupt)) void ADC1_2_IRQHandler(void){
40.
41.     if ( ADC1->SR & ADC_SR_JEOC ){
42.         ADC1->SR = (~ADC_SR_JEOC)&0x1f;
43.         wyniki[2] = ADC1->JDR1;
44.         BB(GPIOB->ODR, PB0) ^= 1;
45.     }
46.
47. }
48.
```

Właściwie to wszystko już było, więc tak jakby nie ma co omawiać...

1) tablica na wyniki, globalna żeby była dostępna w przerwaniu

5 - 12) włączam taktowanie portów, DMA, TIM2, ADC oraz konfiguruję nóżki... i skrzydełka

14 - 17) konfiguracja licznika w trybie *master*, aby co 1s wysyłał sygnał TRGO który będzie wyzwał konwersję grupy wstrzykiwanej

19 - 23) konfiguracja DMA do przesyłu danych z ADC1_DR do tablicy. Włączony tryb kołowy: dzięki temu po przesłaniu dwóch wyników (zapełnieniu tablicy), wartość rejestru DMA_CNDTR się „odnowi” a inkrementowany wskaźnik zapisu po stronie pamięci „cofnie” się do pozycji początkowej. Jak ktoś się zgubił to wersja krok po kroczku:

- pierwsze żądanie DMA (pochodzące z ADC) spowoduje przesłanie wyniku konwersji kanału IN10 z ADC1_DR do *wyniki[0]*
- wartość licznika transakcji zmniejszy się o jeden (CNDTR = 1), a wskaźnik zapisu po stronie pamięci przesunie się na kolejną pozycję w tablicy (bo jest włączona inkrementacja adresu)
- drugie żądanie DMA (z ADC) spowoduje przesłanie wyniku konwersji kolejnego kanału (IN11) z ADC1_DR do *wyniki[1]*
- wartość licznika transakcji znowu zmniejszy się o jeden i tym samym wyzeruje (CNDTR = 0)
- gdyby tryb kołowy nie był włączony, to wyzerowanie CNDTR zablokowałoby dalsze działanie kanału DMA i nie reagowałby on na kolejne żądania z ADC
- włączony tryb kołowy powoduje przywrócenie wartości CNDTR (=2) i przywrócenie adresów które były inkrementowane
- trzecie żądanie DMA (z ADC) spowoduje przesłanie wyniku konwersji kanału IN10 z ADC1_DR do *wyniki[0]*
- ... zapętlaj

25) dłużańska linijka, ale nowości mało: wybudzenie ADC (było!), włączenie generowania żądań DMA (było!), włączenie i wybór sygnału trygierującego grupę strzykawkową (było!), włączenie trybu ciągłego (nie było!). Tryb ciągły (*continuous mode*) powoduje, że po wykonaniu wszystkich konwersji z grupy regularnej, przetwornik zaczyna od początku bez czekania na cokolwiek (no kto by się spodziewał).

27) włączony tryb wielokanałowy²⁰⁴ i przerwanie od flagi końca konwersji z grupy iniekcyjnej (flagi JEOC). Gdzieś już wspominałem, że grupa iniekcyjna nie może generować żądań DMA. W związku z tym, jedyną metodą pozwalającą odbierać (na bieżąco) wyniki konwersji tej grupy, jest wykorzystanie przerwań. Dalej jest prymitywne opóźnienie (po wybudzeniu ADC) i konfiguracja konwersji w dwóch grupach.

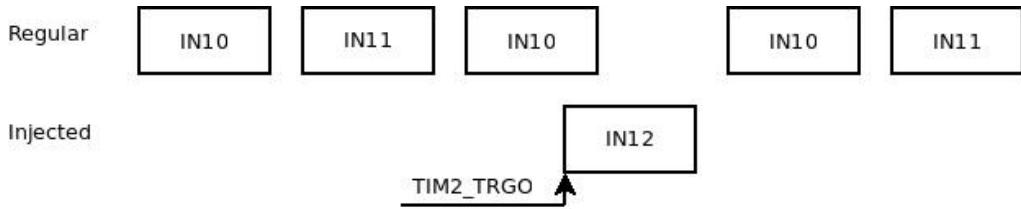
33 - 35) włączenie przerwania w NVICu, rozpoczęcie konwersji grupy regularnej i uruchomienie licznika TIM2

41 - 48) procedura obsługi przerwania: sprawdzenie źródła przerwania, skasowanie flagi, zapisanie wyniku, mignięcie diodą, koniec :)

Schemacik dla wzrokowców: po pojawienniu się sygnału TIM2_TRGO przerywana jest konwersja w ramach grupy regularnej (IN10, IN11, IN10, IN11, ...) i wykonywana jest konwersja grupy wstrzykiwanej (IN12). W przykładzie grupa wstrzykiwana obejmuje jedną konwersję, ale

²⁰⁴ tak sobie myślę, że nazwa *Multichannel Mode* byłaby bardziej adekwatna niż *Scan Mode*... ST chyba też tak myśli i w kilku dokumentach używa nazwy *Multichannel* zamiast *Scan...*

oczywiście może być ich więcej. Swoją drogą: w RMie to się nazywa *triggered injection* gdyby ktoś pytał :)



Rys. 13.1 Konwersja regularna przerwana przez konwersję wstrzykiwaną.

Dlaczego tryb ciągły (*continuous*) zadziałał tylko na grupę regularną, a nie spowodował ciągłego przetwarzania konwersji grupy strzykawowej? Dobre pytanie. W RMie nie jest to wyraźnie napisane (względnie przeoczyłem). Wydaje mi się, że generalnie grupa wstrzykiwana nie jest przeznaczona do „ciąglej pracy” tak jak grupa regularna, i stąd tryb ciągły dotyczy w domyśle tylko grupy regularnej. W razie potrzeby można włączyć funkcję *auto injection*, która powoduje automatyczne odpalanie grupy wstrzykiowanej (bit ADCx_CR1_JAUTO). W trybie automatycznej iniekcji konwersje grupy wstrzykiwanej uruchamiane są po zakończeniu konwersji grupy regularnej bez dodatkowych trygierzy.

Uwaga! Nie jest możliwe uzyskanie ciągłego (*continuous*) przetwarzania grupy wstrzykiwanej bez grupy regularnej. W trybie ciągłym **zawsze** uruchamiana jest przynajmniej jedna konwersja grupy regularnej. Nie ma fizycznej możliwości skonfigurowania grupy regularnej tak, aby zaprogramowane było zero konwersji.

Z egzotycznych ciekawostek zostały jeszcze tryby *discontinuous*. Działanie trybu nieciągłego zależy od grupy, i tak:

- w grupie regularnej tryb *discontinuous* (bit ADCx_CR1_DISCEN) pozwala na podzielenie konwersji na mniejsze porcje (o długości ustawianej w polu ADCx_CR1_DISCNUM) odpalone kolejnymi trygierami.

Przykład: założmy że skonfigurowaliśmy grupę regularną tak aby dokonywała ośmiu konwersji kolejno IN0, IN1, IN2, IN3, IN4, IN5, IN6, IN7 i ustawiliśmy wartość DISCNUM na 3, wtedy w trybie nieciągłym:

- pierwszy trygier uruchomi sekwencję trzech konwersji: IN0, IN1, IN2
- drugi trygierz uruchomi sekwencję trzech konwersji: IN3, IN4, IN5
- trzeci trygierz uruchomi sekwencję pozostałych dwóch konwersji: IN6, IN7
- czwarty trygierz zadziała tak jak pierwszy i zapętli...

- w grupie iniekcyjnej, tryb *discontinuous* (bit ADCx_CR1_JDISCEN) powoduje, że każda kolejna konwersja z grupy, wymaga osobnego trygierza. Czyli: założmy że skonfigurowaliśmy grupę tryskawkową tak, aby dokonywała trzech konwersji IN0, IN1, IN2. Wówczas:
 - pierwszy trygierz uruchomi konwersję kanału IN0
 - drugi trygierz uruchomi konwersję kanału IN1
 - trzeci trygierz uruchomi konwersję kanału IN2
 - czwarty trygierz uruchomi konwersję kanału IN3
 - zapętlilj...

Tryb nieciągły należy stosować tylko dla jednej grupy jednocześnie!

I tym sposobem dotarliśmy do końca rozdziału. Jest mietlik w głowie? Powinien. Spróbujmy jakoś podsumować te tryby (nazwy trybów takie jak w RM):

Tabela 13.2 Tryby pracy pojedynczego przetwornika ADC

tryb	opis
<i>Single Channel, Single Conversation Mode</i>	przetwornik wykonuje pojedynczą konwersję jednego kanału grupy regularnej lub wstrzykiwanej
<i>Single Channel, Continuous Conversation Mode</i>	przetwornik w kółko mieli jeden kanał z grupy regularnej
<i>Multichannel (Scan), Single Conversation Mode</i>	przelatuje wszystkie kanały grupy wstrzykiwanej lub regularnej
<i>Multichannel (Scan), Continuous Conversation Mode</i>	jw. i się zapętla
<i>Injected conversion mode</i>	konwersje wstrzykiwane przerywają konwersje regularne
<i>Auto-injection Mode</i>	konwersje wstrzykiwane są automatyczne wykonywane po zakończeniu konwersji regularnych
<i>Discontinuous Regular</i>	konwersje grupy regularnej zostają podzielone na krótsze sekwencje
<i>Discontinuous Injected</i>	każda kolejna konwersja z grupy wstrzykiowanej wymaga oddzielnego trygierza

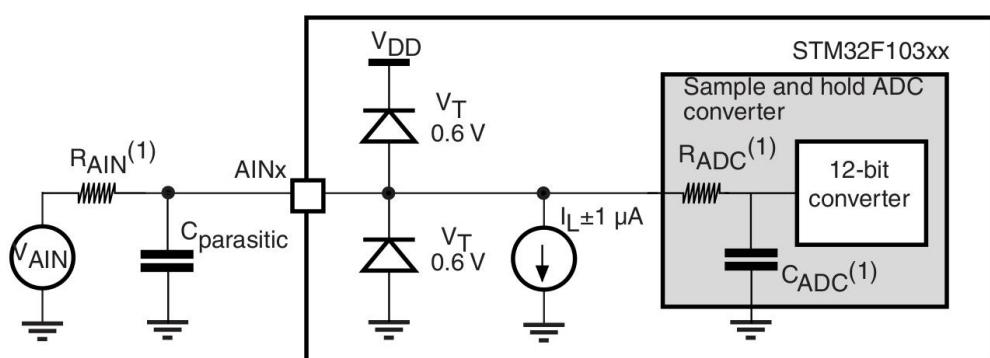
Co warto zapamiętać z tego rozdziału?

- warto mieć ogólne pojęcie o tym jakie tryby są dostępne (ale na Boga - nie uczyć się tego na pamięć!)
- pamiętać cyrki z bitem ADON (wybudzanie przetwornika, uruchamianie konwersji regularnych)

- po wybudzeniu przetwornika należy chwilę odczekać zanim będzie w 100% gotowy do pracy
- DMA Twoim przyjacielem przy ADC

13.3. Czas próbkowania (F103)

Temat trochę poboczny, ale całkiem olać nie można... będzie w skrócie i na chybcika... Być może zwróciłeś uwagę na rejestyry *Sample Time Register* (ADCx_SMPRy). Umożliwiają one ustawienie czasu próbkowania (niezależnie dla każdego kanału). O co chodzi? Uwaga dla humanistów: będzie trochę matematyki :) Prosty schemat zastępczy obwodu pomiarowego z ADC mógłby wyglądać w ogólnych zarysach jakoś tak :



Rys. 13.2. Schemat zastępczy obwodu pomiarowego (źródło: *datasheet STM32F10x*)

gdzie:

- V_{AIN} - źródło mierzonego sygnału
- $R_{AIN}^{(1)}$ - rezystancja źródła, ścieżek etc etc
- $C_{parasitic}$ - pojemność ścieżek, nóżki mikrokontrolera etc etc (kilka/naście pF)
- AIN_x - nóżka mikrokontrolera
- $R_{ADC}^{(1)}$ - rezystancja selektora kanałów ($1k\Omega$)
- $C_{ADC}^{(1)}$ - pojemność kondensatora układu próbkującego ($8pF$)

Mamy źródło sygnału mierzonego (V_{AIN}), szeregową rezystancję i pojemność. Po każdej zmianie kanału, multiplekser (takie małe przełączniczki w mikrokontrolerze) łączy jedno z wejść analogowych z układem próbkującym. W tym momencie pojemność C_{ADC} jest ładowana przez wszystkie rezystancje po drodze. Zakładam, że wiesz, jak wygląda przebieg napięcia na ładowanym kondensatorze w funkcji czasu?

I teraz myk jest taki, że z jednej strony chcemy aby próbkowanie (czyli ten czas kiedy jest wybrany jakiś kanał i ładuje się kondensator) był jak najkrótszy - bo dzięki temu skróceniu ulega czas pomiaru. Czyli możemy mierzyć częściej, a więc i możemy mierzyć sygnały o większej częstotliwości. Z drugiej strony kondensator musi zdążyć naładować się do napięcia źródła, żeby nie było przeklamań. Trzeba znaleźć złoty środek. I teraz tak:

- sumaryczny czas konwersji to czas próbkowania (czyli tego ładowania kondensatora) plus 12,5 tików zegara²⁰⁵ ADC:

$$T_{conv} = T_s + 12.5$$

- dzięki Panom: Harremu N. i Claude'owi S.²⁰⁶ wiemy, że częstotliwość próbkowania powinna być min. 2x większa niż częstotliwość sygnału który chcemy móc odwzorować później z próbek (mówimy tu cały czas o sygnałach zmiennych):

$$\frac{1}{T_{conv}} > 2 \cdot f_{in}$$

- z datasheetu mamy ładny wzór na maksymalną wartość rezystancji wejściowej (R_{AIN} , patrz rys. 13.2), która przy danym czasie próbkowania, zapewni nam wiarygodne pomiary:

$$R_{AIN} < \frac{T_s}{f_{ADC} \cdot C_{ADC} \cdot \ln(2^{N+2})} - R_{ADC}$$

gdzie:

- T_s - czas próbkowania w cyklach zegara ADC
- N - rozdzielcość przetwornika (12 bit)
- f_{ADC} - częstotliwość taktowania bloku ADC
- R_{ADC} - rezystancja selektora kanałów (1kΩ)

Co z tego wynika? A założmy sobie, że ADC jest taktowane z maksymalną dopuszczalną częstotliwością (patrz rozdział o systemie zegarowym 17) - 14MHz - i policzmy graniczną (maksymalną) rezystancję źródła sygnału (R_{AIN}) dla maksymalnego i minimalnego czasu próbkowania (patrz rejesty ADCx_SMP Ry):

²⁰⁵ to 12,5 to jest jakiś stały czas, w którym ADC mieli wynik zanim go wypluje; coś mi chodzi po głowie, że przy pierwszym pomiarze po wyłączeniu ADC ten czas może być dłuższy, ale nie chce mi się szukać szczegółów

²⁰⁶ Harry Nyquist i Claude Shannon

- minimalny czas próbkowania (1,5 tiku zegara ADC):

$$R_{AIN} < \frac{1.5}{14e6 \cdot 8e-12 \cdot \ln(2^{12+2})} - 1e3 \approx 380 \Omega$$

- sumaryczny czas konwersji (przy $f_{ADC} = 14\text{MHz}$):

$$T_{conv} = 1.5 + 12.5 = 14 \text{ cykli zegara ADC} \rightarrow 14/14\text{MHz} = 1\mu s$$

- maksymalny czas próbkowania (239,5 tiku zegara ADC):

$$R_{AIN} < \frac{239.5}{14e6 \cdot 8e-12 \cdot \ln(2^{12+2})} - 1e3 \approx 219 k\Omega$$

- sumaryczny czas konwersji (przy $f_{ADC} = 14\text{MHz}$):

$$T_{conv} = 239.5 + 12.5 = 252 \text{ cykle zegara ADC} \rightarrow 252/14\text{MHz} = 18\mu s$$

Przy najkrótszym czasie konwersji ($1\mu s$) możemy uzyskać do miliona pomiarów na sekundę (1MHz , czy jak kto woli 1Msps). Czyli możemy odwzorować sygnał o częstotliwości do 500kHz . Ale! Rezystancja źródła sygnału musi być mniejsza niż 380Ω . A to jest bardzo mało...

Jeśli rezystancja będzie większa, to musimy wydłużyć czas próbkowania co pociągnie za sobą spadek maksymalnej częstotliwości. Przy maksymalnym czasie próbkowania, rezystancja może wynosić teoretycznie ponad $219\text{k}\Omega^{207}$. Niestety maksymalna częstotliwość odwzorowywanego sygnału wynosi wtedy tylko około 28kHz ! Uff starczy - jedziemy dalej.

Co warto zapamiętać z tego rozdziału?

- im większa jest impedancja źródła sygnału, tym dłuższy musi być czas próbkowania
- im dłuższy czas próbkowania tym dłuższy pomiar
- im dłuższy pomiar tym mniejsza częstotliwość pomiarów
- im mniejsza częstotliwość pomiarów, tym mniejsza częstotliwość sygnału jaki możemy odwzorować z uzyskanych pomiarów
- jeśli mierzymy sygnały stałe/wolnozmienne²⁰⁸ i nie zależy nam na częstych pomiarach to mamy to wszystko głęboko w... ustawiamy np. maksymalny czas próbkowania i olewamy sprawę

207 datasheet podaje, że maksymalna rezystancja źródła nie powinna przekraczać $50\text{k}\Omega$

208 niektórzy uważają, że poniżej 10GHz to praktycznie stały sygnał :)

13.4. Tryby pracy dwóch przetworników ADC (F103)

Primo: jakieś pomysły jak przetłumaczyć *dual mode* aby miało to ręce i nogi? Tryby *dual mode* to tryby, w których dwa przetworniki (zawsze ADC1 i ADC2) są ze sobą połączone i *master* (zawsze ADC1) steruje pracą (wyzwala) *slave'a* (zawsze ADC2). Ogólnie tryby *dual* mają za zadanie albo zwiększyć częstotliwość pomiarów danego kanału, albo zsynchronizować pomiary na różnych kanałach.

Kilka uwag na początek:

- w trybie *dual* wybór sygnału trygierującego dokonywany jest tylko w przetworniku *master*
- w konfiguracji przetwornika *slave* należy ustawić wyzwalanie zewnętrzne sygnałem SWSTART
- przetwornik ADC3 nie ma możliwości pracy w trybie *dual mode*²⁰⁹
- w trybie *dual mode* wynik konwersji *regularnych* przetwornika ADC2 (pod warunkiem włączenia bitu DMA) jest z automatu zapisywany w górnej połowie rejestru ADC1_DR. Czyli rejestr *mastera* (ADC1_DR) zawiera wyniki konwersji obu przetworników (rejestr ma 32bity, wyniki po 12 bitów) - patrz rysunek:

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

Rys. 13.3. Rozmieszczenie wyników konwersji dwóch przetworników w jednym rejestrze. Wynik konwersji ADC1 zajmuje bity 0-11 (kolor nadziei), wynik konwersji ADC2 zajmuje bity 16-27 (kolor srebrny).

Trybów pracy jest coś koło dziewięciu... ale nie ma co się przerażać - kilka jest egzotycznych i poza ogólnym pojęciem, że coś takiego istnieje, można o nich zapomnieć:

1. *Injected Simultaneous*
2. *Regular Simultaneous*
3. *Fast Interleaved*
4. *Slow Interleaved*
5. *Alternate Trigger*
6. *Independent*
7. *Injected Simultaneous + Regular Simultaneous*
8. *Regular Simultaneous + Alternate Trigger*
9. *Injected Simultaneous + Interleaved*

209 i dobrze! bo jeszcze by zrobili *Triple Mode* i kolejny pierdylion trybów pracy

Od razu z listy możemy skreślić tryb *independent mode*... Really? AYFKM? Tak! ST postanowiło sytuację, w której dwa przetworniki działają zupełnie niezależnie, mianować pełnoprawnym trybem *dual independent mode*...

Dwa tryby *simultaneous mode* (*injected* i *regular*) to sytuacja w której oba przetworniki pracują po prostu równocześnie. Czyli pojawia się trygierz *mastera* i oba przetworniki na niego reagują - cała filozofia. W trybie *injected* oba konwertują swoje grupy *wstrzykiwane*. W trybie *regular*... zgadnij sam. To są przydatne tryby kiedy musimy jednocześnie mierzyć dwa kanały (np. prąd i napięcie aby policzyć moc, przesunięcie fazowe i tak dalej...). Dwa zastrzeżenia:

- oba przetworniki nie mogą jednocześnie mierzyć tego samego kanału
- kanały mierzone równocześnie muszą mieć ustalony ten sam czas próbkowania

Zadanie domowe 13.5: uruchomić pomiary w trybie *dual simultaneous continuous external trigger regular mode*²¹⁰, czyli po ludzku: dwa przetworniki mają jednocześnie mierzyć swoje grupy *regularne* (po jednym kanale w grupie wystarczy). Pomiar ma być uruchamiany co 1ms zewnętrznym trygierzem. Wyniki wysyłane przez DMA do bufora w SRAMie. W sumie niech w buforze lądują wyniki z 10 tys. pomiarów. Niechaj kod wsadem się stanie!

²¹⁰ coś chyba pokręciłem z tą nazwą... ale nawet ST się w tym gubi i w każdym dokumencie nazywa tryby inaczej :)

Przykładowe rozwiążanie (F103, pomiar na PC0 i PC1):

```
1. int main(void) {
2.
3.     static volatile uint16_t wyniki[20000];
4.
5.     RCC->APB2ENR = RCC_APB2ENR_IOPCEN | RCC_APB2ENR_ADC1EN | RCC_APB2ENR_ADC2EN;
6.     RCC->AHBENR |= RCC_AHBENR_DMA1EN;
7.     RCC->APB1ENR = RCC_APB1ENR_TIM3EN;
8.
9.     gpio_pin_cfg(GPIOC, PC0, gpio_mode_input_analog);
10.    gpio_pin_cfg(GPIOC, PC1, gpio_mode_input_analog);
11.
12.    DMA1_Channel1->CPAR = (uint32_t)&ADC1->DR;
13.    DMA1_Channel1->CMAR = (uint32_t)wyniki;
14.    DMA1_Channel1->CNDTR = 10000;
15.    DMA1_Channel1->CCR = DMA_CCR1_EN | DMA_CCR1_PSIZE_1 | DMA_CCR1_MSIZE_1 | DMA_CCR1_MINC;
16.
17.    TIM3->PSC = 4000-1;
18.    TIM3->ARR = 1;
19.    TIM3->EGR = TIM_EGR_UG;
20.    TIM3->CR2 = TIM_CR2_MMS_1;
21.
22.    ADC1->CR2 = ADC_CR2_ADON;
23.    ADC2->CR2 = ADC_CR2_ADON;
24.    for(volatile uint32_t delay = 100000; delay; delay--);
25.
26.    ADC1->CR1 = ADC_CR1_DUALMOD_1 | ADC_CR1_DUALMOD_2;
27.    ADC1->CR2 |= ADC_CR2_EXTTRIG | ADC_CR2_EXTSEL_2 | ADC_CR2_DMA;
28.    ADC2->CR2 |= ADC_CR2_EXTTRIG | ADC_CR2_EXTSEL;
29.    ADC1->SQR3 = 10;
30.    ADC2->SQR3 = 11;
31.
32.    BB(TIM3->CR1, TIM_CR1_CEN) = 1;
33.
34.    while ((DMA1->ISR & DMA_ISR_TCIF1) == 0);
35.    __BKPT();
36.
37. } /* main */
```

3) bufor na próbki; uważaj na zajętość pamięci! 10 000 pomiarów x2 kanały to będzie 20 000 wyników, każdy wynik zapisany w zmiennej 2B czyli w sumie będzie 40 000B danych (~39kB a procekk ma „tylko” 48kB)

... nuda ...

12 - 15) nasz przyjaciel DMA :) Jest tu odrobinka magii jeśli chodzi o rozmiary danych. Zwróć uwagę, że zarówno po stronie pamięci jak i peryferiala ustawilem rozmiar 32b. A wyniki będziemy zapisywać w tablicy uintów_16. Ciemno wszędzie, co to będzie!? Powoli:

- przetwornik ADC2 nie potrafi generować żądań DMA, więc:
- w trybie *dual mode* wynik konwersji przetwornika ADC2 jest zapisywany w górnej połówce rejestru ADC1_DR, czyli:
- w rejestrze ADC1_DR siedzą oba wyniki (patrz rysunek 13.3)
- DMA przesyła tą wartość (32bitową) do tablicy zmiennych 16bitowych
- w efekcie jeden przesył DMA zapisze od razu dwie pozycje w tablicy! trzeba się pilnować z rozmiarami danych, tablic i ilością przesyłów - inaczej zamażemy sobie coś w pamięci i zapłaczemy się próbując dociec co się właściwie dzieje :)

- nie wdając się w szczegóły: w tablicy na pozycja parzystych ($0^{211}, 2, 4, 6, 8, \dots$) będą kolejne wyniki konwersji przetwornika ADC1
- na pozycjach nieparzystych ($1, 3, 5, 7, \dots$) będą kolejne wyniki konwersji przetwornika ADC2

17 - 20) konfigurację licznika mamy w małym palcu. Zwracam uwagę na to, że w tym przykładzie korzystam z innego licznika niż w poprzednich. Jeśli nie wiesz czemu, to wróć do opisu rozwiązania zadania 13.2.

22 - 24) wybudzenie obu przetworników z trybu uśpienia

26) konfigurujemy wybrany tryb dual (uwaga! tryb dual konfiguruje się tylko w opcjach ADC1)

27) konfiguracja wyzwalania ADC1 i włączenie generowania żądań DMA (tylko w opcjach ADC1)

28) ADC2 konfigurujemy tak, aby był wyzwalany zewnętrznie, bitem SWSTART

29, 30) konfiguracja grup obu przetworników

32)łączmy generator trygierzy :)

34) czekamy na koniec DMA (czyli na przesłanie 10 000 wyników) i podglądamy w debuggerze. Wklejenie wszystkich wyników sobie daruję :) Uwierz - działa!

Kolejne dwa tryby są podobne do siebie: *fast interleaved mode* i *slow interleaved mode*. Te tryby stworzone są w celu zwiększenia częstotliwości pomiarów jednego kanału grupy regularnej. Tryb „szybki przeplatany” lub ”przeplatany szybko” (*fast interleaved*) działa tak, że najpierw ADC2 (wyzwalane trygierzem mastera) próbuje wybrany kanał. Następnie po 7 tikach zegara ADC, kiedy ADC2 zakończyło już próbkowanie, odpala się ADC1 i próbuje ten sam kanał. Dzięki temu możemy podwoić częstotliwość pomiarów uzyskując maksymalnie 2Ms/s (miliony próbek na sekundę, 2MHz, 2Msps... jak kto lubi) przy najkrótszym czasie próbkowania. Uwaga! Długość próbkowania kanału w tym trybie, musi być mniejsza niż 7 tików zegara ADC.

W trybie wolnym przeplatanym (*slow interleaved*) jest identycznie, tylko opóźnienie uruchomienia ADC1 wynosi 14 cykli, czas próbkowania można wydłużyć do 14 cykli a pomiary się z automatu zapętlają (*continuous*).

Do czego to wykorzystać? Tryb fast umożliwia pomiary szybkich sygnałów. Tryb slow...:

Przykładzik (źródło: AN3116, tylko tam coś źle policzyli bo się nie zgadza z datasheetem...). Założmy, że mamy sygnał o częstotliwości 500kHz a rezystancja jego źródła wynosi $10k\Omega$ (jak ktoś nie pamięta to odsyłam do rozdziału 13.3). Aby móc taki sygnał odwzorować z pomiarów, musimy go próbkować z częstotliwością minimum 1MHz. Da się to zrobić jednym przetwornikiem?

211 zero jest liczbą parzystą?

Policzmy co nie co:

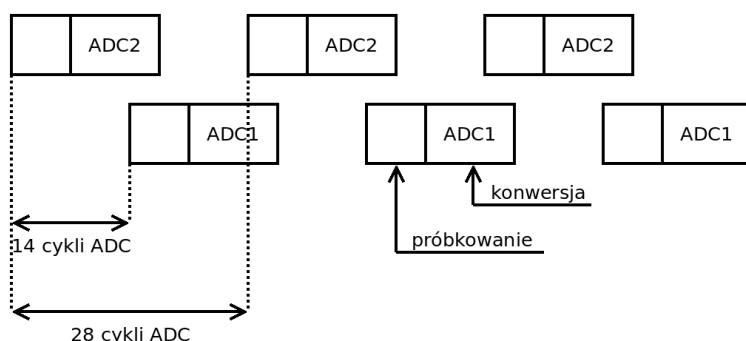
- minimalny czas konwersji (dla zegara ADC równego 14MHz) wynosi $1\mu s$, stąd maksymalna częstotliwość próbkowania jest równa 1MHz - czyli ok!
- maksymalna rezystancja źródła (patrz rozdział 13.3): 380Ω - czyli nie jest ok!

Nasze źródło ma zdecydowanie zbyt dużą rezystancję ($10k\Omega >> 380\Omega$)! Trzeba wydłużyć czas próbkowania kosztem częstotliwości pomiarów:

- biorąc pod uwagę rezystancję źródła ($10k\Omega$), obliczmy wymagany minimalny czas próbkowania (przekształcenia wzorów sobie daruję, to nie kółko matematyczne): wynik to 12 cykli zegara ADC (przy częstotliwości zegara ADC równej 14MHz)
- najbliższa wartość (zaokrąglona w górę) jaką można ustawić w rejestrach ADCx_SMPR to 13,5 cyklu
- częstotliwość próbkowania dla jednego przetwornika z czasem próbkowania ustawionym na 13,5 tyknięć zegara ADC, wyniesie:

$$f_{ADC} / (13.5 + 12.5) \approx 540 \text{ kHz}$$

Zdecydowanie za mało jak na nasz sygnał (potrzebujemy minimum 1MHz)... i tu z pomocą przychodzi *slow interleaved*! W tym trybie próbkowanie może wynosić do 14 cykli (łapiemy się z naszym 13,5). Tuż po zakończeniu próbkowania przez jeden przetwornik (czyli w czasie tych 12,5 tików kiedy ADC mieli wynik), uruchamiany jest drugi przetwornik. Obrazek pogladowy (kolejny przetwornik uruchamiany jest z opóźnieniem 14 cykli):



Rys. 13.4. Tryb *slow interleaved*

Na obrazku ładnie widać, że kolejny pomiar jest rozpoczęty co 14 cykli ADC. Czyli co 14 cykli będziemy mieli nowy wynik pomiaru. Czyli, przy taktowaniu ADC z częstotliwością

14MHz, będziemy mieli pomiary z częstotliwością 1MHz - tak jak chcieliśmy :) Kapitalnie! O to nam chodziło.

Przykładowego kodu nie będzie, bo ciężko pokazać zalety trybu, a konfiguracja sprowadza się tylko do zabawy polem ADCx_CR1_DUALMOD.

Co tam dalej... właściwie to już nuda do końca:

- *Alternate Trigger*: na trygierz wykonuje się konwersja grupy strzykanej ADC1, na kolejny trygierz odpala się grupa strzykana ADC2
- *Combined Regular + Injected Simultaneous Mode*: pamiętasz przykład z przerywaniem konwersji grupy regularnej na czas pomiaru baterii w grupie wstrzykiwanej (zadanie 13.4)? Tu jest dokładnie to samo tylko oba przetworniki naraz przerywają swoje grupy regularne i odpalają wstrzykiwane
- *Combined Regular Simultaneous + Alternate Trigger Mode*: przerywamy jednocześnie przetwarzanie grup regularnych przetworników, wstrzykując pojedynczy kanał z grupy strzykanej
- *Combined Injected Simultaneous + Interleaved*: przerwanie trybu *interleaved* przez grupę strzykaną

Jakaś masakra jest z tymi trybami... tzn. fajnie, że jest ich dużo i na każdą okazję, ale można pierdolca dostać jeśli ktoś będzie chciał to wszystko ogarnąć. Podsumowanie w formie matrixa:

Tabela 13.3. Tryby *dual* przetworników ADC

tryb	opis
<i>Regular Simultaneous</i>	równoległe przetwarzanie grup <i>zwyczajnych</i> przez przetworniki
<i>Injected Simultaneous</i>	równoległe przetwarzanie grup <i>wstrzykiwanych</i> przez przetworniki
<i>Fast Interleaved</i>	dwa przetworniki próbują jeden kanał, opóźnienie wyzwalania drugiego ADC - 7 cykli
<i>Slow Interleaved</i>	dwa przetworniki próbują jeden kanał, opóźnienie wyzwalania drugiego ADC - 14 cykli
<i>Alternate Trigger</i>	przetworniki na przemian wykonują konwersje swoich grup <i>wstrzykiwanych</i>
<i>Independent Dual</i>	niepodważalny dowód na to, że redaktorzy RMA mają specyficzne poczucie humoru
<i>Injected Simultaneous + Regular Simultaneous</i>	w obu przetwornikach następuje przerwanie konwersji grup <i>zwyczajnych</i> i przetwarzane są grupy <i>wstrzykiwane</i>
<i>Regular Simultaneous + Alternate Trigger</i>	jw. ale konwertowane są pojedyncze kanały grup <i>wstrzykiwanych</i>
<i>Injected Simultaneous + Interleaved</i>	przerwanie trybu <i>Interleaved</i> przed grupą <i>wstrzykiwaną</i>

Co warto zapamiętać z tego rozdziału?

- tryby *dual* dotyczą tylko ADC1 i ADC2
- tryby *dual* pozwalają zwiększyć częstotliwość próbkowania kanału lub zsynchronizować pomiary kilku kanałów
- warto mieć jakieś pojęcia o dostępnych trybach *dual*
- wynik konwersji ADC2 jest zapisywany w górnej połowie rejestru ADC1_DR

13.5. Bajery (F103)

Do wodotrysków zaliczam:

- „analogowy” *watchdog*
- wbudowany czujnik temperatury
- źródłko napięcia odniesienia
- funkcję samo kalibracji
- *offset* dla kanałów *iniekcyjnych*
- możliwość zmiany wyrównania wyników

„Analogowy” *watchdog* to dosyć intuicyjny układ. Sprawdza na bieżąco czy wynik konwersji mieści się w założonych widełkach. W sumie to nie bardzo rozumiem co jest w nim analogowego, dla mnie to on jest cyfrowy skoro porównują cyfrową wartość wyplutą przez ADC... ale ja się nie znam. Co by tu o nim napisać... to może przykład :)

Zadanie domowe 13.6: niech ADC sobie w kółko międli jeden kanał. Jeśli napięcie jest poniżej ~1,1V to pali się jedna dioda, jeśli jest powyżej ~2,2V to pali się druga dioda. „Pomiędzy” nie pali się nic. Oczywiście chodzi o wykorzystanie AWD.

Przykładowe rozwiązań (F103, wejście analogowe PC0, diody na PB0 i PB1):

```
1. const uint32_t awd_htr = 2731;
2. const uint32_t awd_ltr = 1365;
3.
4. int main(void) {
5.
6.     RCC->APB2ENR = RCC_APB2ENR_IOPCEN | RCC_APB2ENR_ADC1EN | RCC_APB2ENR_IOPBEN;
7.     gpio_pin_cfg(GPIOC, PC0, gpio_mode_input_analog);
8.     gpio_pin_cfg(GPIOB, PB0, gpio_mode_output_PP_2MHz);
9.     gpio_pin_cfg(GPIOB, PB1, gpio_mode_output_PP_2MHz);
10.
11.    ADC1->CR2 = ADC_CR2_ADON | ADC_CR2_CONT;
12.    for(volatile uint32_t delay = 100000; delay; delay--);
13.
14.    ADC1->CR1 = ADC_CR1_AWDEN | ADC_CR1_AWDIE | ADC_CR1_AWDSGL | 10;
15.    ADC1->SQR3 = 10;
16.    ADC1->LTR = awd_ltr;
17.    ADC1->HTR = awd_htr;
18.    ADC1->CR2 |= ADC_CR2_ADON;
19.
20.    NVIC_EnableIRQ(ADC1_2_IRQn);
21.    while(1);
22.
23. } /* main */
24.
25. __attribute__((interrupt)) void ADC1_2_IRQHandler(void){
26.
27.     if ( ADC1->SR & ADC_SR_AWD ){
28.         ADC1->SR = (~ ADC_SR_AWD) & 0x1f;
29.         BB(GPIOB->ODR, PB0) = (ADC1->DR > awd_htr ? 1 : 0);
30.         BB(GPIOB->ODR, PB1) = (ADC1->DR < awd_ltr ? 1 : 0);
31.     }
32. }
```

1, 2) wartość górnego i dolnego progu AWD, dla wygody w zmiennych tylko do odczytu (*const* w języku C nie oznacza stałej!)

... nuda ...

14) włączam AWD dla kanałów grupy regularnej, włączam przerwanie od AWD i ustawiam numer kanału nadzorowanego przez AWD. Tutaj się na chwilę zatrzymajmy.

AWD może nadzorować wybrany kanał jednej z grup lub wszystkie kanały grupy/grup. Jeśli ma nadzorować jeden kanał to należy ustawić bit ADCx_CR1_AWDSGL i numer kanału w polu ADCx_CR1_AWDCH. Wybór grupy objętej kontrolą watchdoga dokonywany jest poprzez bit ADCx_CR1_AWDEN i ADCx_CR1_JAWDEN. Odsyłam do tabelki *Analog watchdog channel selection*. I niby wszystko cacy, ale:

- czy trzeba ustawać bit AWDSGL jeśli konwersji podlega tylko jeden kanał (nie jest włączony *scan mode*)?
- co się stanie jeśli ustawiemy AWDSGL i w polu AWDCH wybierzemy numer kanału, który w ogóle nie podlega konwersji?

Na te pytania nie znalazłem odpowiedzi w RMie, więc trochę poeksperymentowałem. Z obserwacji wynika, że jeśli konwersji podlega tylko jeden kanał to AWDSGL nic nie zmienia. Co do drugiej

kropki - AWD na moje oko przestał działać - i to ma sens, bo cały czas czekał na konwersje jakiegoś kanału, która nie następowała. Ale proszę nie przyjmować tego za pewnik :) Koniec OT!

27) w przerwaniu sprawdzam flagę AWD i ją kasuję. Flaga AWD jest ustawiana (i odpala przerwanie), gdy wynik konwersji ADC przekroczy jeden z progów. Niestety nie ma żadnych sprzętowych flag informujących o tym, jaki jest aktualny stosunek wartości z ADC i progów - stąd porównywanie na piechotę.

Zadanie domowe 13.7: sprawdzić jak zachowa się AWD jeśli górnego prógu (HTR) będzie ustawiony niżej niż dolny (LTR).

Kolejna nowinka to **czujnik temperatury**. Podpięty jest na stałe do kanału 16 przetwornika ADC1 (tylko do ADC1). Zalecany czas konwersji przy pomiarze z tego kanału wynosi minimum 17.1 μ s. Czujnik jest raczej mało przydatny do pomiaru temperatury bezwzględnej, różnice wskazań między kilkoma mikrokontrolerami mogą dochodzić do 45°C. Można go natomiast z powodzeniem użyć do wykrycia zmian temperatury, np. w celu kalibracji ADC po wyniesieniu urządzenia na dwór (nagły spadek temperatury). Wartość temperatury obliczana jest za pomocą formułki:

$$T = \frac{V_{25} - V_{sense}}{\text{Avg_slope}} + 25 \quad [{}^{\circ}\text{C}]$$

gdzie (źródło datasheet):

- V_{25} - napięcie czujnika przy 25°C (1,34...1,52V)
- V_{sense} - zmierzone napięcie czujnika
- Avg_slope - współczynnik nachylenia charakterystyki czujnika (4,0...4,6 mV/°C)

W STMa wbudowane jest ponadto **wewnętrzne źródło napięcia odniesienia** ($V_{refint} = 1,2\text{V}$). Rola źródełka jest jednak całkowicie odmienna niż w AVR. Nie można użyć go jako napięcia odniesienia dla ADC czy DAC. Źródełko jest na stałe podpięte do kanału 17 ADC1. Ktoś zapyta po co? Otóż umożliwia to pomiary przy nieznanym napięciu referencyjnym (np. kiedy układ jest zasilany prosto z baterii której napięcie się zmienia w miarę rozładowywania). Nie możemy wtedy przeliczyć wartości z ADC na napięcie, bo nie znamy V_{ref} . W takiej sytuacji można wykonać pomiar wbudowanego napięcia referencyjnego (o znanej wartości) i na tej podstawie (przez proporcje) policzyć sobie zależność między wynikiem wypluwianym przez ADC a wartością napięcia.

W sumie to nie do końca „bajer”, ale na własny rozdział nie zasłużyło - **samo-kalibracja**. Dokumentacja jest dosyć oszczędna w kwestii wy tłumaczenia, na czym ta kalibracja właściwie polega. Zalecam uruchomić przynajmniej raz połączeniu zasilania. Od siebie dodam, że warto powtarzać kalibrację jeśli nastąpi znacząca zmiana temperatury otoczenia układu²¹². Pod względem programowym sprawa jest prosta jak zawsze:

Zadanie domowe 13.8: uruchomić przetwornik ADC tak aby mierzył jakiś kanał. Wykonać 10 pomiarów, wyniki zapisywać w tablicy. Następnie przeprowadzić procedurę kalibracji ADC i powtórzyć pomiary. Wyniki porównać i potwierdzić, że działa a Poradnik jest wspaniały :)

Przykładowe rozwiązanie (F103, pomiary na PC0):

```

1. int main(void) {
2.
3.     static volatile uint16_t wyniki_bezCalib[10];
4.     static volatile uint16_t wyniki_poCalib[10];
5.
6.     RCC->APB2ENR = RCC_APB2ENR_IOPCEN | RCC_APB2ENR_ADC1EN;
7.     RCC->AHBENR |= RCC_AHBENR_DMA1EN;
8.     gpio_pin_cfg(GPIOC, PC0, gpio_mode_input_analog);
9.
10.    DMA1_Channel1->CPAR = (uint32_t)&ADC1->DR;
11.    DMA1_Channel1->CMAR = (uint32_t)wyniki_bezCalib;
12.    DMA1_Channel1->CNDTR = 10;
13.    DMA1_Channel1->CCR = DMA_CCR1_MSIZE_0 | DMA_CCR1_PSIZE_1 | DMA_CCR1_MINC | DMA_CCR1_EN;
14.
15.    ADC1->CR2 = ADC_CR2_ADON | ADC_CR2_CONT | ADC_CR2_DMA;
16.    for(volatile uint32_t delay = 100000; delay; delay--);
17.    ADC1->SQR3 = 10;
18.    ADC1->CR2 |= ADC_CR2_ADON;
19.
20.    while( !(DMA1->ISR & DMA_ISR_TCIF1) );
21.    BB(ADC1->CR2, ADC_CR2_ADON) = 0;
22.
23.    BB(DMA1_Channel1->CCR, DMA_CCR1_EN) = 0;
24.    DMA1_Channel1->CMAR = (uint32_t)wyniki_poCalib;
25.    DMA1_Channel1->CNDTR = 10;
26.
27.    BB(ADC1->CR2, ADC_CR2_ADON) = 1;
28.    for(volatile uint32_t delay = 100000; delay; delay--);
29.    BB(ADC1->CR2, ADC_CR2_RSTCAL) = 1;
30.    while (BB(ADC1->CR2, ADC_CR2_RSTCAL));
31.    BB(ADC1->CR2, ADC_CR2_CAL) = 1;
32.    while (BB(ADC1->CR2, ADC_CR2_CAL));
33.
34.    BB(DMA1_Channel1->CCR, DMA_CCR1_EN) = 1;
35.    BB(ADC1->CR2, ADC_CR2_ADON) = 1;
36.
37.    while( !(DMA1->ISR & DMA_ISR_TCIF1) );
38.    __BKPT();
39.
40. } /* main */

```

Długańskie wyszło... Od początku: dwie tablice na wyniki pomiarów, zegary, konfiguracja wejścia IN10, DMA (przesłanie 10-ciu wyników do pierwszej tablicy), konfiguracja i włączenie ADC. Wszystko już było :)

20) czekamy za zakończenie przesyłu przez DMA, czyli na przesłanie 10-ciu wyników z ADeCe i...

212 nie wiem czy gdzieś to wyczytałem czy mi się samo uroipo :)

- 21)** wyłączamy ADeCe, żeby przestał mierzyć (tzn. wprowadzamy go w tryb *power down*)²¹³
- 23)** wyłączamy kanał DMA żeby móc zmienić jego konfigurację
- 24, 25)** zmieniamy miejsce docelowe (druga tablica) i ustawiamy znowu 10 przesyłów
- 27)** w linijce 21 uśpiliśmy ADC, teraz go budzimy
- 29, 30)** bit RSTCAL powoduje zresetowanie danych kalibracyjnych (nie wiem, nie pytać!), jest sprzętowo kasowany po zakończeniu tej operacji - czekamy
- 31, 32)** bit CAL włącza auto kalibrację, po zakończeniu procesu jest sprzętowo kasowany, czekamy
- 34, 35)** nazad włączamy DMA i uruchamiamy konwersje ADC
- 37, 38)** gdy DMA prześle 10 wyników, program jest przerywany

Wyniki odczytane debuggerem (na IN10 mniej więcej $\frac{1}{2}$ napięcia odniesienia):

- przed kalibracją: 1962, 1964, 1964, 1965, 1965, 1964, 1963, 1965, 1966, 1964
(średnia²¹⁴: 1964,2)
- po kalibracji: 2034, 2036, 2035, 2036, 2037, 2036, 2035, 2035, 2036, 2042
(średnia: 2036,2)

wierzę że jest to zmiana na lepsze i wynika z kalibracji :)

No to jeszcze został *offset* i wyrównanie danych. *Offset* nawet może się do czegoś przydać. Polega to na tym, że po zakończeniu konwersji w trybie wstrzykiwanym, od wyniku odejmowana jest wartość offsetu, a dopiero potem wynik jest zapisywany w rejestrze ADCx_JDRy. Offset ustawiany jest w czterech rejestrach ADCx_JOFRy, osobno dla każdego rejestru danych grupy wstrzykiwanej. Uwaga! W wyniku odejmowania może powstać liczba ujemna ze znakiem!

Za pomocą bitu ADCx_CR2_ALIGN można sobie wybrać wyrównanie danych w rejestrach ADCx_DR i ADCx_JDRy - do lewej, albo do prawej - odsyłam do diagramów w RMie... ot i cała filozofia.

Co warto zapamiętać z tego rozdziału?

- ADC ma funkcję automatycznej kalibracji, którą należy odpalić przynajmniej raz po każdym włączeniu zasilania²¹⁵
- do kontroli wyników konwersji ADC mamy układ (pseudo) *analogowego watchdoga*
- jest coś takiego jak *offset*, czujnik temperatury, źródło napięcia odniesienia

213 właściwie to nie wiem czy trzeba koniecznie to robić... ale tak mi się wydaje logicznie

214 średnia jest jak bikini: bardzo dużo pokazuje, tylko nie to co istotne :)

215 wiem, w przykładach tego nie było - *mea culpa, mea maxima culpa!* - nie chciałem komplikować przykładów :)

13.6. Ogólne spojrzenie na tryby pracy przetwornika ADC (F103 i F429)

Czytając opis ADC w *Reference Manualu* do STM32F429, znalazłem potwierdzenie kilku domysłów jakie opisałem przy omawianiu przetwornika w F103. To mnie cieszy! Z drugiej strony wydaje mi się, że twórcy RMa obrali kiepską drogę przy opisie ADC. Chodzi mi o to, że naprodukowali mnóstwo trybów, każdemu nadali fikuśną nazwę, opisali dokładnie jak działa i jak krok po kroku należy skonfigurować poszczególne bity aby ten tryb uruchomić. Według mnie takie podejście niepotrzebnie zaciemnia temat i lepiej byłoby po prostu opisać elementarne działanie poszczególnych bitów konfiguracyjnych. Bez bawienia się w odrębne *configuration mode'y* w których można się pogubić i które nie wyczerpują wszystkich możliwości konfiguracji. I to spróbuję zrobić :)

Po pierwsze: grupa *wstrzykiwana* jest predysponowana dla konwersji wyzwalanych jakimś sygnałem; takich które muszą być wykonane szybko i jednokrotnie. Grupa *wstrzykiwana* praktycznie zawsze może przerwać konwersje z grupy *zwyczajnej*. Bez względu na to jaki mamy tryb, *Single, Dual, Triple* itd...

Po włączeniu przetwornika (bez jakiś specjalnych ustawień) wykona się konwersja jednego kanału z grupy - bez względu na podaną liczbę konwersji w ustawieniach grupy. Której grupy? To zależy od sygnału, który wyzwoilił konwersje - każda grupa ma swoje trygierze. Który kanał? Ten który będzie ustawiony w konfiguracji grupy.

Jeżeli chcemy przeprowadzić konwersję większej liczby kanałów to trzeba włączyć bit SCAN. Przetwornik wykona wtedy tyle konwersji, ile ustawiono w konfiguracji danej grupy. Poza tym nic to nie zmienia! I bez sensu tworzyć otoczkę z nowym *trybem*. Po prostu będzie tyle konwersji ile ustawiono. Koniec tematu.

Niezależnie od powyższego, jeśli chcemy aby przetwornik nie wyłączał się po zakończeniu konwersji, należy włączyć bit CONT. To zapętli konwersje grupy regularnej. Grupa wstrzykiwana nie jest zapętlana - patrz akapit o grupach. W każdej chwili można jednak grupą wstrzykiwaną przerwać konwersje regularne (znowu patrz akapit o grupach).

Jedyny wyjątek, kiedy grupa wstrzykiwana pracuje na okrągło, jest związany z bitem JAUTO. Powoduje on automatyczne odpalanie grupy wstrzykiwanej po zakończeniu konwersji regularnych. Ma to tylko jedno logiczne zastosowanie: jeśli potrzebujemy ustawić tak długą sekwencję konwersji, że 16 pozycji w grupie regularnej to dla nas zbyt mało.

Do tego dochodzi jeszcze bit DISC, który powoduje podzielenie konwersji z grupy regularnej na mniejsze porcje odpalone kolejnymi wyzwalaczami (trygierzami) zaś w przypadku grupy strzykawkowej sprawia, że każda konwersja wymaga osobnego wyzwolenia. I tyle. Można

go włączyć praktycznie w każdej opisanej wcześniej i później opcji. Nie ma różnicy czy jeden przetwornik, czy *dual mode* - DISC zawsze działa tak samo.

Tryby podwójne... oba tryby *simultaneous* to po prostu wyzwalanie dwóch przetworników jednym sygnałem (trygierzem). Żadnej magii. Żadnego osobnego, hermetycznego trybu. Wszystkie opisane przed chwilą opcje (SCAN, DISC, CONT...) dalej działają tak samo²¹⁶! Zaraz pojawi się *triple mode*²¹⁷. I co? I działa tak samo - tylko teraz trzy przetworniki będą jednocześnie mierzyły swoje grupy kanałów.

Interleaved... slow, fast - co za różnica - cały czas chodzi o to samo - żeby mierzyć ten sam kanał i wyżyłować częstotliwość próbkowania. W F429 ST zrezygnowało z podziału na *slow* i *fast*. Zamiast tego wprowadzili jeden *interleaved* z konfigurowalnym opóźnieniem... i fajnie.

Mówiłem o tym, że grupa wstrzykiwana praktycznie zawsze może przerwać konwersje regularne? Mówiłem :) To po co osobne tryby: *injected simultaneous* + *interleaved* i *injected simultaneous* + *regular simultaneous*. Przecież to właśnie o przerwanie konwersji regularnej przez wstrzykiwaną się rozchodzi.

Jeszcze *alternate trigger* został... jedyny odstający od ogółu :) Na kolejne trygierze, kolejne przetworniki odpalają swoje grupy iniekcyjne. Cała reszta bez zmian. Czyli przetworniki mogły równie dobrze mielić swoje grupy zwyczajne co zostało przerwane przez wyzwolenie grup wstrzykiwanych (i po co nazywać to osobnym trybem *combined regular simultaneous* + *alternate trigger mode*?).

Te ustawienia są w większości niezależne i po mojemu upychanie tego w odrębne *tryby* jest bez sensu i tylko zaciemnia obraz. Człowiek koncentruje się na *trybie* i traci obraz całości. To tak jak czasem, przy czytaniu czegoś nudnego, niby się rozumie słowa i zdania, ale za cholę nie wie się o co właściwie chodzi. Ot i cała magia.

Co warto zapamiętać z tego rozdziału?

- czasem drzewa przesłaniają las...

13.7. Różnice w STM32F429 (F429)

Tak jak wspomniałem, lektura opisu ADC dla tytułowego mikrokontrolera potwierdziła kilka wcześniejszych przypuszczeń. Nie wiem czy to kwestia n-tego czytania tych opisów, czy jakości samych opisów, ale coraz więcej z tego ogarniam tak „globalnie” :)

216 oczywiście są jakieś wyjątki kiedy dana konfiguracja nie ma sensu: np. wyzwalanie grupy wstrzykiwanej zewnętrznym sygnałem i jednocześnie JAUTO

217 tak.. zrobili mi to w F429...

Przetworniki w obu prockach są z grubsza podobne. Mam wrażenie, że w F429 dopieszczono kilka niezbyt przemyślanych rozwiązań z F103. Prześledźmy najważniejsze różnice. Na pierwszy ogień weźmiemy parametry datasheetyczne:

- ilość przetworników: 3
- rozdzielcość: konfigurowalna (6b, 8b, 10b, 12b)
- liczba kanałów: 19/24²¹⁸ (w tym 16 zewnętrznych)
- częstotliwość zegara ADC (f_{ADC}):
 - 0,6 - 18MHz (dla V_{DDA} od 1,7V do 2,4V)
 - 0,6 - 36MHz (dla V_{DDA} od 2,4V do 3,6V)
- maksymalna rezystancja źródła mierzonego sygnału: $R_{AIN\ max} = 50k\Omega$
- rezystancja wejściowa ADC: $R_{ADC} = 6k\Omega$
- pojemność kondensatora układu próbkującego: $C_{ADC\ max} = 7pF$
- czas stabilizacji ADC po wybudzeniu: $t_{STAB\ max} = 3\mu s$
- prąd pobierany z wejścia napięcia referencyjnego: $I_{Vref\ max} = 500\mu A$
- prąd pobierany z zasilania części analogowej: $I_{VDDA\ max} = 1,8mA$
- prąd upływu pinu wejściowego: $I_{leakage\ max} = \pm 1\mu A$
- napięcie referencyjne: $1,8V \leq V_{ref+} \leq V_{DDA}$

Parametry czujnika temperatury (dalej ostrzegają że jest kiepski²¹⁹) i źródła referencyjnego:

- nachylenie charakterystyki: $Avg_slope\ typ = 2,5mV/\text{ }^{\circ}\text{C}$
- napięcie przy 25°C : $V_{25\ typ} = 0,76V$
- minimalny czas próbkowania: $t_{sample\ min} = 10\mu s$
- napięcie referencyjne: $V_{refint\ typ} = 1,21V$

Drobnej modyfikacji uległ ponadto wzorek na maksymalną rezystancję źródła mierzonego sygnału:

$$R_{AIN\ max} = \frac{k - 0.5}{f_{ADC} \cdot C_{ADC} \cdot \ln (2^{N+2})} - R_{ADC}$$

gdzie:

²¹⁸ RM podaje liczbę 19, datasheet 24...

²¹⁹ w mikrokontrolerze zaszyte są dane kalibracyjne czujnika temperatury i źródła odniesienia, patrz zadanie 13.12

- $R_{AIN \max}$ - maksymalna wartość rezystancji źródła sygnału, ścieżek, etc...
- k - liczba okresów próbkowania ustawiona w rejestrze `ADCx_SMPRy`
- f_{ADC} - częstotliwość taktowania modułu *ADC*
- C_{ADC} - pojemność kondensatora układu próbkującego (7pF)
- N - rozdzielcość przetwornika (6b / 8b / 10b / 12b)
- R_{ADC} - rezystancja wejściowa *ADC* (6k Ω)

Inne drobne różnice:

- w F103 grupę regularną można było wystartować programowo na dwa sposoby: poprzez drugie ustawienie bitu ADON lub ustawienie wyzwalania bitem SWSTART, w F429 wyrzucono tą pierwszą możliwość - teraz ADON służy tylko i wyłącznie do budzenia i usypiania *ADC*
- w F103 uruchomienie konwersji poprzez ustawienie bitów SWSTART (dla grupy regularnej) i JSWSTART (dla grupy wstrzykiwanej) było zaliczone do *external triggers* (mało logiczne); w F429 te bity działają niezależnie od wybranego trygierza i nie są zaliczane do wyzwalaczy zewnętrznych
- w F429 nie ma możliwości programowego wpływania na kalibrację *ADC*, prawdopodobnie kalibracja wykonywana jest automatycznie przy włączaniu przetwornika, ale to niepotwierdzone info
- F429 ma trzy wewnętrzne kanały *ADC*: przetwornik temperatury, źródło napięcia odniesienia i napięcie baterii podtrzymującej pamięć i RTC (do przetwornika dochodzi napięcie V_{bat} podzielone przez 4, aby zapobiec sytuacji w której $V_{bat} > V_{dda}$)
- czas mielenia wyniku przez *ADC* (nie czas próbkowania tylko to stałe 12,5 cyklu z F103) zależy od wybranej rozdzielcości przetwornika, tak się miło złożyło że mielenie wyniku trwa dokładnie tyle taktów zegara ile bitów rozdzielcości ustawiono :) Oczywiście zamiast powiedzieć to wprost w jednym zdaniu, ST postanowiło zrobić z tego osobny „tryb”: *fast conversion mode...* na szczęścia róża pachnie tak samo bez względu na to jak się nazywa.
- w F429 poza możliwością wyboru konkretnego źródła trygierza, można też wybrać zbocze na które ma reagować
- w F429 wprowadzono bit konfiguracyjny `ADCx_CR2_EOCS`, za jego pomocą można ustalić czy flaga końca konwersji (EOC) ma być ustawiana po zakończeniu wszystkich konwersji grupy (EOCS=0) czy po każdej konwersji z osobna (EOCS=1)
- rejesty danych (`ADCx_DR`) są 16b

Jest jeszcze kilka różnic wymagających więcej gadaniny. W F429 dodano coś o czym pisałem przy okazji F103 (mój pomysł :]) - flagę informującą o nadpisaniu danych w rejestrze danych (flaga *overrun* - OVR). Nadpisanie danych może też wygenerować przerwanie. Działanie funkcji *overrun* zależne jest od stanu kilku bitów konfiguracyjnych. Funkcja jest aktywna jeśli korzystamy z przesyłania wyników przez DMA lub jeśli ustawiony jest bit ADCx_CR2_EOCS, powodujący ustawianie flagi EOC po każdej konwersji (co pozwala na odbiór wyników w przerwaniu od końca konwersji).

Tabela 13.4 Aktywność funkcji *overrun*

bitы конфигурационные		<i>overrun</i>	опис
DMA	EOCS		
0	0	неактивные	одбір результатів на пішоте :)
0	1	активные	одбір результатів в прерванні
1	x	активные	одбір результатів через DMA

W momencie wystąpienia *overruna* z automatu:

- wyłączany jest strumień DMA (jeśli korzystamy z DMA)
- ADC przestaje reagować na kolejne trygierze

Zadziaływanie funkcji *overrun* gwarantuje, że dane przesłane do tej pory są prawidłowe. Aby przywrócić działanie ADC i ewentualnie DMA należy:

- skasować flagę OVR
- wyłączyć strumień DMA
- ponownie skonfigurować strumień DMA (adres i rejestr NDTR)
- włączyć strumień DMA
- odpalić ponownie ADC

Szczególnym przypadkiem jest sytuacja, w której DMA przestaje odbierać wyniki z ADC z powodu wyzerowania licznika przesyłów (NDTR). Wtedy możliwe są dwa scenariusze:

- nie chcemy więcej przesyłów, bo np. zapełniliśmy jakiś bufor próbami i zwijamy interes
- DMA pracuje w trybie kołowym i chcemy aby występowały następne przesyły

Pierwsza opcja jest o tyle kłopotliwa, że jak DMA przestanie odbierać dane z ADC to zadziała funkcja *overrun*. A niekoniecznie tego chcieliśmy... Tu z pomocą przychodzi nam bit DDS. Gdy DDS jest skasowane to ADC przerywa pracę po wyzerowaniu licznika konwersji DMA (i dzięki temu nie wyrzuci OVR). Aby wznowić pracę ADC należy skasować i ponownie ustawić bit DMA w konfiguracji przetwornika. Natomiast jeśli DDS będzie ustawione to przetwornik będzie kontynuował pracę mimo wyzerowania licznika NDTR (przydatne np. przy trybie kołowym DMA).

Kolejna spora zmiana dotyczy trybów „wielokrotnych”. W F103 był tylko *dual*, w F429 jest i *triple* (wykrakałem w poprzednim rozdziale). ADC1 dalej jest masterem. ADC2 i ADC3 to slave'y. Zanim przejdziemy do opisu tych trybów (zbrzydło mi to słowo), musimy jeszcze zerknąć na sposoby współpracy DMA z ADC w trybach wielokrotnych *dual* i *triple*. Dostępne są bowiem trzy tryby pracy o wdzięcznych i wyszukanych nazwach:

- *DMA mode 1*: żądanie DMA generowane jest po każdej konwersji. Każde żądanie powoduje przesłanie 2B zawierających pojedynczy wynik konwersji. RM zaleca używanie tego trybu przy równoległej konwersji grup regularnych trzech przetworników. Czyli np. w trybie triple kolejne transakcje DMA będą zawierały wyniki konwersji z: ADC1, ADC2, ADC3...
- *DMA mode 2*: żądanie jest generowane co dwie konwersje (gdy dostępne są dwa nowe wyniki konwersji). Podobnie jak w F103 oba wyniki lądują w jednym rejestrze. Każde żądanie powoduje przesłanie 4B zawierających dwa wyniki konwersji. Tryb polecaný do konfiguracji *interleaved* i podwójnej konwersji równoległej (*dual simultaneous...*). Przykładowo w trybie triple kolejne transakcje DMA będą zawierały wyniki: ADC2 z ADC1, ADC1 z ADC3, ADC3 z ADC2, ...
- *DMA mode 3*: to jest praktycznie to samo co tryb 2, ale dotyczy sytuacji gdy rozdzielcość przetworników jest ustawiona na 6 lub 8 bitów. Wtedy dwa wyniki konwersji mieszczą się w 2B, więc DMA nie przesyła całych 4B a jedynie 2B...

Podsumowanie macierzowe:

Tabela 13.5 Tryby współpracy DMA z ADC

tryb	żądanie generowane gdy	wielkość przesyłanych danych
mode 1	dostępny 1 wynik konwersji	2B (pół słowa)
mode 2	dostępne 2 wyniki konwersji	4B (całe słowo)
mode 3		2B (pół słowa)

Kolejna nowość przy trybach wielokrotnych to rejestrysty. Dodano specjalne rejestrysty związane z trybami wielokrotnymi. Zwracam uwagę na literkę C w nazwie rejestrów (od *common* - wspólny dla kilku przetworników). Dodano:

- „wspólny” rejestr danych, w którym lądują wyniki konwersji w trybach wielokrotnych: ADC_CDR (*Common Data Register*)
- „wspólny” rejestr konfiguracyjny, w który wrzucono bity związane z konfiguracją trybów wielokrotnych i to co nie zmieściło się gdzie indziej: ADC_CCR (*Common Control Register*)
- „wspólny” rejestr statusowy (tylko do odczytu), w którym zamieszczono kopie wszystkich flag poszczególnych przetworników ADC: ADC_CSR (*Common Status Register*)

Tryby... generalnie są te same:

- *simultaneous regular/injected* - czyli równoległe przetwarzanie grup przez dwa lub trzy przetworniki
- *interleaved mode* - czyli próbkowanie jednego kanału przez kilka przetworników. Zrezygnowano z podziału na wersję *slow* i *fast*. Zamiast tego wprowadzono możliwość konfigurowania opóźnienia (ADC_CCR_DELAY). Przy czym jeśli ustawione zostanie opóźnienie krótsze niż czas konwersji kanału, to zostanie ono automatycznie wydłużone na czas próbkowania + 2 cykle zegara ADC.
- *alternate trigger* - naprzemienne przetwarzanie grup wstrzykiwanych przez dwa lub trzy przetworniki
- *injected simultaneous + regular simultaneous* - czyli przerwanie równolegle przetwarzanych grup regularnych przez wstrzykiwane
- *regular simultaneous + alternate trigger* - jw. ale grupy wstrzykiwane odpalone są naprzemienne w kilku przetwornikach

Przydałoby się trochę przykładów, nieprawdaż :) Trywialne przykłady sobie darujemy... bo są trywialne :) Oblecimy te choć odrobine ciekawsze.

Zadanie domowe 13.9: pojedynczy przetwornik ADC konwertuje dwa kanały w grupie regularnej bez przerwy. Ponadto co 1s wykonuje pomiar trzeciego kanału i migi wtedy diodą. Program napisać w trzech wersjach:

- wyniki konwersji grupy regularnej przesyłane przez DMA do dwuelementowej tablicy (cały czas)

- wyniki konwersji grupy regularnej przesyłane przez DMA do dwustu-elementowej tablicy (tylko raz, do zapełnienia tablicy)
- wyniki konwersji grupy regularnej odczytywane w przerwaniu do dwóch różnych zmiennych

Przykładowe rozwiążanie - wariant 1 (F429, pomiar na PA5, PA7, PC3, dioda na PG13):

```

1. int main(void) {
2.
3.     static volatile uint16_t wyniki[2];
4.
5.     RCC->AHB1ENR |= RCC_AHB1ENR_GPIOGEN | RCC_AHB1ENR_GPIOAEN | RCC_AHB1ENR_GPIOCEN |
6.                 RCC_AHB1ENR_DMA2EN;
7.     RCC->APB2ENR = RCC_APB2ENR_ADC1EN;
8.     RCC->APB1ENR = RCC_APB1ENR_TIM2EN;
9.     __DSB();
10.
11.    gpio_pin_cfg(GPIOA, PA5, gpio_mode_analog);
12.    gpio_pin_cfg(GPIOA, PA7, gpio_mode_analog);
13.    gpio_pin_cfg(GPIOC, PC3, gpio_mode_analog);
14.    gpio_pin_cfg(GPIOG, PG13, gpio_mode_out_PP_LS);
15.
16.    TIM2->PSC = 8000-1;
17.    TIM2->ARR = 2000-1;
18.    TIM2->CR2 = TIM_CR2_MMS_1;
19.    TIM2->EGR = TIM_EGR_UG;
20.
21.    DMA2_Stream0->MQAR = (uint32_t)wyniki;
22.    DMA2_Stream0->PAR = (uint32_t)&ADC1->DR;
23.    DMA2_Stream0->NDTR = 2;
24.    DMA2_Stream0->CR = DMA_SxCR_MSIZE_0 | DMA_SxCR_PSIZE_0 | DMA_SxCR_MINC | DMA_SxCR_CIRC |
25.                      DMA_SxCR_EN;
26.
27.    ADC1->CR2 = ADC_CR2_ADON | ADC_CR2_CONT | ADC_CR2_JEXTEN_0 | ADC_CR2_JEXTSEL_0 |
28.                 ADC_CR2_JEXTSEL_1 | ADC_CR2_DDS | ADC_CR2_DMA;
29.    ADC1->CR1 = ADC_CR1_SCAN | ADC_CR1_JE0CIE;
30.    ADC1->SMPR1 = 7<<9;
31.    ADC1->SMPR2 = 7<<21 | 7<<15;
32.    ADC1->SQR1 = 1<<20;
33.    ADC1->SQR3 = 5<<0 | 7<<5;
34.    ADC1->JSQR = 13<<15;
35.
36.    NVIC_EnableIRQ(ADC_IRQn);
37.
38.    ADC1->CR2 |= ADC_CR2_SWSTART;
39.    TIM2->CR1 |= TIM_CR1_CEN;
40.
41.    while (1);
42.
43. } /* main */
44.
45. void ADC_IRQHandler(void){
46.     if (ADC1->SR & ADC_SR_JEOC){
47.         ADC1->SR &= ~ADC_SR_JEOC;
48.         GPIOG->ODR ^= PG13;
49.     }
50. }
```

No i jakiś ładny kodzik :) Początek nie powinien budzić wątpliwości. Tablica na wyniki, zegary, konfiguracja pinów. Zwracam uwagę na to, że rejestr RCC_AHB1ENR domyślnie nie jest równy zero, stąd suma bitowa w linii 5. Następnie jest konfiguracja licznika, tak aby wyrzucał UEV jako TRGO co 1s, bułka z masłem :) Konfiguracja DMA. Dwa 16b przesyły z rejestru danych ADC do tablicy, z inkrementacją po stronie pamięci i w trybie kołowym. Swoją drogą chyba wolę DMA w F103. Aha! Przy konfiguracji strumienia DMA trzeba wybrać kanał (czyli źródło żądań), odsyłam

do tabeli *DMAx request mapping*. Tak się, mało dydaktycznie złożyło, że ADC1 to kanał 0 dla strumienia 0 kontrolera DMA2 i nic nie trzeba ustawiać bo to domyślna opcja.

27) dojechaliśmy do konfiguracji ADC. Z ciekawych rzeczy jest tutaj:

- włączenie przetwornika (darowałem sobie opóźnienie po ustawieniu bitu ADON, bo przed uruchomieniem konwersji jest jeszcze kilka operacji które wprowadzą pewne opóźnienie; poza tym to tylko przykład i nie ma co rozwlekać)
- włączenie wyzwalania grupy wstrzykiwanej zewnętrznym trygierzem, a dokładniej zboczem rosnącym. W przypadku sygnału o UEV (impuls) wybór zbocza nie ma wielkiego znaczenia, ale: gdybyśmy np. wyzwalali ADC sygnałem *Compare* to już by była inna historia
- wybór źródła trygierza: odsyłam do opisu rejestru ADCx_CR2
- CONT, DMA - to chyba nie budzi wątpliwości
- DDS - domyślnie, po wyzerowaniu rejestru NDTR (czyli kiedy DMA przestaje odbierać dane), przetwornik ADC przestaje generować żądania DMA i dzięki temu nie wywala *overrun*. W takim układzie nie dałoby się zrealizować trybu kołowego DMA, bo po „przekręceniu” NDTR przetwornik ADC by się blokował. Ustawienie bitu DDS powoduje, że ADC nie przejmuje się NDTRem w DMA i działa dalej w najlepsze.

29) włączenie przerwania od zakończenia konwersji grupy wstrzykiwanej oraz trybu wielokanałowego

30, 31) ustawiam czas próbkowania na jakiś tam... bo czemu by nie :)

32, 33) dwie konwersje regularne (kanały 5 i 7)

34) jedna konwersja tryskawkowa (kanał 13)

Następnie jest włączenie przerwania, ADC, licznika. Zwracam uwagę na inne nazwy i wektory przerwań niż w F103. W przerwaniu kasuję flagę i macham diodą.

W wyniku działania programu:

- wynik konwersji wstrzykiwanej (IN13) ląduje w rejestrze ADC1_JDR1
- wyniki konwersji regularnych lądują w tablicy *wyniki*, przy czym:
 - *wyniki[0]* = IN5
 - *wyniki[1]* = IN7

Przykładowe rozwiązanie - wariant 2 (F429, dioda na PG13, wejścia analogowe: PA5, PA7, PC3):

```
1. int main(void) {
2.
3.     static volatile uint16_t wyniki[200];
4.
5.     RCC->AHB1ENR |= RCC_AHB1ENR_GPIOGEN | RCC_AHB1ENR_GPIOAEN | RCC_AHB1ENR_GPIOCEN |
6.                 RCC_AHB1ENR_DMA2EN;
7.     RCC->APB2ENR = RCC_APB2ENR_ADC1EN;
8.     RCC->APB1ENR = RCC_APB1ENR_TIM2EN;
9.     __DSB();
10.
11.    gpio_pin_cfg(GPIOA, PA5, gpio_mode_analog);
12.    gpio_pin_cfg(GPIOA, PA7, gpio_mode_analog);
13.    gpio_pin_cfg(GPIOC, PC3, gpio_mode_analog);
14.    gpio_pin_cfg(GPIOG, PG13, gpio_mode_out_PP_LS);
15.
16.    TIM2->PSC = 8000-1;
17.    TIM2->ARR = 2000-1;
18.    TIM2->CR2 = TIM_CR2_MMS_1;
19.    TIM2->EGR = TIM_EGR_UG;
20.
21.    DMA2_Stream0->M0AR = (uint32_t)wyniki;
22.    DMA2_Stream0->PAR = (uint32_t)&ADC1->DR;
23.    DMA2_Stream0->NDTR = 200;
24.    DMA2_Stream0->CR = DMA_SxCR_MSIZE_1 | DMA_SxCR_PSIZE_0 | DMA_SxCR_MINC | DMA_SxCR_EN;
25.    DMA2_Stream0->FCR = DMA_SxFCR_DMDIS | DMA_SxFCR_FTH_0;
26.
27.    ADC1->CR2 = ADC_CR2_ADON | ADC_CR2_CONT | ADC_CR2_JEXTEN_0 | ADC_CR2_JEXTSEL_0 |
28.                  ADC_CR2_JEXTSEL_1 | ADC_CR2_DMA;
29.    ADC1->CR1 = ADC_CR1_SCAN | ADC_CR1_JEOCIE;
30.    ADC1->SMPR1 = 7<<9;
31.    ADC1->SMPR2 = 7<<21 | 7<<15;
32.    ADC1->SQR1 = 1<<20;
33.    ADC1->SQR3 = 5<<0 | 7<<5;
34.    ADC1->JSQR = 13<<15;
35.
36.    NVIC_EnableIRQ(ADC_IRQn);
37.
38.    ADC1->CR2 |= ADC_CR2_SWSTART;
39.    TIM2->CR1 |= TIM_CR1_CEN;
40.
41.    while (!(DMA2->LISR & DMA_LISR_TCIF0)) __WFI();
42.    __BKPT();
43.
44. } /* main */
45.
46. void ADC_IRQHandler(void){
47.     if (ADC1->SR & ADC_SR_JEOC){
48.         ADC1->SR &= ~ADC_SR_JEOC;
49.         GPIOG->ODR ^= PG13;
50.     }
51. }
```

Cóż się zmieniło:

- rozmiar tablicy i ilość przesyłów DMA - wiadomo, wynika to z treści zadania
- konfiguracja DMA: po stronie peryferiala dalej są odczytu 16b, ale po stronie pamięci ustawilem na 32b żeby było ciekawiej²²⁰. Włączyłem do tego bufor FIFO.
- w konfiguracji DMA wyłączyłem tryb kołowy (bo tablica ma się zapełnić tylko jeden raz - takie założenie zadania)

220 można zostawić *direct mode* i rozmiary 16b po obu stronach - też będzie dobrze działać

- z konfiguracji ADC wyrzuciłem ustawianie bitu DDS: nie korzystamy z trybu kołowego, więc nie chcemy kolejnych żądań DMA po wykonaniu wszystkich przesyłów (zapobiega to wykryciu *overrun/nadpisania* rejestru danych)
- na końcu programu jest pętla oczekująca na flagę końca transmisji strumienia DMA, po zakończeniu przesyłów (zapełnieniu tablicy) program jest przerywany (*breakpoint*)
- w pętli oczekiwania na koniec transmisji wrzuciłem usypianie procka, dla urozmaicenia przykładu

W wyniku działania programu:

- wynik konwersji wstrzykiwanej (IN13) ląduje w rejestrze ADC1_JDR1
- wyniki konwersji regularnych lądują w tablicy *wyniki*, przy czym:
 - *wyniki[0]* = IN5
 - *wyniki[1]* = IN7
 - *wyniki[2]* = IN5
 - *wyniki[3]* = IN7
 - ...

Dla chętnych: sprawdzenie czy po ustawieniu bitu DDS (żądania DMA będą dalej generowane po skończeniu transmisji) zadziała funkcja *overrun*.

Przykładowe rozwiązanie - wariant 3 (F429, dioda na PG13, wejścia analogowe: PA5, PA7, PC3):

```
1. volatile uint16_t wynik1, wynik2;
2.
3. int main(void) {
4.
5.     RCC->AHB1ENR |= RCC_AHB1ENR_GPIOGEN | RCC_AHB1ENR_GPIOAEN | RCC_AHB1ENR_GPIOCEN;
6.     RCC->APB2ENR = RCC_APB2ENR_ADC1EN;
7.     RCC->APB1ENR = RCC_APB1ENR_TIM2EN;
8.     __DSB();
9.
10.    gpio_pin_cfg(GPIOA, PA5, gpio_mode_analog);
11.    gpio_pin_cfg(GPIOA, PA7, gpio_mode_analog);
12.    gpio_pin_cfg(GPIOC, PC3, gpio_mode_analog);
13.    gpio_pin_cfg(GPIOG, PG13, gpio_mode_out_PP_LS);
14.
15.    TIM2->PSC = 8000-1;
16.    TIM2->ARR = 2000-1;
17.    TIM2->CR2 = TIM_CR2_MMS_1;
18.    TIM2->EGR = TIM_EGR_UG;
19.
20.    ADC1->CR2 = ADC_CR2_ADON | ADC_CR2_CONT | ADC_CR2_JEXTEN_0 | ADC_CR2_JEXTSEL_0 |
21.                ADC_CR2_JEXTSEL_1 | ADC_CR2_EOCS;
22.    ADC1->CR1 = ADC_CR1_SCAN | ADC_CR1_JEOCIE | ADC_CR1_EOCIE;
23.    ADC1->SMPR1 = 7<<9;
24.    ADC1->SMPR2 = 7<<21 | 7<<15;
25.    ADC1->SQR1 = 1<<20;
26.    ADC1->SQR3 = 5<<0 | 7<<5;
27.    ADC1->JSQR = 13<<15;
28.
29.    NVIC_EnableIRQ(ADC_IRQn);
30.
31.    ADC1->CR2 |= ADC_CR2_SWSTART;
32.    TIM2->CR1 |= TIM_CR1_CEN;
33.
34.    for(volatile uint32_t delay = 1000000; delay; delay--) __NOP();
35.    ADC1->CR2 &= ~ADC_CR2_ADON;
36.    __BKPT();
37. } /* main */
38.
39. void ADC_IRQHandler(void){
40.     if (ADC1->SR & ADC_SR_JEOC){
41.         ADC1->SR &= ~ADC_SR_JEOC;
42.         GPIOG->ODR ^= PG13;
43.     }
44.
45.     if (ADC1->SR & ADC_SR_EOC){
46.         ADC1->SR &= ~ADC_SR_EOC;
47.         static uint32_t flaga;
48.         if (flaga) wynik1 = ADC1->DR;
49.         else wynik2 = ADC1->DR;
50.         flaga ^= 1;
51.     }
52. }
```

Przypominam, że w tym przykładzie wyniki konwersji grupy regularnej mają być, w przerwaniu, zapisywane do dwóch zmiennych. Cóż tu mamy ciekawego:

- zmienne na wyniki są globalne... bo przerwanie
- wyleciało DMA :)
- w konfiguracji ADC pojawiło się ustawienie bitu EOCS, powoduje on że flaga końca konwersji (EOC) jest ustawiana po zakończeniu każdej pojedynczej konwersji z grupy. Bez tego bitu, flaga byłaby ustawiana po zakończeniu konwersji całej grupy. W tym przykładzie wyniki są odczytywane w przerwaniu. Przerwanie jest wywoływana po ustawieniu flagi EOC. Wyniki

musimy odczytywać po każdej konwersji, inaczej się nadpiszą. Dlatego też chcemy mieć flagę (i przerwanie) po każdej konwersji a nie po całej sekwencji z grupy.

22) doszło włączenie przerwania od flagi EOC

34 - 36) jakiś dziwoląg: opóźnienie, wyłączenie ADC i *breakpoint*. O co chodzi? Dlaczego nie ma po prostu nieskończonej pętli DUŚ²²¹? Bo po zatrzymaniu procka debuggerem, ADC sygnalizował nadpisanie rejestru danych (*overrun*). Przy odbieraniu danych przez DMA (wcześniejsze przykłady) tego problemu nie było. Tutaj dane są odbierane w przerwaniu i taka ciekawostka się zadziała. Żeby więc nie musieć zatrzymywać procka debuggerem, dorzuciłem to co widać.

45) w ISR przybył nowy warunek. Dane są odczytywane z rejestru ADC1_DR i zapisywane do dwóch zmiennych. Niestety nie ma żadnego sposobu, aby sprawdzić z którego kanału pochodzą dane w rejestrze DR. Z tego względu trzeba „programowo” pilnować wpisywania kolejnych wyników do odpowiednich zmiennych, stąd kombinacje ze zmienną *flaga*.

Dla dociekliwych: sprawdzić co się stanie jeśli nie zostanie ustawiony bit EOCS.

Zadanie domowe 13.10: uruchomić tryb *dual simultaneous regular*.... czyli żeby dwa przetworniki równolegle konwertowały grupy regularne. Po jednym kanale na przetwornik wystarczy. Wyniki przesyłane przez DMA do 200 elementowej tablicy (raz). Program napisać w dwóch wersjach:

- z wykorzystaniem *DMA mode 1*
- z wykorzystaniem *DMA mode 2*

221 Do Usianej Śmierci

Przykładowe rozwiązanie - wariant 1 (F429, wejścia analogowe: PA5, PA7):

```
1. int main(void) {
2.
3.     static volatile uint16_t wyniki[200];
4.
5.     RCC->AHB1ENR |= RCC_AHB1ENR_GPIOAEN | RCC_AHB1ENR_DMA2EN;
6.     RCC->APB2ENR = RCC_APB2ENR_ADC1EN | RCC_APB2ENR_ADC2EN;
7.     __DSB();
8.
9.     gpio_pin_cfg(GPIOA, PA5, gpio_mode_analog);
10.    gpio_pin_cfg(GPIOA, PA7, gpio_mode_analog);
11.
12.    DMA2_Stream0->M0AR = (uint32_t)wyniki;
13.    DMA2_Stream0->PAR = (uint32_t)&ADC->CDR;
14.    DMA2_Stream0->NDTR = 200;
15.    DMA2_Stream0->CR = DMA_SxCR_MSIZE_0 | DMA_SxCR_PSIZE_0 | DMA_SxCR_MINC | DMA_SxCR_CIRC |
16.        DMA_SxCR_EN;
17.
18.    ADC->CCR = ADC_CCR_DMA_0 | ADC_CCR_MULTI_1 | ADC_CCR_MULTI_2;
19.
20.    ADC1->CR2 = ADC_CR2_ADON | ADC_CR2_CONT;
21.    ADC2->CR2 = ADC_CR2_ADON | ADC_CR2_CONT;
22.    ADC1->SQR3 = 5<<0;
23.    ADC2->SQR3 = 7<<0;
24.
25.    ADC1->CR2 |= ADC_CR2_SWSTART;
26.
27.    while(!(DMA2->LISR & DMA_LISR_TCIF0)) ;
28.    __BKPT();
29. } /* main */
```

Tu jest trochę subtelnych nowości związanych z trybem podwójnym (*dual*):

13) zmienił się rejestr z którego odbieramy wyniki konwersji. W F103 wyniki konwersji w trybie podwójnym ląowały w rejestrze mastera. W F429 wprowadzono specjalne rejesty dla trybów wielokrotnych. Jednym z nich jest ADC_CDR (*Common Data Register*). W nim zapisywane są wyniki konwersji w trybach wielokrotnych. Żeby było weselej, wyniki konwersji poszczególnych przetworników można dalej odczytać z ich „osobistych” rejestrów danych.

15) tutaj się odrobinę gubię i właściwie nie wiem czemu to działa. W *DMA mode 1* (założenia przykładu), żądanie jest generowane po każdej konwersji. Wynik konwersji powinien być zapisany w rejestrze ADC_CDR. Według opisu rejestru wyniki z poszczególnych przetworników powinny być zapisywane w dwóch połówkach rejestrów. Przykład działa zgodnie z założeniami przy rozmiarze po stronie peryferiala (w konfiguracji DMA) - 16b. Na logikę DMA ustawione na 16b powinno cały czas odczytywać wyniki tylko jednej połówki rejestrów ADC_CDR (jednego przetwornika). A jednak kod działa! Przynajmniej, że doszedłem do tego metodą prób i trochę błędów²²². Także ten...

18) do konfiguracji wielokrotnych ADcCe, służy specjalny rejestr ADC_CCR (*Common Control Register*). Siedzą w nim m.in. bity odpowiedzialne za wybór konkretnego trybu wielokrotnego i trybu działania DMA.

20 - 23) włączam dwa przetworniki, tryb ciągły, ustawiam kanały. I wio!

222 staram się jak najrzadziej, ale czasem tak jest najszybciej...

27) po zapełnieniu tablicy program jest przerywany

Dla dociekliwych: co się stanie jeśli w konfiguracji tylko jednego przetwornika zostanie włączony tryb ciągły?

Dla dociekliwych (bis): sprawdzić czy w trybie podwójnym, jest możliwe użycie dwóch różnych strumieni DMA do przesyłania danych z rejestrów danych przetworników (ADC1_DR i ADC2_DR) do różnych buforów?

Przykładowe rozwiązań - wariant 2 (F429, wejścia analogowe: PA5, PA7):

```
1. int main(void) {
2.
3.     static volatile uint16_t wyniki[200];
4.
5.     RCC->AHB1ENR |= RCC_AHB1ENR_GPIOAEN | RCC_AHB1ENR_DMA2EN;
6.     RCC->APB2ENR = RCC_APB2ENR_ADC1EN | RCC_APB2ENR_ADC2EN;
7.     __DSB();
8.
9.     gpio_pin_cfg(GPIOA, PA5, gpio_mode_analog);
10.    gpio_pin_cfg(GPIOA, PA7, gpio_mode_analog);
11.
12.    DMA2_Stream0->M0AR = (uint32_t)wyniki;
13.    DMA2_Stream0->PAR = (uint32_t)&ADC->CDR;
14.    DMA2_Stream0->NDTR = 100;
15.    DMA2_Stream0->CR = DMA_SxCR_MSIZE_1 | DMA_SxCR_PSIZE_1 | DMA_SxCR_MINC | DMA_SxCR_CIRC |
16.        DMA_SxCR_EN;
17.
18.    ADC->CCR = ADC_CCR_DMA_1 | ADC_CCR_MULTI_1 | ADC_CCR_MULTI_2;
19.
20.    ADC1->CR2 = ADC_CR2_ADON | ADC_CR2_CONT;
21.    ADC2->CR2 = ADC_CR2_ADON | ADC_CR2_CONT;
22.    ADC1->SQR3 = 5<<0;
23.    ADC2->SQR3 = 7<<0;
24.
25.    ADC1->CR2 |= ADC_CR2_SWSTART;
26.
27.    while(!(DMA2->LISR & DMA_LISR_TCIF0)) ;
28.    __BKPT();
29. } /* main */
```

Tym razem założono wykorzystanie DMA w trybie *Mode 2*. Czyli generowane będzie jedno żądanie DMA, gdy dostępne będą wyniki dwóch konwersji ADC. Cóż się zmieniło w kodzie:

- liczba transakcji w konfiguracji DMA spadła o połowę, gdyż każdy przesył DMA będzie zawierał dwa wyniki konwersji. Stąd do zapełnienia tablicy 200-elementowej potrzebna tylko 100 przesyłów.
- rozmiary danych przesyłanych przez DMA. W poprzednim przykładzie przesyłany był pojedynczy wynik konwersji, który mieścił się w połowie rejestru (16b). Teraz przesyłane są dwa wyniki zajmujące cały rejestr (32b).
- w konfiguracji ADC wybrany został inny tryb DMA

Efekt działania tego programu jest identyczny jak poprzedniego. Różnica polega tylko na tym, że w pierwszej wersji, żądanie przesyłu DMA było generowane po każdej konwersji. Natomiast w drugim wariantie żądanie jest generowane po zakończeniu dwóch konwersji.

Zadanie domowe 13.11: uruchomić tryb *triple simultaneous regular*.... czyli żeby wszystkie trzy przetworniki, równolegle konwertowały grupy regularne. Po jednym kanale na przetwornik wystarczy. Wyniki przesyłane przez DMA do 9-cio elementowej tablicy tak, aby zawsze były w niej po trzy, najświeższe, wyniki konwersji każdego z kanałów.

Przykładowe rozwiązanie (F429, wejścia analogowe PA5, PA7, PC3):

```
1. int main(void) {
2.
3.     static volatile uint16_t wyniki[9];
4.
5.     RCC->AHB1ENR |= RCC_AHB1ENR_GPIOCEN | RCC_AHB1ENR_GPIOAEN | RCC_AHB1ENR_DMA2EN;
6.     RCC->APB2ENR = RCC_APB2ENR_ADC1EN | RCC_APB2ENR_ADC2EN | RCC_APB2ENR_ADC3EN;
7.     __DSB();
8.
9.     gpio_pin_cfg(GPIOC, PC3, gpio_mode_analog);
10.    gpio_pin_cfg(GPIOA, PA5, gpio_mode_analog);
11.    gpio_pin_cfg(GPIOA, PA7, gpio_mode_analog);
12.
13.    DMA2_Stream0->MOAR = (uint32_t)wyniki;
14.    DMA2_Stream0->PAR = (uint32_t)&ADC->CDR;
15.    DMA2_Stream0->NDTR = 9;
16.    DMA2_Stream0->CR = DMA_SxCR_MSIZE_0 | DMA_SxCR_PSIZE_0 | DMA_SxCR_MINC | DMA_SxCR_CIRC |
17.        DMA_SxCR_EN;
18.
19.    ADC->CCR = 0b10110 | ADC_CCR_DMA_0 | ADC_CCR DDS;
20.
21.    ADC1->CR2 = ADC_CR2_ADON | ADC_CR2_CONT;
22.    ADC2->CR2 = ADC_CR2_ADON | ADC_CR2_CONT;
23.    ADC3->CR2 = ADC_CR2_ADON | ADC_CR2_CONT;
24.    ADC1->SQR3 = 5<<0;
25.    ADC2->SQR3 = 7<<0;
26.    ADC3->SQR3 = 13<<0;
27.
28.    ADC1->CR2 |= ADC_CR2_SWSTART;
29.
30.    while(1);
31.
32. } /* main */
```

6) włączenie taktowania wszystkich trzech przetworników ADC

15) przy konfiguracji ilości przesyłów DMA trzeba się zastanowić w jakim trybie DMA będzie to pracowało. Najwygodniejszy wydaje się być tryb 1 (żądanie DMA po każdej konwersji). Czyli przesyłane będą pojedyncze wyniki (16b) i będzie 9 przesyłów do zapełnienia tablicy.

19) wybór trybu (*triple simultaneous regular*), trybu DMA (*mode 1*) i włączenie bitu DDS żeby umożliwić pracę z buforem kołowym (*DMA circular mode*).

Dalej jest już nuda. Wybudzenie przetworników, włączenie im trybów ciągłych, ustawienie kanałów i wio! Przypominam, że przy wyborze kanałów trzeba uważać! Różne kanały mogą pracować z określonymi przetwornikami (np. ADC12_INx - nie będzie działał z przetwornikiem 3).

W wyniku działania programu, otrzymujemy:

- `wyniki[0] = IN5`
- `wyniki[1] = IN7`
- `wyniki[2] = IN13`
- `wyniki[3] = IN5`
- `wyniki[4] = IN7`
- `wyniki[5] = IN13`
- ...

Dla chętnych: *triple interleaved*. Niech trzy przetworniki próbują ten sam kanał (jedna taka sekwencja), a wyniki niech wylądują w trzy elementowej tablicy... mnie się nie udało zmusić ADC żeby generował żądanie DMA po każdej konwersji w *triple interleaved*. Może Tobie się uda!

Nie ma co się bać specjalnie tego wszystkiego. Mając do dyspozycji debugger i trochę zaparcia (prawie) wszystko da się zrobić. Nie ukrywam, że czasem kombinowałem metodą prób i błędów... Szczególnie przy rozgryzaniu sytuacji, w której kilka przetworników przetwarza kilka kanałów i to leci przez DMA do tablicy. Warto sobie podpiąć testowe sygnały po kolej do każdego kanału i upewnić się, że wyniki są tam gdzie się się spodziewamy :)

Dobra, pomału dosyć z tym ADC... ale pozostajemy w sferze analogowej.

Co warto zapamiętać z tego rozdziału?

- automatyczna kalibracja po włączeniu
- bit EN służy tylko do wybudzania/usypiania przetwornika
- czas pomiaru zależy od wybranej rozdzielczości
- funkcja *overrun* i bit DDS

13.8. Różnice w STM32F334 (F334)

Tytułowy mikrokontroler posiada dwa przetworniki ADC mogące obsłużyć do 18 multipleksowanych kanałów. Kanały dzielą się na zewnętrzne (21 nóżek mogących współpracować z ADC) i wewnętrzne (4 kanały). Kanały wewnętrzne to:

- wbudowany czujnik temperatury (tylko ADC1)

- napięcie baterii podtrzymującej *dominium batoryjne* (tylko ADC1, napięcie z nóżki V_{bat} jest dzielone /2 przed podaniem na przetwornik aby nie przekroczyło dopuszczalnych granic)
- wewnętrzne źródło napięcia odniesienia $V_{refint\ typ} = 1,2V$ (oba przetworniki)
- wyjściowe napięcie referencyjne wbudowanego wzmacniacza OPAMP2 (nie wiem, nie pytać, jeszcze nie doczytałem)

Dodatkowo kanały zewnętrzne dzielą się na szybkie i wolne. Kanały szybkie (ADCx_IN1...5) umożliwiają konwersję w czasie $0,19\mu s$ przy 12-bitowej rozdzielczości (5,1Ms/s). Kanały wolne zaś: $0,21\mu s$ (4,8Ms/s).

Przetwornik umożliwia programową zmianę rozdzielczości (6, 8, 10, 12 bitów), tak jak w F429. Im mniejsza rozdzielcość, tym szybszy pomiar. Przetwornik taktowany jest zegarem o częstotliwości z zakresu 0,14 - 72MHz. Zegar dla bloku ADC może pochodzić z dwóch źródeł (wyjaśni się w rozdziale 17.4):

- szyny AHB (HCLK) - to rozwiązanie pozwala uniknąć problemów wynikających z synchronizacji dwóch domen zegarowych oraz zapewnia precyzyjne wyzwalanie przetwornika
- „niezależnego” źródła (z SYSCLK) - to rozwiązanie uniezależnia czas konwersji od aktualnej prędkości zegara AHB

Przy czym, jeżeli wykorzystywane są kanały wstrzykiwane, to należy spełnić dodatkowe obostrzenia dotyczące sygnału zegarowego. Odsyłam do rozdziału *Analog-to-digital converters (ADC) → ADC functional description → Clocks* i sekcji *Clock ratio constraint between ADC clock and AHB clock* w RMie.

Konfiguracja kanałów, podział na grupy zwyczajne i wstrzykiwane, tryby działania (pojedyncze i podwójne) generalnie działają bez zmian w stosunku do poprzednio omówionych mikrokontrolerów. Nie ma sensu powtarzać tego po raz n-ty.

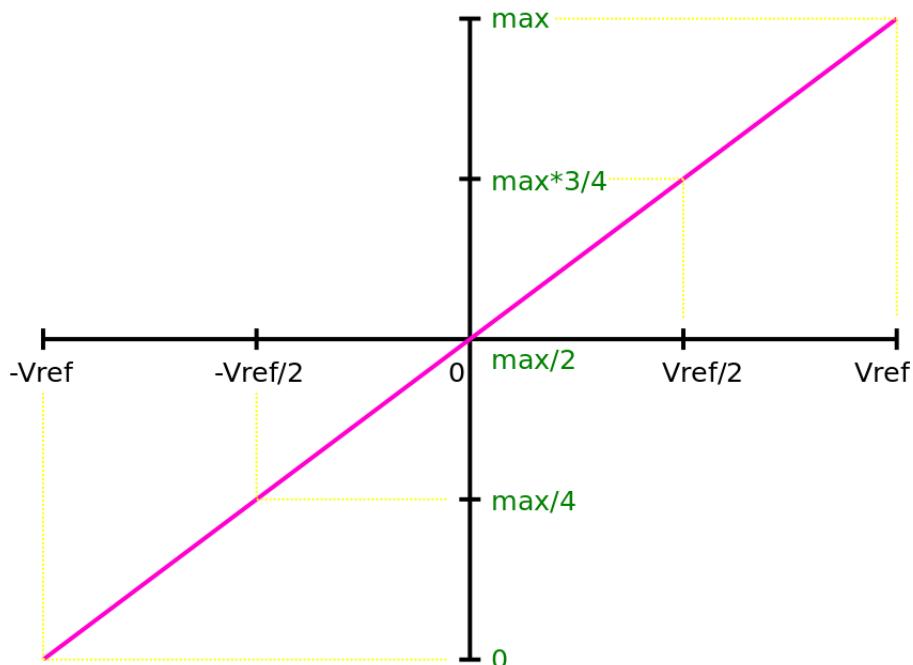
Sporej zmianie uległ sposób włączania przetwornika. W pierwszym kroku należy uruchomić stabilizator napięcia przetwornika ADC (patrz bity ADVREGEN w ADCx_CR). Następnie należałoby (obowiązku nie ma) przeprowadzić kalibrację przetwornika. Na samym końcu można włączyć przetwornik bitem ADEN. Warto zwrócić uwagę na to, że spora część bitów odpowiedzialnych za konfigurację/sterowanie przetwornikiem (patrz rejestr ADCx_CR) to bity, które można jedynie ustawić lub odczytać (oznaczenia *rs* - *read/set*). Przykładem może być bit włączający przetwornik - ADEN. Nie można go skasować poprzez wpisanie zera (bo *rs*), więc za jego pomocą nie można wyłączyć przetwornika! Do wyłączania przetwornika służy bit ADDIS.

Ustawienie go, uruchamia „procedurę” wyłączania przetwornika, co m.in. spowoduje skasowanie bitu ADEN.

Całkowitą nowością jest możliwość konfiguracji kanałów w trybie różnicowym. Omówione dotychczas przetworniki mikrokontrolerów F103 i F429 mogły pracować jedynie w trybie „pojedynczym” (*single ended*). Napięcie między mierzonym kanałem a masą analogową (V_{SSA}) było przyrównywane do napięcia odniesienia/referencyjnego (między V_{ref+} a V_{ref-}). Tryb różnicowy (*differential input*) działa w ten sposób, że przetwornik porównuje, z napięciem odniesienia, różnicę napięć pomiędzy dwoma kanałami - kanałem „dodatnim” (V_{in+}) i „ujemnym” (V_{in-}). Napięcie różnicowe (V_{diff}) dane jest wzorem:

$$V_{diff} = V_{in+} - V_{in-}$$

Może ono przyjmować wartości dodatnie i ujemne w zależności od relacji napięć obu kanałów. Zależność między wartością napięcia różnicowego a wynikiem konwersji ADC pokazana została na wykresie z rysunku 13.5.



Rys. 13.5. Zależność między wynikiem konwersji ADC (os Y) a napięciem różnicowym (os X)

Wynik konwersji jest równy zero dla napięcia różnicowego $-V_{ref}$ ²²³. W miarę zwiększania wartości V_{diff} wartość wyniku konwersji rośnie „w miarę” liniowo. Przy zerowym napięciu różnicowym wynik konwersji wynosi około połowę rozdzielczości przetwornika, np. dla

223 w zestawie Nucleo napięcie odniesienia wynosi 3,3V

rozdzielczości 12 bitowej wyniesie w przybliżeniu 2048. Maksymalna wartość wyniku (4095 dla rozdzielczości 12 bit) uzyskiwana jest gdy $V_{diff} = V_{ref}$. Przerabiając powyższe na formułkę wychodzi coś takiego:

$$adc = \frac{ADC_{max}}{2} \cdot \left(\frac{V_{diff}}{V_{ref}} + 1 \right)$$

lub w drugą stronę:

$$V_{diff} = \left(2 \cdot \frac{adc}{ADC_{max}} - 1 \right) \cdot V_{ref}$$

Włączenie konfiguracji różnicowej na i-tym kanale (ADCx_INi, patrz rejestr ADCx_DIFSEL) powoduje, że przetwornik będzie mierzył różnicę napięć między tym kanałem (wejście „dodatnie”) i kanałem o numerze $i+1$ (wejście „ujemne”/zanegowane). Tzn.:

$$V_{diff} = V_{in^+} - V_{in^-} = V_{in\ i} - V_{in\ i+1}$$

Uruchomienie konwersji kanału i wyzwoli konwersję w trybie różnicowym. Jeżeli korzystasz z trybu różnicowego to kanał $i+1$, wykorzystywany jako wejście „ujemne”, nie może być „bezpośrednio” konwertowany. Innymi słowy - nie należy uruchamiać bezpośrednio konwersji tego kanału.

Przetwornik ADC posiada funkcję automatycznej kalibracji. Kalibrację należy uruchamiać ręcznie (przynajmniej tyle zaleca RM):

- po włączeniu przetwornika
- po zmianie warunków pracy (np. napięcia odniesienia)
- po wybudzeniu z trybu *standby* (wartości kalibracyjne są tracone)

Wartości kalibracyjne dostępne są w rejestrze ADCx_CALFACT. Po przeprowadzeniu kalibracji przetwornika można je odczytać i zapisać (np. w rejestrach BKP). Dzięki temu, dajmy na to po wybudzeniu mikrokontrolera z trybu standby, nie trzeba będzie przeprowadzać ponownej kalibracji (trwa 112 cykli zegara ADC). Zamiast tego będzie można wpisać do rejestru, zapisane wcześniej, wartości kalibracyjne,

Dane kalibracyjne zawierają dwie wartości (CALFACT_S i CALFACT_D), oddzielnie dla trybów *single-ended* i *differential*. Jeżeli w programie wykorzystywane będą oba tryby pracy przetwornika, to należy przeprowadzić kalibrację dwukrotnie - raz w celu skalibrowanie trybu pojedynczego, drugi raz dla trybu różnicowego. Za wybór trybu kalibracji odpowiada bit ADCx_CR_ADCCALDIF. W czasie pracy przetwornika, wybór odpowiedniej wartości kalibracyjnej, następuje automatycznie.

Dalej mamy kilka mniejszych zmian:

- jest trochę więcej flag, m.in.:
 - EOSMP - koniec próbowania kanału
 - EOC - koniec pojedynczej konwersji (nowe dane gotowe do odczytania)
 - EOS - koniec sekwencji konwersji (grupy regularnej)
- są trzy *watchdogi pseudo-analogowe*
- w przypadku wykrycia *overrun* można sobie skonfigurować czy wolimy porzucić nowy wynik konwersji (który nadpisałby poprzedni) czy też wolimy nadpisanie (patrz bit OVRMOD)
- zmieniły się tryby DMA:
 - *one shot mode* - przetwornik ADC przestaje generować żądania DMA po wyzerowaniu rejestru DMA_CNDTR
 - *circular mode* - przetwornik ADC nie przestaje generować żądań po wyzerowaniu CNDTR (do realizacji trybu kołowego)
- coś się zmieniło w kwestii offsetów (chyba można ustawić offset również dla kanałów regularnych, ale się nie zagłębiałem)
- przybyła funkcja *injected conversions queue of context*, ale (ponownie) się nie wczytywałem bo to jakieś nudne
- pojawiła się nowa funkcja - *auto-delayed conversion* - polega to na tym, że jeżeli nowa konwersja nadpisałaby wynik poprzedniej (*overrun*) to zostaje ona wstrzymana (trygierze są ignorowane), do czasu odczytania poprzedniego wyniku

Nie będziemy się bawić w odkrywanie kanałów regularnych i wstrzykiwanych na nowo, bo mamy ciekawsze rzeczy do roboty :] Toteż szybki przykład na koniec i pojedziemy dalej.

Zadanie domowe 13.12: budujemy prosty układ składający się z trzech, połączonych szeregowo, oporników (kolejno R1, R2, R3). Skrajne nóżki tego dwójnika dołączamy do +3,3V i masy płytki Nucleo. Wartości oporników bez znaczenia (oczywiście w rozsądnych granicach!).

W programie należy uruchomić i skalibrować przetwornik ADC. Następnie skonfigurować ustrojstwo tak, aby wykonało:

- 100 pomiarów napięcia wbudowanego źródła odniesienia (ino migiem :))
- 100 pomiarów napięcia na oporniku R2 (jak najszybciej, równocześnie z pomiarami z pierwszej kropki)
- pomiar temperatury (z wykorzystaniem wbudowanego czujnika), co 1ms

Wyniki dwóch pierwszych kropek mają wylądować w dwóch 100-elementowych tablicach. Ostatnia kropka ma zapełnić tablicę o rozmiarze 10-ciu elementów. W tablicy ma lądować wartość temperatury a nie goły odczyt z ADC. Po zakończeniu pomiarów program ma zapalić diodę, żeby była wiadomo że to już :) Międz zabawy.

Rozwiążanie (nie czytaj póki sam nie rozwiązesz zadania lub przynajmniej nie spróbujesz rozwiązać!): przykład wyszedł nieco bardziej rozbudowany, niżem się spodziewał. Dlatego też trochę go poszatkujemy i omówimy „po kawałku”. Zaczniemy od ustalenia planu operacyjnego. Treść zadania jest oczywiście skonstruowana podstępnie i ma wymusić zastosowanie, i tym samym poznanie, określonych mechanizmów. I tak:

- pomiar napięcia na oporniku R2, którego żadna nóżka nie jest na potencjale masy, aż się prosi o wykorzystanie różnicowego trybu pracy
- pomiar źródła odniesienia jak najszybciej sugeruje, aby użyć konwersji w grupie regularnej bez wyzwalania sprzętowego - taki *free run* znany z AVR
- słówko klucz „równocześnie” przy pomiarze napięcia na R2, implikuje użycie trybu podwójnego (*dual*)
- pomiar temperatury, który ma być niezależnie od pozostałych pomiarów i wyzwalany co 1ms, aż się prosi o użycie grupy wstrzykiwanej

Dwa z wymienionych kanałów (czujnik temperatury i wbudowane źródło odniesienia) to kanały wewnętrzne²²⁴. Nie każdy kanał wewnętrzny może współpracować z dowolnym

²²⁴ wybrane z lenistwa - nie trzeba organizować dodatkowego hardware'u

przetwornikiem. Trzeba to uwzględnić już na wczesnym etapie planowania. Dostępność kanałów wewnętrznych można sprawdzić w kilku miejscach, przykładowo:

- RM, rozdział dotyczący ADC, podrozdział *ADC main features* - przy spisie wewnętrznych kanałów ADC mamy od razu informacje o tym, z jakimi przetwornikami mogą współpracować, przykładowo: *One [channel] from internal temperature sensor (VTS), connected to ADC1*
- na schemacie blokowym w rozdziale *ADC1/2 connectivity* można odszukać interesujące nas źródła wewnętrzne i sprawdzić do którego przetwornika są podłączone
- rozdział opisujący czujnik temperatury (analogicznie w przypadku innych źródeł wewnętrznych): *The temperature sensor is internally connected to the ADC1_IN16 input channel*

Gdzie byśmy nie sprawdzili, wnioski są zawsze takie same:

- czujnik temperatury podłączony jest tylko do ADC1
- źródło wewnętrzne może współpracować z ADC1 i ADC2

W kwestii kanałów mierzących napięcia na R2 sprawa jest prostsza, bo kanałów zewnętrznych jest pod dostatkiem. Ja akurat wybrałem ADC1_CH1 i ADC1_CH2, nóżki odpowiednio PA0 i PA1. Przypominam, że będziemy pracować w trybie różnicowym, czyli musimy wybrać dwa kolejne kanały (np. CH1 i CH2, CH2 i CH3, CH3 i CH4, ...).

Summa summarum zaplanowałem sobie, że:

- ADC1 będzie mierzył (w grupie regularnej) napięcie na R2 oraz temperaturę w grupie strzykawowej
- ADC2 będzie mierzył (w grupie regularnej) napięcie wewnętrznego źródła odniesienia

Za wyzwalanie grupy strzykawowej, oczywiście, odpowiedzialny będzie jakiś timer. No ale to chyba nie budzi wątpliwości :) Który timer? Oto jest pytanie. Udajemy się z nim do rozdziału *Conversion on external trigger and trigger polarity (EXTSEL, EXTE, JEXTSEL, JEXTEN)* RMa. Znajdują się tam dwie tabelki, w których zawarto wszystkie możliwe źródła wyzwalania dla konwersji regularnych i wstrzykiwanych. Ja wybrałem JEXT14 - TIM6_TRGO. Czemu? A czemu nie?

Jedziemy dalej - co z wynikami? Oczywiście użyjemy naszego cichego pomocnika. Wyniki konwersji grup regularnych mają lądować w dwóch niezależnych tablicach - w związku z tym, użyjemy dwóch kanałów DMA. Każdy będzie transportował efekty pracy jednego przetwornika do osobnej tablicy. W trybie *dual* przetwornika ADC, jest możliwość użycia jednego kanału DMA do przesyłania wyników z obu przetworników, z wspólnego rejestru danych. Jednak takie rozwiązanie uniemożliwia rozdzielenie wyników z obu przetworników do dwóch osobnych tablic. Wyniki zostałyby umieszczone naprzemiennie w jednej tablicy (patrz zadanie 13.5). A tego nie chcemy :]

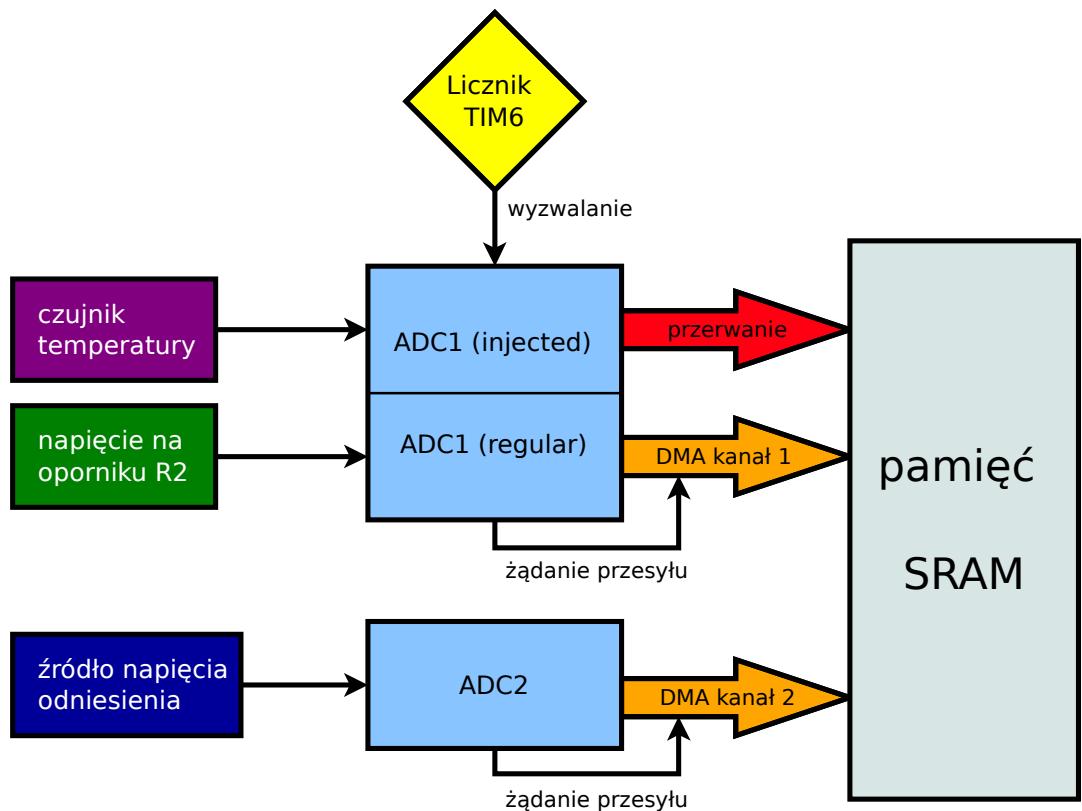
Czas dobrać kanały DMA. W tym celu udajemy się do naszej ulubionej pozycji pe-de-eficznnej (RM), do rozdziału *DMA request mapping*. Na schemacie bądź w tabelce, jak kto woli, sprawdzamy które kanały DMA mogą być wyzwalane przez przetworniki ADC. W tabelce wygodniej :] Wynik poszukiwań to:

- ADC1 może żądać przesyłów kanałem pierwszym
- ADC2 może żądać przesyłów kanałem drugim lub czwartym

Dla porządku wybieramy kanały 1 i 2.

Grupa iniekcyjna nie może generować żądań DMA. Zresztą tutaj każdy kanał ma osobny rejestr danych, więc nie ma takiego popłochu z odbieraniem wyników. Dodatkowo chcemy od razu przeliczać wynik konwersji na wartość temperatury (w stopniach), a tego nam DMA nie załatwia. To musimy zrobić sami. Z tego względu, obróbkę wyników z grupy wstrzykiwanej, zrobimy w przerwaniu odpalanym po zakończeniu konwersji.

Podsumujmy dotychczasowe ustalenia w formie artystycznego bohomazu (tak wiem - znowu rysunek i to kolorowy - ostatnio szaleję):



Rys. 13.6 Plan operacyjny do zadania 13.12

No to już prawie wszystko wiemy. Przejdźmy do kodu. Na pierwszy ogień weźmiemy sobie funkcję „główną”.

Przykładowe rozwiążanie - funkcja główna (F334, wejścia analogowe PA0, PA1):

```
1. static volatile bool ch1_tc = false;
2. static volatile bool ch2_tc = false;
3. static volatile bool ch3_tc = false;
4.
5. int main(void){
6.
7.     RCC->AHBENR = RCC_AHBENR_GPIOAEN | RCC_AHBENR_DMA1EN | RCC_AHBENR_ADC12EN;
8.     RCC->APB1ENR = RCC_APB1ENR_TIM6EN;
9.
10.    gpio_pin_cfg(GPIOA, PA5, gpio_mode_out_PP_LS);
11.    gpio_pin_cfg(GPIOA, PA0, gpio_mode_analog);
12.    gpio_pin_cfg(GPIOA, PA1, gpio_mode_analog);
13.
14.    TIM6->PSC = 0;
15.    TIM6->ARR = 79;
16.    TIM6->EGR = TIM_EGR_UG;
17.
18.    ADC1_2_COMMON->CCR = ADC12_CCR_CKMODE_0 | ADC12_CCR_CKMODE_1 | ADC12_CCR_TSEN |
19.        ADC12_CCR_VREFEN | 0b110;
20.
21.    adc1_calib();
22.    adc2_calib();
23.
24.    ADC1->DIFSEL = ADC_DIFSEL_DIFSEL_0;
25.
26.    adc1_start();
27.    adc2_start();
28.
29.    adc_cfg();
30.    dma_cfg();
31.
32.    ADC1->CR |= ADC_CR_ADSTART | ADC_CR_JADSTART;
33.
34.    RCC->APB1RSTR = RCC_APB1RSTR_TIM6RST;
35.    RCC->APB1RSTR = 0;
36.
37.    TIM6->PSC = 40 - 1;
38.    TIM6->ARR = 200 - 1;
39.    TIM6->EGR = TIM_EGR_UG;
40.    TIM6->CR2 = TIM_CR2_MMS_1;
41.    TIM6->CR1 = TIM_CR1_CEN;
42.
43.    NVIC_EnableIRQ(ADC1_2_IRQn);
44.    NVIC_EnableIRQ(DMA1_Channel1_IRQn);
45.    NVIC_EnableIRQ(DMA1_Channel2_IRQn);
46.
47.    while (!(ch1_tc && ch2_tc && ch3_tc));
48.
49.    GPIOA->BSRR = GPIO_BSRR_BS_5;
50.    __BKPT();
51.
52. }
```

Jedziemy:

7, 8) włączenie zegarów dla wykorzystywanych peryferiali: portu A, kontrolera DMA, przetworników ADC i licznika

10 - 12) konfiguracja wyprowadzeń: PA0 i PA1 to wejścia analogowe (będą pracować w trybie różnicowym), PA5 to błyskotka (ledka)

14 - 16) konfiguracja licznika TIM6, ale jeszcze nie do wyzwalania ADC! W trakcie uruchamiania (nieco upierdliwych) przetworników ADC w F334, kilka razy będziemy potrzebowali krótkiego opóźnienia (rzędu 10μs). Funkcja opóźniająca będzie oparta właśnie na tym liczniku. Wróćmy do tego później.

18, 19) tak jak już wcześniej wspominałem (a przynajmniej tak mi się wydaje) ADC może być taktowane z kilku źródeł. Wszystko wyjaśni się po lekturze rozdziału 17 - na razie przyjmuj na wiarę. Przed dalszą zabawą musimyłączyć jedno ze źródeł zegara dla przetworników. Pole odpowiedzialne za wybór źródła (CKMODE) znajduje się w rejestrze kontrolnym wspólnym dla obu przetworników (CCR - Common Control Register). Z niewyjaśnionych dla mnie przyczyn, zamiast nazwać blok zawierający wspólne rejesty obu przetworników ładną, krótką i wszystko mówiącą nazwą: ADC12; komuś urodził się szalony pomysł nazwania go ADC1_2_COMMON. Dobrze że nazwy bitów nie zawierają już tego członu.

Tak czy inaczej wybieram tryb z zegarem HCLK/4 (cierpliwości, wyjaśni się). Warto w tym momencie zwrócić uwagę na małe ograniczenie dotyczące zegara. Cofamy się do rozdziału *Clocks* w opisie ADC i szukamy punktu: *Clock ratio constraint between ADC clock and AHB clock*. W telegraficznym skrócie: generalnie ograniczeń nie ma... no chyba, że używamy kanałów wstrzykiwanych. W takim przypadku częstotliwość zegara ADC musi być odpowiednio mniejsza od częstotliwości szyny AHB (wspominałem już, że wyjaśni się w rozdziale 17?). Wymagana różnica częstotliwości jest uzależniona od wybranej rozdzielczości przetwornika. Np. dla 12 i 10-bitowej rozdzielczości f_{ADC} musi być przynajmniej 4x mniejsze od f_{AHB} . Trzeba o tym pamiętać i tyle.

Skoro już grzebiemy w rejestrze CCR to przy okazjiłączam źródło referencyjne (bit VREFEN), czujnik temperatury (TSEN) i wybieram tryb *dual (regular simultaneous mode)*. Jak ktoś ma wątpliwości dotyczące wyboru trybu to śmiało - RTFM :)

21, 22) przetworniki ADC w F334 są nieco upierdliwe w obyciu. Samo uruchomienie i skalibrowanie zajmuje więcej (linijk kodu) niż całe proste przykłady dla F103. Kalibrację przetworników wrzuciłem do osobnych funkcji, żeby nie przeszkadzała teraz. Zajmiemy się nią później.

24) pomału przymierzamy się do włączenia przetwornika, ale zanim to zrobimy konfigurujemy jeszcze tryb różnicowy. Tu ważna uwaga! Upierdliwość ADC w F334 objawia się m.in. tym, że istnieje sporo ograniczeń dotyczących modyfikacji bitów sterujących. Opisuje to dokładnie rozdział *Constraints when writing the ADC control bits²²⁵*. Niby wszystko po zastanowieniu jest logiczne, ale i tak powoduje jakiś niesmak. Jedno z ograniczeń dotyczy właśnie rejestru DIFSEL - modyfikować go należy tylko, gdy przetwornik jest wyłączony (ADEN = 0). Toteż właśnie modyfikujemy.

Chwili zastanowienia może wymagać nazwa bitu: ADC_DIFSEL_DIFSEL_0. I teraz zagwozdka - zero na końcu nazwy oznacza:

²²⁵ no i dodatkowo są uwagi i dopiski przy opisie bitów/rejestrów podlegających obostrzeniom

- numer bitu w rejestrze DIFSEL
- numer bitu w polu DIFSEL
- numer kanału dla którego włączany jest tryb różnicowy

Poprawna odpowiedź to druga kropka. To jest zerowy bit pola DIFSEL znajdującego się w rejestrze DIFSEL. W samym rejestrze jest to bit numer 1 (przy numeracji od zera). Jego ustawienie powoduje włączenie trybu różnicowego dla kanału pierwszego (ADC_IN1). Czepiam się (być może przesadzam), bo wydaje mi się to trochę średnio pomyślane. Mogli nazwać poszczególne bity rejestru DIFSEL tak, aby nazwa jednoznacznie wskazywała kanał ADC, np.: ADC_DIFSEL_CH1. No ale ja się lubię czepiać.

26, 27) włączenie przetworników. Dla przejrzystości wyniesione poza *main*.

29, 30) konfiguracja przetworników ADC i kanałów DMA, w osobnych funkcjach z komentarzem jw.

32) wreszcie wyczekane wystartowanie konwersji. Uruchamiam tylko konwersje przetwornika nadzawanego (ADC1). Slave (ADC2) uruchomi się, że tak powiem, automatycznie (bo mu *master* każe). Dodatkowo zwracam uwagę na operator sumo-przypisania (`|=`), żeby nie nadpisać sobie konfiguracji przetworników z poprzednich linijek.

34, 35) kilka linijek wyżej pisałem o tym, że w procesie rozburzania przetworników będziemy wykorzystywali prostą funkcję opóźniającą opartą o TIM6. Wtedy też, jakoś go tam sobie skonfigurowaliśmy. Teraz jednak chcemy już ustawić TIM6 tak, aby wzywał ADC. W tym celu musimy pozbyć się poprzedniej konfiguracji.

Można oczywiście ręcznie skasować ustawione wcześniej bity, nadpisać wartości rejestrów itd. Ale jeżeli np. kiedyś zmienimy konfigurację TIM6 z linijk 14 - 16 (tą dla opóźnień) - to jest bardzo duża szansa na to, że zapomnimy o cofnięciu tych zmian przy „rekonfiguracji” (do wyzwalania ADC). Dlatego też lepszą opcją wydaje się zresetowanie bloku licznika. No i właśnie to się dzieje w tych linijkach, przywracane są domyślne wartości rejestrów, jak po resecie.Więcej na ten temat w wyczekiwany rozdziale o systemie zegarowym (rozdział 17).

37 - 41) nowa konfiguracja licznika. Przypominam, że ADC ma być wyzwalany co 1 ms sygnałem TRGO licznika TIM6.

43 - 45) włączamy w NVICu trzy przerwania:

- przerwanie od przetwornika ADC - to będzie przerwanie zakończenie konwersji grupy wstrzykiwanej, w nim będziemy przeliczać wynik pomiaru na temperaturę i zapisywać w tablicy

- dwa przerwania od zakończenia konwersji kanałów DMA

Przerwania od końca konwersji DMA posłużą m.in. do ustawiania flag. Wykazując się niebywałą kreatywnością nazwałem je *ch1_tc* i *ch2_tc* (od *transfer complete*) - patrz linijki 1 - 3. Gdy wszystkie trzy pomiary się zakończą (R2, temperatura, napięcie odniesienia) wszystkie flagi zostaną ustawione i spełniony będzie warunek z linii 47. Program zapali wtedy diodę i za-brejk-pointuje się. A my sobie odczytamy efekty jego pracy debuggerem :]

No to główną część programu mamy za sobą. Przejdźmy do uruchomienia i kalibracji przetworników ADC.

Przykładowe rozwiązanie - kalibracja i uruchomienie ADC (F334, wejścia analogowe PA0, PA1):

```

1. void adc1_calib(void){
2.   ADC1->CR = 0;
3.   ADC1->CR = ADC_CR_ADVREGEN_0;
4.   delay_10us();
5.
6.   ADC1->CR = ADC_CR_ADVREGEN_0 | ADC_CR_ADCAL;
7.   while (ADC1->CR & ADC_CR_ADCAL);
8.   delay_10us();
9.
10.  ADC1->CR = ADC_CR_ADVREGEN_0 | ADC_CR_ADCALDIF | ADC_CR_ADCAL;
11.  while (ADC1->CR & ADC_CR_ADCAL);
12.  delay_10us();
13. }
14.
15. void adc1_start(void){
16.   ADC1->CR = ADC_CR_ADVREGEN_0 | ADC_CR_ADEN;
17.   while (~ADC1->ISR & ADC_ISR_ADRD);
18. }
```

Funkcje są analogiczne dla obu przetworników, wklejam tylko wersje dla ADC1.

Pierwszy krok to wybudzenie jakiegoś tajemniczego, wewnętrznego regulatora napięcia ADC. Składają się nań trzy działania:

- skasowanie bitu ADVREGEN_1, który jest domyślnie ustawiony i odpowiada za jakieś tam uśpienie regulatora (to jedyny domyślnie ustawiony bit w ADCx_CR, kasuję go zapisując zero do rejestru)
- ustawienie bitu ADVREGEN_0, który odpowiada za włączenie regulatora
- oczekanie chwilki na rozbudzenie regulatora (10μs w najgorszym wypadku)

Co ważne, kroków 1 i 2 nie można wykonać jednocześnie (tzn. w jednej modyfikacji rejestru zmienić obu bitów). Dokumentacja wymaga, aby najpierw zmienić stan pola ADVREGEN z po-resetowej wartości 0b10 na 0b00 (stan przejściowy) i dopiero z 0b00 na 0b01. Niech im będzie.

Drugi krok to kalibracja przetwornika. Kalibrację uruchamia ustawienie bitu ADCAL. Bit jest sprzętowo kasowany po zakończeniu kalibracji. Kalibrację przeprowadzam dwukrotnie - raz dla trybu *single-ended* i oddzielnie dla trybu *differential*. Za wybór kalibrowanego trybu odpowiada bit ADCALDIF. Oczywiście jeśli nie planujemy wykorzystania obu trybów to nie ma przymusu powtarzania kalibracji. Uzyskane wartości kalibracyjne, w razie potrzeby, można sobie odczytać z rejestru ADCx_CALFACT (lub można je doń wpisać, jeśli je znamy).

W dokumentacji znajduje się informacja, że po zakończeniu kalibracji (sprzętowym skasowaniu bitu ADCAL) muszą upływać minimum 4 cykle zegara ADC zanim przetwornik zostanie uruchomiony (ADEN = 1). Nie ma żadnej wzmianki o wymaganym opóźnieniu między kalibracjami. W testach bojowych zauważylem jednak, że układ kilka razy się zawiesił w oczekiwaniu na skasowanie ADCAL gdy druga kalibracja była uruchamiana bezpośrednio po zakończeniu pierwszej. Dodanie opóźnienia rozwiązało problem.

Druga funkcja odpowiada za (wreszcie) uruchomienie przetwornika ADC. Tzn. nie uruchomienie w sensie, że od teraz zacznie mierzyć... Nie uruchomienie pomiarów, tylko włączenie przetwornika... czy coś. RM nazywa to „*ready to operate*”. No nie ważne. Generalnie włączamy przetwornik i czekamy aż się uruchomi. A dokładniej czekamy na flagę gotowości - ADRDY²²⁶. I tyle w temacie.

No to idąc za ciosem, weźmy teraz na warsztat konfigurację przetworników ADC i kanałów DMA, i psim swędem dorzucimy jeszcze funkcję opóźniającą:

226 tzn. w RMie bit nazywa się ADRDY, zaś w nagłówku zjadło się komuś końcowe Y...

Przykładowe rozwiążanie - konfiguracja ADC, DMA, funkcja opóźniająca (F334, wejścia analogowe PA0, PA1):

```
1. void adc_cfg(void){
2.     ADC1->SMPR1 = 7 << 3;
3.     ADC1->SMPR2 = 7 << 18;
4.     ADC1->SQR1 = 1 << 6;
5.     ADC1->JSQR = 16 << 8 | ADC_JSQR_JEXTEN_0 | 14 << 2 /* TIM6_TRGO */;
6.     ADC1->CFGGR = ADC_CFGGR_CONT | ADC_CFGGR_DMAEN;
7.     ADC1->IER = ADC_IER_JE0C;
8.
9.     ADC2->SMPR2 = 7 << 24;
10.    ADC2->SQR1 = 18 << 6;
11.    ADC2->CFGGR = ADC_CFGGR_DMAEN;
12. }
13.
14. void dma_cfg(void){
15.     DMA1_Channel1->CPAR = (uint32_t) &ADC1->DR;
16.     DMA1_Channel1->CMAR = (uint32_t) adc_r2;
17.     DMA1_Channel1->CNDTR = 100;
18.     DMA1_Channel1->CCR = DMA_CCR_PSIZE_0 | DMA_CCR_MSIZE_0 | DMA_CCR_MINC | DMA_CCR_TCIE |
19.         DMA_CCR_EN;
20.
21.     DMA1_Channel2->CPAR = (uint32_t) &ADC2->DR;
22.     DMA1_Channel2->CMAR = (uint32_t) adc_vrefint;
23.     DMA1_Channel2->CNDTR = 100;
24.     DMA1_Channel2->CCR = DMA_CCR_PSIZE_0 | DMA_CCR_MSIZE_0 | DMA_CCR_MINC | DMA_CCR_TCIE |
25.         DMA_CCR_EN;
26. }
27.
28. void delay_10us(void){
29.     TIM6->CR1 = TIM_CR1_OPM | TIM_CR1_CEN;
30.     while (~TIM6->SR & TIM_SR UIF);
31.     TIM6->SR = 0;
32. }
```

2, 3, 9) nie wnikając z rezystancje źródeł sygnałów i inne takie pierdoły, dla wszystkich kanałów ustawiam maksymalny czas próbkowania - kto bogatemu zabroni

4, 5, 10) wybór kanałów do konwersji w ramach grupy regularnej i wstrzykiwanej oraz wybór źródła wyzwalania dla konwersji wstrzykiwanych (TIM6_TRGO)

6, 11) włączenie trybu ciągłego i generowania żądań DMA. W przypadku przetwornika podzielnego (*slave, adc2*) nie ma konieczności włączania trybu ciągłego. W trybach *dual* przetwornik *adc2* „dziedziczy” część ustawień z mastera. Dziedziczeniu podlegają następujące bity z rejestru CFGGR: CONT, AUTDLY, DISCEN, DISCNUM, JDISCEN, JQM, JAUTO oraz EXTEN, EXTSEL, JEXTEN, JEXTSEL (dotyczy tylko trybów *dual*, w których oba przetworniki są wyzwalane sprzętowo tym samym sygnałem).

7) włączam generowanie przerwań po konwersji wstrzykiwanej

14 - 26) konfiguracja DMA chyba nie budzi wątpliwości. Oba kanały skonfigurowane są tak samo. Przesyłają dane 16 bitowe z rejestrów danych przetworników ADC do buforów w pamięci SRAM mikrokontrolera. Adres po stronie pamięci podlega inkrementacji. Wykonywane jest 100 przesyłów, po czym odpala się przerwanie zakończenia transferu (TC - *transfer complete*).

28 - 32) funkcja opóźniająca to już w ogóle dla nas banał. Timer startuje w trybie *one pulse*, czyli zatrzyma się po przepełnieniu. Pętla *while* czeka na to przepełnienie (na flagę UIF). Konfiguracja

„czasowa” licznika została dobrana tak, aby uzyskać opóźnienie ~10μs (patrz linijki 14 - 16 w kodzie funkcji głównej).

Żeby nie było wątpliwości - uzyskane opóźnienie nie będzie wynosiło równe 10μs. Do 10μs wynikających z oczekiwania na przepelenie timera, dołożą się: wywołanie funkcji opóźniającej, jej prolog, uruchomienie licznika, skasowanie flagi, epilog funkcji, powrót. Z tego względu uzyskamy (sporo) dłuższe opóźnienie, ale akurat tutaj zależy nam na opóźnieniu minimum 10μs (żeby ADC zdążyło się wybudzić). Dłuższe opóźnienie w niczym nie przeszkadza.

No to co nam jeszcze zostało? Ostatni kawałek pizzy, procedury obsługi przerwań :) No to siup.

Przykładowe rozwiążanie - ISRy (F334, wejścia analogowe PA0, PA1):

```
1. static volatile uint16_t adc_vrefint[100];
2. static volatile uint16_t adc_r2[100];
3.
4. static volatile float adc_temp1[10];
5. static volatile uint32_t adc_temp2[10];
6.
7. void DMA1_Channel1_IRQHandler(void){
8.
9.     if ( DMA1->ISR & DMA_ISR_TCIF1 ) {
10.         DMA1->IFCR = DMA_IFCR_CTCIF1;
11.         DMA1_Channel1->CCR = 0;
12.         ch1_tc = true;
13.     }
14.
15.     if ( DMA1->ISR & DMA_ISR_TCIF2 ) {
16.         DMA1->IFCR = DMA_IFCR_CTCIF2;
17.         DMA1_Channel2->CCR = 0;
18.         ch2_tc = true;
19.     }
20.
21.     if ( ch1_tc && ch2_tc ) {
22.         ADC1->CR |= ADC_CR_ADSTP;
23.     }
24. }
25.
26. void DMA1_Channel2_IRQHandler(void) __attribute__((alias("DMA1_Channel1_IRQHandler")));
27.
28. void ADC1_2_IRQHandler(void){
29.
30.     const uint16_t ts_call1 = *(const uint16_t *) 0x1fffff7b8;
31.     const uint16_t ts_call2 = *(const uint16_t *) 0x1fffff7c2;
32.     static uint32_t i;
33.
34.     if ( ADC1->ISR & ADC_ISR_JEOC ) {
35.         ADC1->ISR = ADC_ISR_JEOC;
36.
37.         if ( i < 10 ) {
38.             uint32_t voltage = ADC1->JDR1 * 3300 / 4095;
39.             adc_temp1[i] = 80.0f*(ADC1->JDR1 - ts_call1) / (ts_call2 - ts_call1) + 30.0f;
40.             adc_temp2[i] = (1430 - voltage)*10 / 43 + 25;
41.             i++;
42.         } else {
43.             ADC1->CR |= ADC_CR_JADSTP;
44.             TIM6->CR1 = 0;
45.             ch3_tc = true;
46.         }
47.     }
48. }
```

7) procedura obsługi przerwania od końca transferu pierwszego kanału DMA. Nic odkrywczego: sprawdzam flagę aby zweryfikować źródło przerwania (linijka 9), kasuję flagę (10), wyłączam kanał DMA (11), ustawiam sobie flagę końca konwersji (*ch1_tc*). Wyłączenie kanału nie jest specjalnie potrzebne bo nie pracuje on w trybie kołowym, więc i tak przestanie reagować na żądania z racji wyzerowania rejestru CNDTR. Ale... czemu by nie? Warunkiem z linii 15 - 19 zajmiemy się za momencik.

Gdy oba kanały DMA zakończą pracę, czyli oba bufory na dane z konwersji regularnych będą zapełnione, możemy wyłączyć całkowicie konwersje grup regularnych. Tym zajmuje się warunek z linii 21.

26) tutaj „jest” procedura obsługi przerwania dla drugiego kanału DMA. Tzn. nie do końca jest, bo dla urozmaicenia przykładu ten wektor nie ma osobnego ISRa. Z pomocą przyszedł atrybut funkcji *alias* (niezmiennie kompilator *gcc*). Efekt działania powyższego jest taki, że dwa wpisy w tablicy wektorów (dla pierwszego i drugiego kanału DMA) mają tą samą wartość - wskazującą na adres funkcji *DMA1_Channel1_IRQHandler()*. Czyli w przypadku wystąpienia któregośkolwiek z tych przerwań, procesor ląduje w tej samej funkcji ISR. Podobny efekt można uzyskać przy AVRkach za pomocą „parametru” makra ISR o nazwie „ALIASOF” czy jakoś podobnie.

Jak ktoś ma awersję do atrybutów czy coś w tym stylu, to może w ISR dla drugiego kanału umieścić zwyczajne wywołanie funkcji *DMA1_Channel1_IRQHandler()*. Działanie programu będzie takie samo, choć rozwiążanie nieco mniej finezyjne :] Jedziemy kontynuować dalej.

28) procedura obsługi przerwania wywoływanego po zakończeniu konwersji z grupy wstrzykiwanej (pomiar temperatury). Tu jest trochę magii. Generalnie idea przerwania jest taka:

- sprawdzamy flagę źródła przerwania (linijka 34)
- kasujemy flagę (linijka 35)
- jeżeli nie zapełniliśmy jeszcze całej, 10-cio elementowej, tablicy na wyniki (linijka 37) to przeliczamy to co wypluło ADC na temperaturę (38 - 40)
- jeżeli tablica jest pełna to zatrzymujemy konwersje wstrzykiwane (linijka 43)
- zatrzymujemy timer wyzwalający konwersje, bo już jest niepotrzebny (linijka 44)
- ustawiamy flagę końca pomiarów (linijka 45)

Zatrzymajmy się na trochę przy tym przeliczaniu wyniku ADC na temperaturę. Generalnie nic się nie zmieniło względem F103 i F429, to znaczy temperatura jest określona wzorkiem:

$$T = \frac{V_{25} - V_{sense}}{Avg_slope} + 25 \quad [{}^{\circ}\text{C}]$$

gdzie (źródło datasheet):

- V_{25} - napięcie czujnika przy 25°C (1,34...1,52V, typowo 1,43V)
- V_{sense} - zmierzone przez ADC napięcie czujnika
- Avg_slope - współczynnik nachylenia charakterystyki czujnika ($4,0\ldots4,6 \text{ mV/}^{\circ}\text{C}$, typowo $4,3\text{mV/}^{\circ}\text{C}$)

Ponadto, w dalszym ciągu, producent ostrzega przed znacznym rozrzutem parametrów czujnika. Dokładniej sprawa dotyczy offsetu. Kilka kostek, tego samego modelu mikrokontrolera, może w tej samej temperaturze wskazywać temperatury różniące się o 45°C !

W celu umożliwienia dokonywania jako tako wiarygodnych pomiarów bez konieczności kalibracji każdej kostki przez „użytkownika”, ST zapisuje w każdym mikrokontrolerze dane kalibracyjne czujnika temperatury²²⁷. Zapisywane są one trwale, na etapie produkcji mikrokontrolera. Są to dwie wartości (16 bitowe), zapisane pod konkretnymi adresami w pamięci flash, uzyskane poprzez pomiar (przetwornikiem ADC) czujnika temperatury w dwóch, określonych temperaturach. Zwracam szczególną uwagę na to, że to nie są wartości przeliczone na napięcie tylko „gołe” wyniki wyplute przez ADC. Konkrety znajdziemy u cioci dokumentacji, w naszym ulubionym datasheetcie (rozdział *Electrical characteristics* → *Operating conditions* → *Temperature sensor (TS) characteristics* → tabela *Temperature sensor (TS) calibration values*):

Tabela 13.6. Wartości kalibracyjne czujnika temperatury i wbudowanego źródła napięcia odniesienia (źródło *datasheet*)

Wartość kalibracyjna	Opis	Adres pamięci
TS_CAL1	pomiar w 30°C , VDDA = 3,3V	0xFFFF F7B8
TS_CAL2	pomiar w 110°C , VDDA = 3,3V	0xFFFF F7C2
Vrefint_cal ²²⁸	pomiar w 30°C , VDDA = 3,3V	0xFFFF F7BA

Czyli pod tymi dwoma adresami (z pierwszych dwóch wierszy tabeli) znajdują się dwa wyniki konwersji dla temperatur 30°C i 110°C . Niestety nie dogrzebałem się do informacji na temat dokładności tych danych kalibracyjnych. Ale i specjalnie nie szukałem ;)

227 notabene w F429 też tak było tylko jakoś to przegapiłem...

228 wartość kalibracyjna wbudowanego źródła napięcia odniesienia

Jak wykorzystać te dane kalibracyjne? Sprawa jest jak zawsze prosta, bo oparta o matematykę. Temperatura jest liniową funkcją wyniku konwersji ADC, a przynajmniej tak zakładamy. Znamy dwa wyniki konwersji w dwóch konkretnych temperaturach. Posługując się językiem matematyki „gimnazjalnej” musimy wyznaczyć równanie prostej przechodzącej przez te dwa punkty. Potem wystarczy podstawić do niego dowolny wynik z ADC i ta dam :). Przekształcenia matematyczne zrób sobie sam, wynik przedstawia się następująco:

$$\vartheta = \frac{80}{\text{TS_CAL2} - \text{TS_CAL1}} \cdot (\text{adc} - \text{TS_CAL1}) + 30 \quad [{}^{\circ}\text{C}]$$

No i ten wzorek jest bardzo sympatyczny. Nie dość, że nie trzeba przeliczać wyniku ADC na napięcie to jeszcze mamy mega-hiper dokładność zapewnioną przez unikatowe dane kalibracyjne... o których dokładności niewiele wiadomo. Mniejsza. Tak czy siak działa. Implementacja powyższego wzorku znajduje się w linii 39 programu. Wartości temperatury zapisywane są w tablicy *adc_temp1*. Z lenistwa zgrzeszyłem i jest to tablica floatów. Dla porównania, dorzuciłem jeszcze „standardowe” obliczanie temperatury, w oparciu o wcześniejszy wzorek i wartości z datasheeta. Wyniki lądują w tablicy *adc_temp2*²²⁹.

Tym sposobem mamy omówiony cały program. Hip hip! Czas na wyniki. Bez przedłużania i budowania napięcia jak H. Urbański w Milionerach, wyniki działania programu przedstawiają się następująco:

- wartości kalibracyjne (jakby kogoś interesowało):
 - TS_CAL1 = 1763
 - TS_CAL2 = 1323
 - Vrefint_cal = 1535
- pomiar napięcia odniesienia (goły wynik z ADC, wartość średnia z pomiarów): 1518
- pomiar napięcia na R2 (goły wynik z ADC, wartość średnia z pomiarów): 2729
- temperatura „*adc_temp1*” (wartość średnia): 31,45 °C
- temperatura „*adc_temp2*” (wartość średnia): 28 °C

No i co tu komentować? Wartości kalibracyjne jakie są, takie są. Wynik z pomiaru napięcia odniesienia wygląda w miarę ok (1518 adc = 1,22V). Podobnie różnicowy pomiar napięcia na R2, do budowy „układu” doświadczalnego wykorzystałem trzy jednakowe rezystory. Stąd napięcie

²²⁹ wiwat kreatywność w wymyślaniu nazw dla obiektów w programie :)

zmierzone na rezystorze R2 powinno wynosić 1/3 napięcia zasilania (2729 adc = 1,10V). Temperatury też wyglądają sensownie. Przypominam, że to jest bardziej temperatura „krzemu” w mikrokontrolerze niż temperatura otaczającego eteru - nie mam aktualnie ~30 stopni w pokoju :)

Co warto zapamiętać z tego rozdziału?

- niby wszystko działa tak jak w F429, ale jest kilka różnic
- ADC w F334 jest o wiele szybsze niż w F103/F429 - do 5,2Ms/s dla kanałów „szybkich”
- rozruch przetworników jest nieco upierdliwy
- pomiary różnicowe
- dane kalibracyjne

13.9. Końcowe uwagi

ST wyprodukowało parę not aplikacyjnych poświęconych przetwornikom ADC. O kilku z nich wspomniałem już w poprzednich podrozdziałach. Tutaj postanowiłem zebrać wszystkie, aby były w jednym miejscu. Przyjemniej lektury.

- AN2668 *Improving STM32F1x and STM32L1x ADC resolution by oversampling*
- AN3116 *STM32TM's ADC modes and their applications*
- AN4073 *How to improve ADC accuracy when using STM32F2xx and STM32F4xx microcontrollers*
- AN2558 *STM32F10xxx ADC application examples*
- AN2834 *How to get the best ADC accuracy in STM32Fx Series and STM32L1 Series devices*

Dodatkowo chciałbym jeszcze zwrócić uwagę na kilka zapisów z errat, dotyczących przetworników ADC:

- F103: *Voltage glitch on ADC input 0*
- F334: *DMA Overrun in dual interleaved mode with single DMA channel*
- F334: *Sampling time shortened in JAUTO autodelayed mode*
- F334: *Load multiple not supported by ADC interface*
- F334: *Possible voltage drop caused by a transitory phase when the ADC is switching from a regular channel to an injected channel Rank 1*

- F429: *Internal noise impacting the ADC accuracy*
- F429: *ADC sequencer modification during conversion*

Co warto zapamiętać z tego rozdziału?

- nic nie zapamiętywać tylko czytać noty i ćwiczyć!

14. PRZETWORNIK DAC („*OMNE IGNOTUM PRO MAGNIFICO*”²³⁰)

14.1. Wstęp (parametry i tryby pracy)

Co to jest ten DAC? DAC (*digital to analog converter*) to przetwornik cyfrowo analogowy (C/A), czyli układ który zamienia sygnał cyfrowy na analogowy. Taka odwrotność przetwornika analogowo cyfrowego (ADC). W programie zadajemy wartość napięcia (sygnał cyfrowy), a na wyjściu DACa pojawia się to napięcie (sygnał analogowy). Zależność między wartością podaną w programie a napięciem wyjściowym jest z grubsza liniowa i określona wzorkiem wynikającym z proporcji, analogicznym jak przy ADC:

$$V_{\text{DAC out}} = \frac{DOR}{2^N - 1} \cdot V_{\text{ref}}$$

gdzie:

- $V_{\text{DAC out}}$ - napięcie wyjściowe z przetwornika C/A [V]
- DOR - wartość rejestru danych układu DAC (*data output register*) [-]
- N - rozdzielczość przetwornika (12 bitów) [-]
- V_{ref} - napięcie odniesienia (DAC i ADC korzystają z tego samego źródła zasilania części analogowej i źródła napięcia odniesienia) [V]

Przetwornik ma ograniczoną, skończoną rozdzielczość. Przetwornik w STM32 ma 12 bitów. Tzn. że na jego wyjściu może pojawić się tylko $2^{12} = 4096$ różnych wartości napięć z zakresu od około V_{SSA} do mniej więcej $V_{\text{REF+}}$. Zakładając, że dodatnie napięcie odniesienia będzie wynosiło 3,3V, to zmiana zadanej napięcia (wartości wpisanej do rejestru układu peryferyjnego w programie) o ± 1 powoduje zmianę wyjściowej wartości analogowej o: $\pm 3,3V / 4096 = \pm 0,81mV$.

Mikrokontrolery STM32 F103 i F429 posiadają po dwa przetworniki DAC. Każdy przetwornik ma tylko jeden kanał wyjściowy. W obu mikrokontrolerach:

- DAC_OUT1 to nóżka PA4
- DAC_OUT2 to nóżka PA5

Pan indywidualista (F334) też ma dwa przetworniki, ale pierwszy z nich ma dwa kanały wyjściowe:

- DAC1_OUT1 - nóżka PA4

²³⁰ „*Wszystko, co nieznane, wydaje się wspaniałe.*”

- DAC1_OUT2 - nóżka PA5
- DAC2_OUT1 - nóżka PA6

Po włączeniu przetwornika DAC, związanego z nim wyprowadzenie mikrokontrolera, jest z automatu łączone z przetwornikiem. Zaleca się jednak wcześniejsze ustawienie nóżki w konfiguracji analogowej. Zapobiega to występowaniu jakichś tam pasożytniczych prądów upływu.

Wyjścia przetworników DAC mają dosyć dużą impedancję. Z tego względu nie nadają się one do bezpośredniego sterowania odbiornikami sygnału analogowego. W razie potrzeby należy stosować zewnętrzne układy wzmacniające sygnał. Sytuację odrobinę poprawia wbudowany w mikrokontroler układ buforowania sygnału wyjściowego²³¹ (można go włączyć i wyłączyć programowo). Włączenie buforowania pogarsza, niestety, właściwości dynamiczne wyjścia.

Tabela 14.1 Najważniejsze dane elektryczne przetworników DAC (F103 i 429)

wielkość	wartość	opis
R _{LOAD min}	5kΩ	minimalna wartość rezystancji obciążającej wyjście DAC przy włączonym buforowaniu ²³²
	1,5MΩ	minimalna wartość rezystancji obciążającej wyjście DAC przy wyłączonym buforze ²³²
C _{LOAD max}	50pF	maksymalna wartość pojemności obciążającej wyjścia DAC przy włączonym buforze
Z _{OUT max}	15kΩ	maksymalna wartość impedancji wyjścia DAC przy wyłączonym buforze
V _{OUT min}	0,5mV	minimalna wartość napięcia wyjściowego przy wyłączonym buforze
	200mV	minimalna wartość napięcia wyjściowego przy włączonym buforze
V _{OUT max}	V _{ref+} - 1LSB	maksymalna wartość napięcia wyjściowego przy wyłączonym buforze
	V _{DDA} - 0,2V	maksymalna wartość napięcia wyjściowego przy włączonym buforze
t _{settling max}	F103: 4μs F334: 4μs F429: 6μs	maks. czas ustalania się napięcia po wyjściu po "diametralnej" zmianie
t _{wakeup max}	10μs	maks. czas wybudzania przetwornika z trybu uśpienia
update rate	1MS/s	maksymalna częstotliwość małych zmian napięcia wyjściowego (do 1LSB)

Przetworniki DAC oczywiście mają parę bajarów. Pierwsza sprawa dotyczy zadawania wartości napięcia wyjściowego. Są trzy możliwości podawania tej wartości różniące się rozdzielcością (do wyboru 8 lub 12 bitów) i wyrównaniem danych (do lewej lub do prawej strony). W zależności od preferowanej opcji, wartość napięcia wpisuje się do innego rejestru:

231 w przypadku mikrokontrolera F334 bufor jest dostępny tylko na wyjściu DAC1_OUT1

232 zwiększenie obciążenia (zmniejszenie rezystancji) spowoduje odjechanie wartości napięcia wyjściowego od zadanej (przetwornik nie będzie w stanie wymusić odpowiedniej wartości napięcia), względnie coś się sfajczy

- wartość 8b wyrównana do prawej strony: rejestr ADC_DHR8Rx²³³
- wartość 12b wyrównana do prawej strony: rejestr ADC_DHR12Rx
- wartość 12b wyrównana do lewej strony: rejestr ADC_DHR12Lx

Po szczegóły odsyłam do RMa - są tam nawet rysunki pokazujące jak mają być umieszczone dane w rejestrach.

Podobnie jak przy ADC, przetworniki mogą pracować sprzężone ze sobą - w trybach podwójnych (*dual mode*). Dane dotyczące obu przetworników podawane są wówczas w rejestrach z końcówką „D” jak *dual* (ADC_DHR8RD, ADC_DHR12RD, ADC_DHR12LD).

Wszystkie powyższe rejestyry danych są ze sobą powiązane. Tzn. że po wpisaniu wartości do któregokolwiek z nich, pojawia się ona we wszystkich rejestrach danych. Przy czym w każdym rejestrze będzie inaczej wyrównana.

Rejestry danych DACów są buforowane. Nowe wartości wpisywane są w programie do rejestrów „tymczasowych” DAC_DHR (*data holding register*). Zostają one zapamiętane gdzieś w czeluściach przetwornika i:

- jeżeli nie korzystamy z zewnętrznego wyzwalania DACa to zostają przepisane do rejestrów ustalających napięcie na wyjściu przetwornika DOR (*data output register*) po 1 cyklu zegara szyny APB1
- jeżeli korzystamy z zewnętrznego wyzwalania to zostają przepisane do rejestrów DOR (*data output register*) po nadaniu trygierza (+ 3 cykle zegara APB1²³⁴)

i następuje zmiana napięcia na wyjściu DACa. Ustalenie nowej wartości napięcia zajmuje jakiś czas (patrz $t_{settling}$ w tabelce 14.1). Program nie ma możliwości zapisu bezpośredniego do rejestrów DOR. Ma natomiast możliwość odczytu ich aktualnej zawartości.

Źródłami wyzwalania dla przetworników mogą być liczniki, jakaś tam linia EXTI i wyzwalacz programowy (bit SWTRIG). Różnica między wyzwalaniem programowo (bit SWTRIG) a nie korzystaniem z wyzwalania w ogóle (tryb automatyczny) polega na tym, że po wyzwoleniu poprzez SWTRIG następuje jednokrotne przepisanie wartości z DHR do DOR. Bit SWTRIG jest sprzętowo kasowany i przetwornik czeka na kolejny wyzwalacz. W trybie automatycznym dane z DHR do DOR są przepisywane od razu po wpisaniu nowej wartości, bez żadnych wyzwalaczy.

Przetworniki mogą oczywiście współpracować z DMA. Współpraca ta wygląda następująco:

233 x to numer przetwornika

234 lub po 1 cyklu w przypadku wyzwalania bitem SWTRIG, taki wyjątek :)

- pojawia się trygierz DACa (ale nie SWTRIG!)
- DAC przepisuje wartość z DHR do DOR i wysyła żądanie DMA
- DMA przesyła nową wartość skądś do rejestru DHR DACa
- zapętlilj

Żądania DMA nie są kolejkowane. Tzn. że jeśli kolejne żądanie pojawi się zanim poprzednie zostanie do końca obsłużone, to to nowe zostanie olane. DAC w F103 nie ma możliwości generowania przerwań (a to ci peszek).

Zostały jeszcze dwa bajery związane z DACiem. DAC ma wbudowaną opcję, sprzętowego generowania szumu i przebiegu trójkątnego. Szum generowany jest w oparciu o układ rejestru przesuwającego z liniowym sprzężeniem zwrotnym (*linear feedback shift register*²³⁵, LFSR). Wygenerowany szum, o amplitudzie zależnej od zawartości bitów ADC_CR_MAMPx, jest dodawany do wartości z rejestru DHR. Wynik sumowania jest zapisywany w rejestrze wyjściowym DOR. Szum można wykorzystać np. przy nadpróbkowywaniu ADC (patrz nota AN2668), w zastosowaniach audio i w czym dusza zapragnie. Przebieg trójkątny również działa na zasadzie dodawania do DHR przed przepisaniem do DOR. Działanie generatora trójkąta jest proste i opiera się na liczniku. Licznik, co trygierz, zlicza sobie od 0 do wartości ADC_CR_MAMP i znowu do zera itd... Wartość licznika jest dodawana do DHR. Do czego to wykorzystać? Jakieś audio może? Oba generatory (szumu i trójkąta) wymagają wyzwalania przetwornika sygnałem zewnętrznym. Swoją drogą, ja bym tam oddał szum, trójkąt i dwa USARTy za generator sinusa :)

No to na koniec zostały tryby podwójne. Jest ich w sumie 11 i w większości nie mają sensu. Tzn. nie ma sensu nazywanie każdego z nich osobnym trybem. Lista (obecności):

- *independent trigger without wave generation*
- *independent trigger with same LFSR*
- *independent trigger with different LFSR*
- *independent trigger with same triangle*
- *independent trigger with different triangle*
- *simultaneous software start*
- *simultaneous trigger without wave generation*
- *simultaneous trigger with same LFSR*

235 jest schemat w RMie, jak ktoś chce to niech sobie analizuje :)

- *simultaneous trigger with different LFSR*
- *simultaneous trigger with same triangle*
- *simultaneous trigger with different triangle*

Nie będziemy omawiać tego po kolej, bo to bez sensu. Spojrzymy globalnie! Tryby niezależne (*independent*) polegają na tym, że dane dla obu przetworników (wartości napięcia) są podawane we wspólnym rejestrze z literką D na końcu, przy czym każdy przetwornik ma swój niezależny trygierz. Dane nie muszą być identyczne bo w rejestrach z „D” są wydzielone odrebine pola bitowe dla dwóch przetworników. Czyli, żeby była jasność: jedyna różnica między całkowicie niezależnym używaniem dwóch przetworników a trybem *dual independent* jest taka, że:

- przy niezależnym używaniu przetworników, wartość napięcia dla każdego z nich ustawiamy w osobnym rejestrze
- w trybie *dual independent* dwie wartości wpisujemy do jednego rejestru, wspólny reżestr danych powoduje ponadto, że wartości dla obu przetworników muszą mieć tą samą długość (8/12bit) i wyrównanie

Tryby równolegle wyzwalane (*simultaneous trigger*) to sytuacja, w której dwa przetworniki mają ustawione to samo źródło sygnału trygierującego... i nic poza tym. Równoległa praca bez wyzwalania (*simultaneous softstart*) to sytuacja w której oba przetworniki nie mają ustawionych trygierzy (czyli pracują w trybie automatycznym), a wartości napięcia wpisywane są do rejestrów danych z „D” na końcu.

Trybu podwójne *without wave generation...* czyli nie jest włączony generator szumu lub trójkąta. Jeśli w obu przetwornikach włączymy generatory szumu lub trójkąta to będziemy mieli tryby *with LFSR/triangle*. W zależności od ustawień wartości MAMP otrzymamy, w obu kanałach, takie same szumy/trójkąty (*with same LFSR/triangle*) lub różne (*with different...*). I to cała filozofia tych 11 trybów.

Znowu odnoszę wrażenie, że upychanie tych opisów na siłę w „tryby” jest bez sensu. I dam sobie rękę uciąć, że te 11 „trybów” nie wyczerpuje tematu. Zapewne można włączyć np. generator szumu tylko w pierwszym kanale a trójkąta w drugim i wtedy powstanie kolejny tryb o fikuśnej nazwie np. *dual simultaneous trigger with LFSR and triangle generation*.

Co warto zapamiętać z tego rozdziału?

- DAC, C/A, przetwornik cyfrowo analogowy zamienia sygnał cyfrowy na analogowy
- w omawianych mikrokontrolerach są dwa przetworniki o rozdzielczości 12 bitów
- wyprowadzenie przetwornika (nóżkę) należy wcześniej ustawić w tryb analogowy
- wyjście przetwornika ma dużą impedancję (małą „wydajność”)
- rejesty danych (zadanego napięcia) są buforowane

14.2. Zadania praktyczne (F103)

Zadanie domowe 14.1: uruchomić oba przetworniki DAC (osobno). Jeden ma dawać na wyjściu $\sim 1/3$ napięcia odniesienia, drugi $\sim 2/3$ napięcia odniesienia. Zmierzyć napięcia bez obciążenia i po obciążeniu wyjść rezystorami $10\text{k}\Omega$. Następnie włączyć buforowanie i powtórzyć pomiary. Ponadto przeprowadzić pomiary maksymalnej i minimalnej wartości napięć wyjściowych z i bez buforowania. Wyciągnąć mądre wnioski :)

Przykładowe rozwiązanie (F103, wyjścia analogowe PA4 i PA5):

```

1. int main(void) {
2.
3.     RCC->APB1ENR = RCC_APB1ENR_DACEN;
4.     RCC->APB2ENR = RCC_APB2ENR_IOPAEN;
5.
6.     gpio_pin_cfg(GPIOA, PA4, gpio_mode_input_analog);
7.     gpio_pin_cfg(GPIOA, PA5, gpio_mode_input_analog);
8.
9.     DAC->CR = DAC_CR_EN1 | DAC_CR_EN2 | DAC_CR_BOFF1 | DAC_CR_BOFF2;
10.    DAC->DHR12R1 = 4095/3;
11.    DAC->DHR12R2 = 2*4095/3;
12.
13.    while(1);
14.
15. }
```

1 - 7) włączenie zegarów i konfiguracja pinów w trybie analogowym. Tak jak wspominałem konfiguracja nóżek nie jest konieczna, bo po włączeniu DAC automatycznie się z nim łączą, ale wymuszenie konfiguracji analogowej zmniejsza pasożytniczy pobór prądu.

9) włączenie obu przetworników (bity EN) i wyłączenie buforowania (bity BOFF). Warto zwrócić uwagę na to, że przetworniki DAC są dwa, ale konfiguracja odbywa się w jednym rejestrze. Tak samo w układzie zegarowym jest tylko jeden bit odpowiedzialny za włączenie taktowania obu DACów.

10, 11) ustawienie wartości napięć. W programie nie wykorzystuję wyzwalania trygierzem, więc nowa wartość napięcia zostanie od razu wystawiona na wyjściu (tryb automatyczny).

Tabela 14.2 Wyniki pomiarów ($V_{DDA} = 3,313V$; $V_{ref+} = 3,313V$)

kanał	bez buforowania				z buforowaniem			
	obciążenie		napięcie		obciążenie		napięcie	
	brak	10kΩ	min.	maks.	brak	10kΩ	min.	maks.
OUT1 (PA4)	-	2,328V	1,835V	3,311V	-	1,108V	124mV	3,263V
OUT2 (PA5)	2,205V	0,972V	0,9mV	3,309V	2,211	2,209V	65,5mV	3,262V

Prawie mądro wnioski i obserwacje:

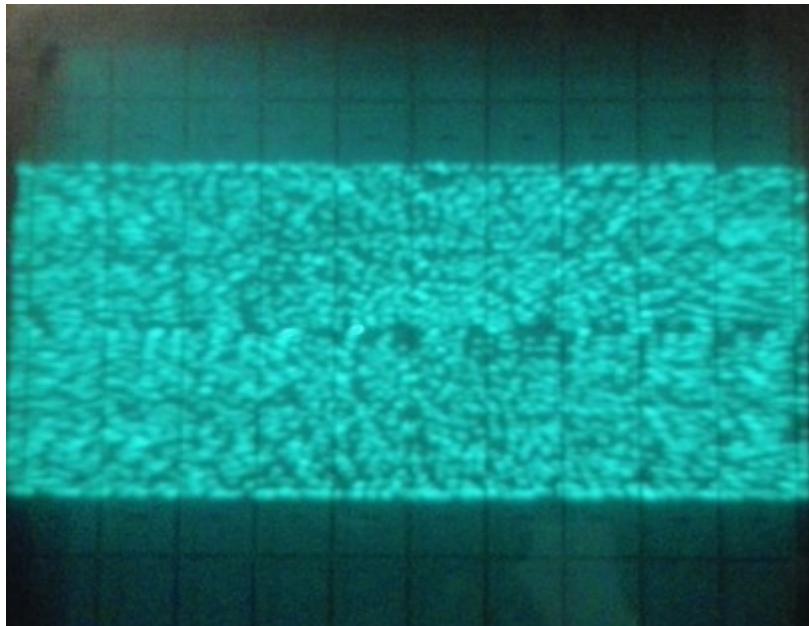
- na PA4, bez bufora, wychodzą bzdury (!) bo w zestawie HY-mini ten pin jest podciagnięty do V_{CC} przez rezystor 10kΩ. Trzeba uważać na takie pułapki korzystając z zestawów rozwojowych. Swego czasu, na forach, był wysyp tematów dotyczących nie działającego USARTu w którejś z płyt Discovery. Tam była podobna pułapka - coś wisiało na pinach tego USARTu.
- obciążenie wyjścia PA5, rezystorem 10kΩ do masy, spowodowało znaczny spadek napięcia - DAC się nie wyrobił z takim obciążeniem
- połączeniu buforowania, DAC radzi sobie na obu kanałach
- włączenie buforowania spowodowało jakąś tam zmianę napięcia na wyjściu
- połączeniu buforowania wzrosło minimalne napięcie jakie można uzyskać na wyjściu i zarazem spadło maksymalne

Zadanie domowe 14.2: uruchomić przetwornik DAC, ustawić ~1/2 napięcia referencyjnego na wyjściu, włączyć buforowanie. Włączyć najpierw generator szumu, potem trójkąta i sprawdzić efekt na oscylografie.

Przykładowe rozwiązanie (F103, wyjście analogowe PA5):

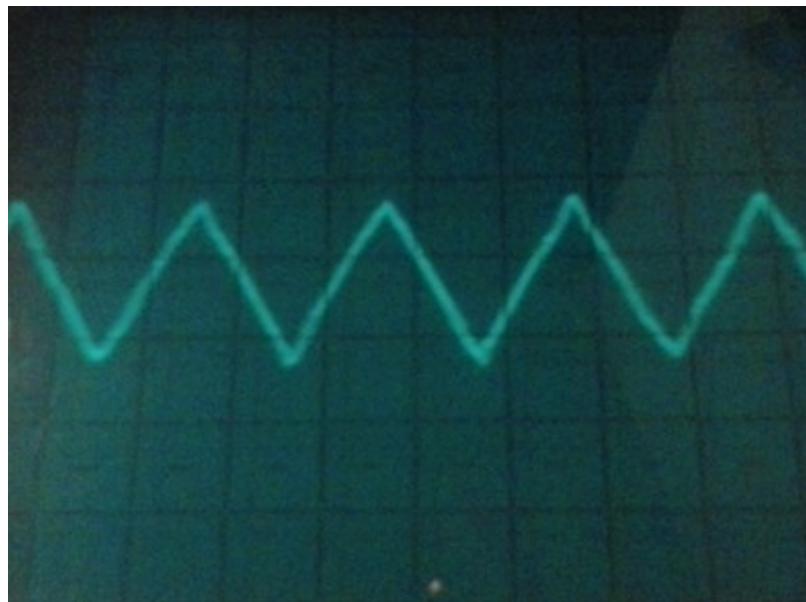
```
1. int main(void) {
2.
3.     RCC->APB1ENR = RCC_APB1ENR_DACEN | RCC_APB1ENR_TIM2EN;
4.     RCC->APB2ENR = RCC_APB2ENR_IOPAEN;
5.
6.     TIM2->PSC = 80-1;
7.     TIM2->ARR = 1;
8.     TIM2->CR2 = TIM_CR2_MMS_1;
9.     TIM2->CR1 = TIM_CR1_CEN;
10.
11.    gpio_pin_cfg(GPIOA, PA5, gpio_mode_input_analog);
12.
13.    DAC->CR = DAC_CR_EN2 | DAC_CR_WAVE2_0 | DAC_CR_TEN2 | DAC_CR_TSEL2_2 | 15<<24;
14.    DAC->DHR12R2 = 4095/2;
15.
16.    while(1);
17.
18. }
```

Programu nie ma co opisywać, bo nic tu specjalnego nie ma. Przypominam, że generator szumu i trójkąta działa tylko kiedy DAC jest wyzwalany sygnałem zewnętrznym i nie jest to wyzwalanie bitem SWTRIG. Efekty działania programu przedstawiam na „zrzutach”²³⁶ z oscylowizora.



Rys. 14.1. Przebieg uzyskany z wykorzystaniem generatora szumu

236 obiecuje, że jak mi ktoś podaruje cyfrowy oscyloskop to podmienię zrzuty na ładniejsze :)



Rys. 14.2. Przebieg uzyskany z wykorzystaniem generatora trójkąta

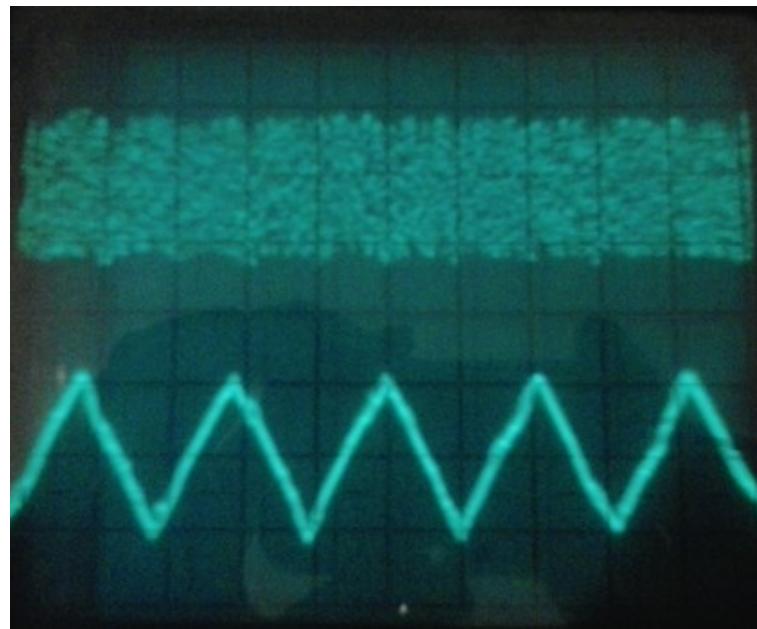
Zadanie domowe 14.3: gdzieś w poprzednim rozdziale (o [tu](#)) dałem sobie uciąć rękę, za to że jest możliwe uruchomienie trybu *dual simultaneous trigger with LFSR and triangle generation...* do dzieła mój Szogunie :)

Przykładowe rozwiązanie (F103, wyjścia analogowe PA4 i PA5):

```

1. int main(void) {
2.
3.     RCC->APB1ENR = RCC_APB1ENR_DACEN | RCC_APB1ENR_TIM2EN;
4.     RCC->APB2ENR = RCC_APB2ENR_IOPAEN;
5.
6.     TIM2->PSC = 80-1;
7.     TIM2->ARR = 1;
8.     TIM2->CR2 = TIM_CR2_MMS_1;
9.     TIM2->CR1 = TIM_CR1_CEN;
10.
11.    gpio_pin_cfg(GPIOA, PA4, gpio_mode_input_analog);
12.    gpio_pin_cfg(GPIOA, PA5, gpio_mode_input_analog);
13.
14.    DAC->CR = DAC_CR_EN2 | DAC_CR_WAVE2_0 | DAC_CR_TEN2 | DAC_CR_TSEL2_2 | 7<<24;
15.    DAC->CR |= DAC_CR_EN1 | DAC_CR_WAVE1_1 | DAC_CR_TEN1 | DAC_CR_TSEL1_2 | 7<<24;
16.    DAC->DHR12R1 = 4095/2;
17.    DAC->DHR12R2 = 4095/2;
18.
19.    while(1);
20. }
```

Rękę jednak zachowam:



Rys. 14.3. Przebieg uzyskany w autorskim trybie *dual simultaneous trigger with LFSR and triangle generation*

Zadanie domowe 14.4: wygenerować na wyjściu DAC przebieg „schodkowy” tak aby co 1ms napięcie na wyjściu rosło o 0,5V. Tzn:

- przez pierwszą milisekundę na wyjściu ma być ~0,5V
- od 1 do 2ms na wyjściu ma być około 1V
- od 2 do 3ms na wyjściu ma być około 1,5V
- ...
- od 5 do 6ms na wyjściu ma być 3V
- i zapętluj

Przykładowe rozwiązań (F103, wyjście analogowe PA5):

```
1. #define WSP (4095/3.3)
2.
3. int main(void) {
4.
5.     const uint16_t wartosci[] = { 0.5*WSP, 1*WSP, 1.5*WSP, 2*WSP, 2.5*WSP, 3*WSP };
6.
7.     RCC->APB1ENR = RCC_APB1ENR_DACEN | RCC_APB1ENR_TIM2EN;
8.     RCC->APB2ENR = RCC_APB2ENR_IOPAEN;
9.     RCC->AHBENR |= RCC_AHBENR_DMA2EN;
10.
11.    TIM2->PSC = 8000-1;
12.    TIM2->ARR = 1;
13.    TIM2->CR2 = TIM_CR2_MMS_1;
14.
15.    DMA2_Channel4->CMAR = (uint32_t)wartosci;
16.    DMA2_Channel4->CPAR = (uint32_t)&DAC->DHR12R2;
17.    DMA2_Channel4->CNDTR = 6;
18.    DMA2_Channel4->CCR = DMA_CCR4_CIRC | DMA_CCR4_DIR | DMA_CCR4_EN | DMA_CCR4_MINC |
19.        DMA_CCR4_MSIZE_0 | DMA_CCR4_PSIZE_0;
20.
21.    DAC->CR = DAC_CR_EN2 | DAC_CR_TEN2 | DAC_CR_TSEL2_2 | DAC_CR_DMAEN2;
22.    gpio_pin_cfg(GPI0A, PA5, gpio_mode_input_analog);
23.
24.    TIM2->CR1 = TIM_CR1_CEN;
25.
26.    while(1);
27.
28. }
```

Wreszcie coś ciekawszego. Do wygenerowania sekwencji napięć wykorzystać można albo przerwania (wpisywanie nowych wartości dla DACa w przerwaniu zegarowym) albo naszego ulubionego cichego przyjaciela czyli DMA. Oczywiście wybieramy bramkę nr dwa, bo im bardziej sprzętowo tym lepiej.

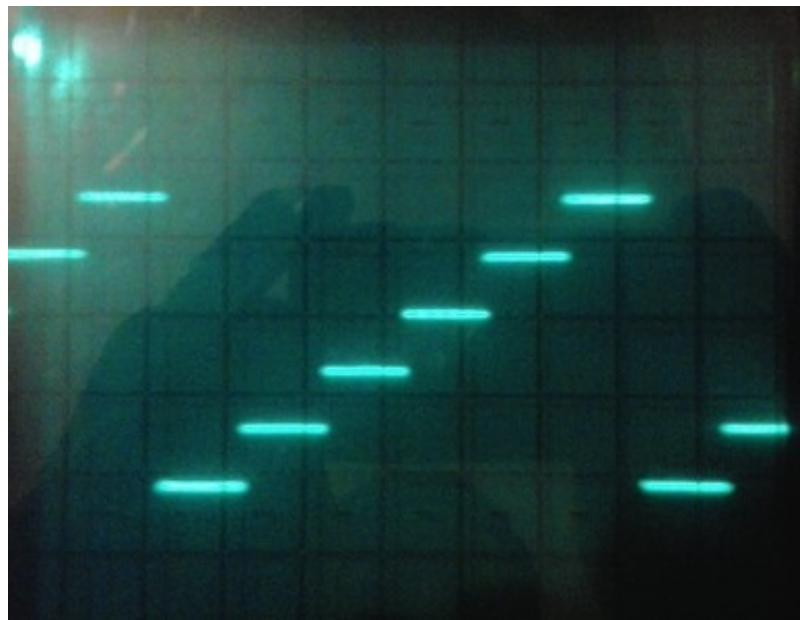
5) tablica z kolejnymi wartościami „napięć” (wartości wpisywane do rejestru DACa). Zgodnie z założeniem, że przykłady mają być szybkie w pisaniu i niekoniecznie eleganckie, wprowadziłem tam takie paskudztwo nazwane *WSP* (współczynnik). Sposób wyliczania wartości nikogo nie powinien dziwić, zwykła proporcja.

11 - 13) timerek będzie trygierzył DACa co 1ms, każdy trygierz spowoduje przepisanie nowej wartości z rejestru DHR do DOR oraz wygeneruje żądanie DMA...

15 - 19) a DMA prześle nową wartość z tablicy do rejestru DHR

21)łączamy DACa i ustawiamy wyzwalanie sygnałem z TIM2 oraz generowanie żądań DMA. Właściwie po włączeniu DACa powinna być chyba krótka przerwa aby zdążył się w pełni wybudzić - jak przy ADC. Zgodnie z tabelą 14.1 czas wybudzania przetwornika może dochodzić do 10μs

23) na sam koniec włączenie generatora trygierzy i paczamy na oscyloskop:



Rys. 14.4. Przebieg uzyskany w rozwiązyaniu zadania 14.4 (standardowo w tle widoczny operator aparatu komórkowego)

Zadanie domowe 14.5: a jakżeby inaczej... do tego skrycie dążyliśmy: niechaj DeAaCze wypluwa najidealniejszy z możliwych w przyrodzie przebieg (znaczy przebieg funkcji sinus) o amplitudzie równej około ~1,2V i częstotliwości 1kHz. Ponadto niechaj rdzeń uśpionym będzie natenczas, a ten opis kodem się stanie!

Przykładowe rozwiązanie (F103, wyjście analogowe PA5):

```

1. #include "sine_lut.h"
2.
3. int main(void) {
4.
5.     RCC->APB1ENR = RCC_APB1ENR_DACEN | RCC_APB1ENR_TIM2EN;
6.     RCC->APB2ENR = RCC_APB2ENR_IOPAEN;
7.     RCC->AHBENR |= RCC_AHBENR_DMA2EN;
8.
9.     TIM2->PSC = 4-1;
10.    TIM2->ARR = 4-1;
11.    TIM2->CR2 = TIM_CR2_MMS_1;
12.
13.    DMA2_Channel4->CMAR = (uint32_t)sine_lut;
14.    DMA2_Channel4->CPAR = (uint32_t)&DAC->DHR12R2;
15.    DMA2_Channel4->CNDFTR = 500;
16.    DMA2_Channel4->CCR = DMA_CCR4_CIRC | DMA_CCR4_DIR | DMA_CCR4_EN | DMA_CCR4_MINC |
17.        DMA_CCR4_MSIZE_0 | DMA_CCR4_PSIZE_0;
18.
19.    DAC->CR = DAC_CR_EN2 | DAC_CR_TEN2 | DAC_CR_TSEL2_2 | DAC_CR_DMAEN2;
20.    gpio_pin_cfg(GPI0A, PA5, gpio_mode_input_analog);
21.
22.    TIM2->CR1 = TIM_CR1_CEN;
23.
24.    __WFI();
25.    __BKPT();
26.
27. }
```

Algorytm działania programu jest identyczny jak w poprzednim przykładzie. Różnica polega na tym, że teraz próbki w tablicy są dobrane tak, aby wyszła z nich sinusoida a nie schodki. No i jest ich zdecydowanie więcej żeby przebieg był gładki jak aksamit. Próbki można sobie policzyć ręcznie na kartce, na przykład z takiego wzoru:

$$y_x = (\sin \left(\frac{2 \cdot \pi}{n_{samples}} \cdot x \right) + 1) \cdot \frac{4095}{V_{ref}} \cdot A$$

gdzie:

- y_x - wartość próbki numer x [-]
- $n_{samples}$ - liczba próbek (w tablicy) na okres sygnału [-]
- x - numer próbki [-]
- V_{ref} - napięcie odniesienia [V]
- A - amplituda przebiegu [V]

Ale to raczej podejście dla kogoś kto nie wie co z czasem zrobić. Inne opcje to wykorzystanie np. arkusza kalkulacyjnego, wszelkiej maści oprogramowań matematycznych (Matlab, Octave, MathCad, ...) czy generatorów online. Te ostatnie są szczególnie sympatyczne: wystarczy podać parametry przebiegu i generator wypluwa gotowiec, który wystarczy potem metodą Copy'ego i Paste'a dokleić sobie do programu. Ja skorzystałem z ostatniej metody i wygenerowałem 500 wartości, z których miał powstać sinus. Wartości (tablicę) wrzuciłem do osobnego pliku, żeby nie zaśmiecać listingu.

Teraz co do częstotliwości. Każdy trygierz (UEV licznika) powoduje przesłanie jednej wartości z tablicy do DACa. Cały okres sinusoidy składa się z założonej liczby ($n_{samples}$) próbek (w przykładzie jest ich 500). Stąd, można sobie wyprowadzić końcowy wzorek na częstotliwość sygnału wyjściowego:

$$f_{\sin} = \frac{f_{UEV}}{n_{samples}} = \frac{f_{tim}}{(ARR + 1) \cdot (PSC + 1) \cdot n_{samples}}$$

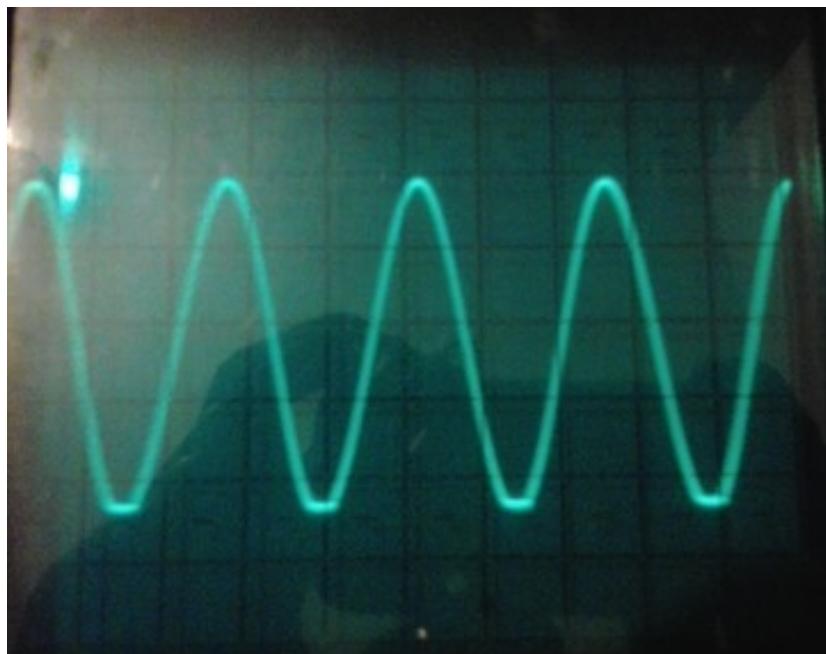
gdzie:

- f_{tim} - częstotliwość taktowania bloku licznika [Hz]
- ARR, PSC - wartości rejestrów konfiguracyjnych licznika [-]
- $n_{samples}$ - liczba próbek w tablicy [-]

W założeniach zadania było podane, że sinus ma mieć 1kHz. Sprawdźmy:

$$f_{\sin} = \frac{f_{tim}}{(ARR + 1) \cdot (PSC + 1) \cdot n_{samples}} = \frac{8\ 000\ 000}{4 \cdot 4 \cdot 500} = 1000 \text{ Hz}$$

Obliczenia muszą się zgadzać, bo przebieg na wyjściu rzeczywiście ma $\pm 1\text{kHz}$ (zmierzyłem) :)



Rys. 14.5. Przebieg sinusoidalny na wyjściu przetwornika DAC

Uważny (generalnie wystarczy żeby nie był ślepy) obserwator na pewno zauważycy, że temu sinusu coś nie teges wyglądają dolne wierzchołki. I racja... Coś mnie zamroczyło i zapomniałem o tym, że DAC (szczególnie z włączonym buforowaniem) nie zjeżdża z napięciem całkiem do zera (patrz rozdział 14.1). Próbki w tablicy powinny być lekko przesunięte w góre, tak aby napięcie nie spadało poniżej tych $\sim 150\text{mV}$. No ale coś mnie zamroczyło, zapomniałem, nie pomyślałem, nie chce mi się poprawiać... Sorry, taki mamy klimat.

Co warto zapamiętać z tego rozdziału?

- buforowanie wyjścia zmniejsza jego impedancję kosztem zakresu możliwych do uzyskania napięć i właściwości dynamicznych
- za pomocą tria DAC+TIM+DMA można uzyskać sprzętowy generator praktycznie dowolnego przebiegu (w szczególności np. sinusoidy)

14.3. Zadania praktyczne (F429)

Różnice w bloku DAC mikrokontrolera F429 są kosmetyczne (w stosunku do F103). Coś tam już zaznaczyłem przy parametrach datasheetycznych w rozdziale 14.1 (inny czas ustalania wartości napięcia na wyjściu czy coś takiego). Druga nowość jest związana z dodaniem flagi *DMA underrun*. Flaga jest ustawiana jeśli trygierze pojawią się za często i DMA nie wyrobi się z przesyłaniem danych. Przesyły DMA zostają wtedy wyłączone i jest możliwość wygenerowania przerwania.

Uwaga! Automatyczne wyłączanie żądań DMA po ustawieniu flagi *underrun* nie działa! ST skopało sprawę i przepuściło babola. Jest to opisane w *erracie*. Jako łok-erand zalecają ręczne wyłączanie odpowiedniego kanału DMA w przerwaniu funkcji *underrun*.

Poza tym DACi w obu prokach są identyczne.

Zadanie domowe 14.6: odpalić sinusa w F429. Żeby było „inaczej” niż poprzednio: 2kHz :)

Przykładowe rozwiązanie (F429, wyjście analogowe PA5):

```
1. #include "sine_lut.h"
2.
3. int main(void){
4.
5.     RCC->AHB1ENR |= RCC_AHB1ENR_GPIOAEN | RCC_AHB1ENR_DMA1EN;
6.     RCC->APB1ENR = RCC_APB1ENR_DACEN | RCC_APB1ENR_TIM2EN;
7.     __DSB();
8.
9.     gpio_pin_cfg(GPIOA, PA5, gpio_mode_analog);
10.
11.    TIM2->PSC = 4-1;
12.    TIM2->ARR = 4-1;
13.    TIM2->CR2 = TIM_CR2_MMS_1;
14.
15.    DMA1_Stream6->M0AR = (uint32_t)sine_lut;
16.    DMA1_Stream6->PAR = (uint32_t)&DAC->DHR12R2;
17.    DMA1_Stream6->NDTR = 500;
18.    DMA1_Stream6->FCR = DMA_SxFCR_DMDIS;
19.    DMA1_Stream6->CR = DMA_SxCR_CHSEL | DMA_SxCR_MSIZE_1 | DMA_SxCR_PSIZE_0 | DMA_SxCR_MINC |
20.        DMA_SxCR_CIRC | DMA_SxCR_DIR_0 | DMA_SxCR_EN;
21.
22.    DAC->CR = DAC_CR_DMAEN2 | DAC_CR_EN2 | DAC_CR_TEN2 | DAC_CR_TSEL2_2;
23.
24.    TIM2->CR1 = TIM_CR1_CEN;
25.    __WFI();
26.
27. }
```

Nie ma co omawiać, bo praktycznie nic się nie zmieniło w kodzie programu...

Co warto zapamiętać z tego rozdziału?

- wedle uznania :}

14.4. Odrobina odmiany (F334)

Mikrokontroler posiada dwa przetworniki DAC, przy czym pierwszy przetwornik ma dwa kanały (nowość!). Żeby było weselej, tryb podwójny (*dual*) wiąże ze sobą nie dwa przetworniki tylko dwa kanały pierwszego przetwornika. Przetwornik DAC2 trzyma się z boku.

Ze smutnych wiadomości tylko pierwszy kanał przetwornika DAC1 ma bufor wyjściowy. Pozostałe wyjścia nie posiadają bufora, więc mają masakrycznie dużą impedancję wyjściową (patrz zadanie 14.1).

Ciekawostka ostatnia: tylko DAC1 posiada możliwość generowania trójkąta i szumu. Tyle w temacie :) W pozostałych kwestiach przetworniki działają tak jak w F429.

Co warto zapamiętać z tego rozdziału?

- ostatnie zdanie

14.5. Uwagi końcowe

Dodatkowe informacje na temat przetworników DAC można znaleźć w notach aplikacyjnych:

- AN4566 *Extending the DAC performance of STM32 microcontrollers*
- AN3126 *Audio and waveform generation using the DAC in STM32 microcontrollers*

W szczególności polecam zwrócić uwagę na tabelę *Maximum sampling time for different STM32 microcontrollers* w AN4566. Tabela zawiera informacje o maksymalnej częstotliwości aktualizacji wartości DAC przez DMA dla różnych rodzin mikrokontrolerów. Innymi słowy tabela pokazuje jak szybko DMA może wstawiać nowe wartości do rejestrów DAC. Od tego, rzecz jasna, zależy maksymalna częstotliwość generowanego przebiegu.

Co warto zapamiętać z tego rozdziału?

- zawartość not aplikacyjnych :)

15. INTERFEJS USART („*VOLENTI NIHIL DIFFICILE*”²³⁷)

15.1. USART (F103)

Można szaleć F103 ma pięć USARTów. Na płytce HY-mini sprawia jest o tyle przyjemna, że wbudowana przejściówka USART ↔ USB pozwala na komunikację z komputerem bez dodatkowych osprzętów. Spróbujmy to uruchomić. Na początek trochę wiadomości *marketingowo-reklamowych* o USARTcie:

- komunikacja synchroniczna i asynchroniczna
- komunikacja half-duplex single wire
- programowalna długość pakietu, stop bitu itp.
- jakaś tam komunikacja LIN, Smartcard, IrDa SIR coś tam
- wykrywanie nadpisania danych
- komunikacja multiprocesorowa
- sprzętowa kontrola transferu

Jednym słowem możliwości jest dużo a większości i tak się nigdy nie używa... jak ktoś czuje potrzebę to niech sobie przeczyta opis w RMie. Mnie się nie chce. Przejdzmy do kodu :)

Zadanie domowe 15.1: niech mikrokontroler odbiera znaki ASCII z komputera (poprzez UART), inkrementuje kod odebranego znaku i odsyła nazad do PC.

²³⁷ „Dla chcącego nic trudnego.”

Przykładowe rozwiązanie (F103, dioda na PB0, USART1: RX na PA10, TX na PA9):

```
1. int main(void) {
2.
3.     RCC->APB2ENR = RCC_APB2ENR_USART1EN | RCC_APB2ENR_IOPAEN | RCC_APB2ENR_IOPBEN;
4.     gpio_pin_cfg(GPIOB, PB0, gpio_mode_output_PP_2MHz);
5.     gpio_pin_cfg(GPIOA, PA9, gpio_mode_alternate_PP_2MHz);
6.     gpio_pin_cfg(GPIOA, PA10, gpio_mode_input_floating);
7.
8.     SysTick_Config(8000000/2);
9.
10.    USART1->BRR = 8000000/9600;
11.    USART1->CR1 = USART_CR1 UE | USART_CR1_RXNEIE | USART_CR1_TE | USART_CR1_RE;
12.
13.    NVIC_EnableIRQ(USART1_IRQn);
14.    while(1) __WFI();
15.
16. }
17.
18. __attribute__((interrupt)) void USART1_IRQHandler(void){
19.     if ( USART1->SR & USART_SR_RXNE){
20.         USART1->SR &= ~USART_SR_RXNE;
21.         uint16_t tmp;
22.         tmp = USART1->DR;
23.         USART1->DR = tmp+1;
24.     }
25. }
26.
27. void SysTick_Handler(void){
28.     BB(GPIOB->ODR, PB0) ^= 1;
29. }
```

Cóż tu mamy ciekawego?

4, 5, 6) konfiguracja nóżek. PB0 to się jakaś dioda migająca zapłatała. PA9 to wyjście USARTu (TX), wybrana funkcja alternatywna. PA10 to RX, nóżka ustawiona jako wejściowa pływająca.

8) SysTick generuje przerwania, w których miga diodą. To zawsze miłe jak coś do nas miga :)

10) tu jest magia. W RMie można znaleźć jakiś nieziemsko zakręcony sposób obliczania wartości BRR w zależności od wybranej prędkości transmisji (tutaj akurat 9600 czegoś tam). Nie wiem co przyświecało twórcom tego opisu... może uzależnienie użytkowników od biblioteki, bo na piechotę nikomu nie będzie się chciało tego liczyć. Tak czy siak, proponuję z ciekawości poczytać twórczość w RM, popukać się w czoło i stosować wzorek jak w linii 10 (częstotliwość zegara szyny na której siedzi USART przez prędkość transmisji) :)

11) a tutaj jest reszta konfiguracji:

- włączenie USARTu
- włączenie przerwania od odebrania danych (RXNEIE - *RX Not Empty Interrupt Enable*)
- włączenie odbiornika i nadajnika

Wszelkie bity parzystości, wielo-stopy itd. itd. zostawiłem w wersji domyślnej (8 bitów danych, 1 stop, bez kontroli parzystości)

13, 14) włączenie przerwania i uśpienie procka

18 - 25) procedura obsługi przerwania. Nic nowego: sprawdzenie i skasowanie flagi, odczytanie odebranego znaku z rejestru danych do zmiennej pomocniczej, inkrementacja i wysłanie nazad.

Trudne? Cała konfiguracja USARTu to dwie linijki. Rejestr prędkości i cztery bity konfiguracji. W AVR zajęło by to więcej. No ale to przecież 32 bitowy mikrokontroler... bez biblioteki lepiej nie podchodzić. Z ciekawości poszukałem w Internecie przykładu konfiguracji USARTu z wykorzystaniem biblioteki SPL. Tak żeby sobie porównać (nie gwarantuję, że to działa):

Przykład konfiguracji interfejsu USART w oparciu o bibliotekę SPL²³⁸:

```

1. USART_InitTypeDef USART_InitStruct;
2. NVIC_InitTypeDef NVIC_InitStruct;
4.
5. USART_InitStructUSART_BaudRate = baudrate;
6. USART_InitStructUSART_WordLength = USART_WordLength_8b;
7. USART_InitStructUSART_StopBits = USART_StopBits_1;
8. USART_InitStructUSART_Parity = USART_Parity_No;
9. USART_InitStructUSART_HardwareFlowControl = USART_HardwareFlowControl_None;
10. USART_InitStructUSART_Mode = USART_Mode_Tx | USART_Mode_Rx;
11.
12. USART_Init(USART1, &USART_InitStruct);
13. USART_ITConfig(USART1, USART_IT_RXNE, ENABLE);
14.
15. NVIC_InitStruct.NVIC_IRQChannel = USART1_IRQn;
16. NVIC_InitStruct.NVIC_IRQChannelSubPriority = 0;
17. NVIC_InitStruct.NVIC_IRQChannelCmd = ENABLE;
18.
19. NVIC_Init(&NVIC_InitStruct);
20. USART_Cmd(USART1, ENABLE);
```

Oczywiście jestem całkowicie apolityczny i nie chcę nic sugerować, ale... u nas, bez biblioteki, to samo zajmuje 3 linijki kodu „na rejestrach” i nie wywołujemy żadnych dodatkowych funkcji (obejrzyj sobie źródła funkcji wołanych z powyższego kodu...) :) Dosyć uszczypliwości! Wprowadźmy małą modyfikację:

Zadanie domowe 15.2: mikrokontroler ma zwracać wszystko co dostanie poprzez USART (funkcja *echo*). Z jednym małym utrudnieniem - rdzeń ma cały czas pozostawać w uśpieniu

Przykładowe rozwiązanie (F103, USART1: RX na PA10, TX na PA9):

```

1. int main(void) {
2.
3.     RCC->APB2ENR = RCC_APB2ENR_USART1EN | RCC_APB2ENR_IOPAEN;
4.     RCC->AHBENR |= RCC_AHBENR_DMA1EN;
5.     gpio_pin_cfg(GPIOA, PA9, gpio_mode_alternate_PP_2MHz);
6.     gpio_pin_cfg(GPIOA, PA10, gpio_mode_input_floating);
7.
8.     USART1->BRR = 8000000/9600;
9.     USART1->CR3 = USART_CR3_DMAR;
10.    USART1->CR1 = USART_CR1_UE | USART_CR1_RXNEIE | USART_CR1_TE | USART_CR1_RE;
11.
12.    DMA1_Channel5->CPAR = (uint32_t)&USART1->DR;
13.    DMA1_Channel5->CMAR = (uint32_t)&USART1->DR;
14.    DMA1_Channel5->CNDTR = 1;
15.    DMA1_Channel5->CCR = DMA_CCR5_EN | DMA_CCR5_PSIZE_0 | DMA_CCR5_MSIZE_0 | DMA_CCR5_CIRC;
16.
17.    __WFI();
18.    __BKPT();
19.
20. }
```

238 źródło: <https://github.com/g4lvanix/STM32F1-workarea/blob/master/Project/USART-example/main.c>

Alleluja chwalmy DMA! Co się zmieniło względem poprzedniego kodu:

- wyleciała migająca na SysTicku dioda, żeby nie budziła procesora
- przy włączaniu zegarów doszło DMA
- w konfiguracji USARTu przybyło ustawienie bitu USART_CR3_DMAR, który odpowiada za generowanie żądań DMA po odebraniu ramki danych. Uwaga babol! Niepotrzebnie został bit RXNEIE... ale specjalnie nie przeszkadza bo przerwanie i tak nie jest włączone w NVICu.
- pojawiła się konfiguracja DMA:
 - kanał wybrany na podstawie rozpiszczy żądań DMA (*DMAx request mapping*) tak, aby przesył w tym kanale mógł być żądany przez zdarzenie USART1_RX
 - przesył z rejestru danych USARTu do tegoż samego (funkcja *echo*)
 - przesył w trybie kołowym (liczba przesyłów mogłaby być inna, nic to nie zmienia w konfiguracji kołowej bez inkrementacji)
 - wielkość przesyłanych danych - pół słowa (2B / 16b)
- na koniec procesor jest usypany i profilaktycznie, żeby się upewnić że się nie wybudza, wrzuciłem na końcu *breakpoint*

USART odbiera dane i wyzwala DMA, które odebrane dane wpycha do bufora nadawczego USARTu. Czyż to nie jest piękne w swojej prostocie? :) Jeszcze jeden przykładzik:

Zadanie domowe 15.3: zrobić "mostek UART", niech mikrokontroler odbiera dane poprzez jeden interfejs (z komputera) i wysyła je innym USARTem. Oczywiście ma być komunikacja w obie strony i bez udziału rdzenia.

Przykładowe rozwiązanie (F103, USART1: RX- PA10, TX-PA9; USART2: RX-PD6, TX-PD5):

```
1. int main(void) {
2.
3.     RCC->APB2ENR = RCC_APB2ENR_USART1EN | RCC_APB2ENR_IOPAEN | RCC_APB2ENR_IOPDEN |
4.             RCC_APB2ENR_AFIOEN;
5.     RCC->APB1ENR = RCC_APB1ENR_USART2EN;
6.     RCC->AHBENR |= RCC_AHBENR_DMA1EN;
7.
8.     gpio_pin_cfg(GPIOA, PA9, gpio_mode_alternate_PP_2MHz);
9.     gpio_pin_cfg(GPIOA, PA10, gpio_mode_input_floating);
10.    gpio_pin_cfg(GPIOD, PD5, gpio_mode_alternate_PP_2MHz);
11.    gpio_pin_cfg(GPIOD, PD6, gpio_mode_input_floating);
12.
13.    AFIO->MAPR = AFIO_MAPR_USART2_REMAP;
14.
15.    USART1->BRR = 8000000/9600;
16.    USART1->CR3 = USART_CR3_DMAR;
17.    USART1->CR1 = USART_CR1_UE | USART_CR1_RXNEIE | USART_CR1_TE | USART_CR1_RE;
18.
19.    USART2->BRR = 8000000/9600;
20.    USART2->CR3 = USART_CR3_DMAR;
21.    USART2->CR1 = USART_CR1_UE | USART_CR1_RXNEIE | USART_CR1_TE | USART_CR1_RE;
22.
23.    DMA1_Channel5->CPAR = (uint32_t)&USART1->DR;
24.    DMA1_Channel5->CMAR = (uint32_t)&USART2->DR;
25.    DMA1_Channel5->CNDTR = 1;
26.    DMA1_Channel5->CCR = DMA_CCR5_EN | DMA_CCR5_PSIZE_0 | DMA_CCR5_MSIZE_0 | DMA_CCR5_CIRC;
27.
28.    DMA1_Channel6->CPAR = (uint32_t)&USART2->DR;
29.    DMA1_Channel6->CMAR = (uint32_t)&USART1->DR;
30.    DMA1_Channel6->CNDTR = 1;
31.    DMA1_Channel6->CCR = DMA_CCR6_EN | DMA_CCR6_PSIZE_0 | DMA_CCR6_MSIZE_0 | DMA_CCR6_CIRC;
32.
33.    while(1);
34.
35. }
```

Jest tu w ogóle co opisywać? Dwa USARTy i dwa kanały DMA. Co pierwszy USART odbierze to drugi wysyła... i vice-versacze²³⁹. A! Jedną ciekawostkę dydaktyczną wcisnęłem w ten kod (w sensie, że nie było to potrzebne, ale dodałem żeby było ciekawiej). Linia 13 i *remap* funkcji alternatywnych na inne wyprowadzenia (patrz rozdział 3.6).

Co warto zapamiętać z tego rozdziału?

- USART ma dużo egzotycznych trybów
- STM32 ma dużo USARTów
- podstawowa konfiguracja to 2 linijki kodu
- sposób obliczania wartości BRR opisany w RM to jakaś pomyłka
- współpraca USARTu i DMA układa się bardzo sympatycznie

239 albo *versucze* jak ktoś woli :)

15.2. USART (F429)

Układy peryferyjne USART w F429 są bardzo podobne do F103. Nie czytałem całego rozdziału więc nie mam bladego pojęcia, jakimi szczegółami się różnią. USART, w takich podstawowych konfiguracjach, jest na tyle prostym układem że można go ustawić na podstawie samego opisu rejestrów. Szczególnie, że konfiguracja w F429 praktycznie niczym nie różni się od F103.

Zestaw STM32F420-Disco, nie ma niestety przejściówkę USART ↔ USB... ale po coś natrudziliśmy się w poprzednim zadaniu domowym („mostek USART” - zadanie 15.3). Płytką HY-mini będzie u mnie robić za sympatyczną przejściówkę USART ↔ USB :)

Zadanie domowe 15.4: niech mikrokontroler odsyła (*echo*) każdy odebrany znak, po uprzedniej inkrementacji kodu tego znaku. Program napisać z użyciem przerwań i dla urozmaicenia tak, aby w całym programie nie było ani jednej pętli!

Przykładowe rozwiązanie (F429, UART RX na PD2, TX na PC12):

```
1. int main(void){  
2.  
3.     RCC->APB1ENR = RCC_APB1ENR_UART5EN;  
4.     RCC->AHB1ENR |= RCC_AHB1ENR_GPIODEN | RCC_AHB1ENR_GPIOCEN;  
5.     __DSB();  
6.  
7.     gpio_pin_cfg(GPIOD, PD2, gpio_mode_AF8_OD_PD_L5);  
8.     gpio_pin_cfg(GPIOC, PC12, gpio_mode_AF8_PP_L5);  
9.  
10.    UART5->BRR = 16000000/9600;  
11.    UART5->CR1 = USART_CR1_UE | USART_CR1_RXNEIE | USART_CR1_TE | USART_CR1_RE;  
12.  
13.    NVIC_EnableIRQ(UART5_IRQn);  
14.    SCB->SCR |= SCB_SCR_SLEEPONEXIT_Msk;  
15.    __WFI();  
16.  
17. }  
18.  
19. void UART5_IRQHandler(void){  
20.     if (UART5->SR & USART_SR_RXNE){  
21.         UART5->SR &= ~USART_SR_RXNE;  
22.         uint16_t tmp;  
23.         tmp = UART5->DR;  
24.         UART5->DR = tmp+1;  
25.     }  
26. }
```

Konfiguracja USART jest praktycznie skopiowana z przykładów dotyczących F103. Zmieniła się tylko prędkość zegara przy obliczaniu wartości rejestru BRR. Na co jeszcze warto zwrócić uwagę? W F103 nóżka realizująca funkcję alternatywną USART_RX była ustawiana jako wejście, w F429 trzeba ją ustawić w trybie alternatywnym. Gdyby kogoś interesowało czemu akurat USART5 - odpowiadam: dlatego, że tylko nóżki związane z tym UARTEM nie są zajęte żadnym badziewiem na płytce Discovery.

Wątpliwości może też budzić to, że nagle zamiast nazwy USART zrobił się UART bez S²⁴⁰. „S” w nazwie oznacza *synchronous*. UART po prostu nie może pracować w trybie synchronicznym, czyli z oddzielną linią zegarową.

Gdyby ktoś pytał czemu zmienna w przerwaniu jest 16b a nie 8b... nie mam pojęcia, a nie chce mi się zmieniać.

Zadanie domowe 15.5: *echo* z wykorzystaniem DMA.

Przykładowe rozwiążanie (F429, UART RX na PD2, TX na PC12):

```
1. int main(void){  
2.  
3.     RCC->APB1ENR = RCC_APB1ENR_UART5EN;  
4.     RCC->AHB1ENR |= RCC_AHB1ENR_GPIODEN | RCC_AHB1ENR_GPIOCEN | RCC_AHB1ENR_DMA2EN;  
5.     __DSB();  
6.  
7.     gpio_pin_cfg(GPIOD, PD2, gpio_mode_AF8_OD_PD_LS);  
8.     gpio_pin_cfg(GPIOC, PC12, gpio_mode_AF8_PP_LS);  
9.  
10.    DMA2_Stream0->PAR = (uint32_t)&UART5->DR;  
11.    DMA2_Stream0->M0AR = (uint32_t)&UART5->DR;  
12.    DMA2_Stream0->NDTR = 1;  
13.    DMA2_Stream0->CR = DMA_SxCR_DIR_1 | DMA_SxCR_PSIZE_0 | DMA_SxCR_MSIZE_0;  
14.  
15.    UART5->BRR = 16000000/9600;  
16.    UART5->CR3 = USART_CR3_DMAR;  
17.    UART5->CR1 = USART_CR1_UE | USART_CR1_TE | USART_CR1_RE | USART_CR1_RXNEIE;  
18.  
19.    NVIC_EnableIRQ(UART5_IRQn);  
20.    while(1);  
21.  
22.}  
23.  
24. void UART5_IRQHandler(void){  
25.     if (UART5->SR & USART_SR_RXNE){  
26.         UART5->SR &= ~USART_SR_RXNE;  
27.  
28.         DMA2->LIFCR = DMA2->LISR;  
29.         DMA2_Stream0->CR |= DMA_SxCR_EN;  
30.     }  
31. }  
32.
```

Ten przykład nie jest zbyt udany. Już mówię czemu. Przy wyborze konkretnego USARTu/UARTu kierowałem się tym, aby wyprowadzenia mikrokontrolera mogące pełnić funkcję alternatywną RX i TX nie były zajęte żadnym dziadostwem na płytce Discovery, np. wielce przydatnym żyroskopem w obudowie, która uniemożliwia odlutowanie go w warunkach amatorskich... Tak został wybrany UART5. Teraz przyszedł czas na DMA. W RMie odszukałem numer kontrolera i strumienia DMA związanego ze zdarzeniem UART5_RX, jest to DMA1 strumień 0. Czyli tylko ten strumień może być wyzwalany (sprzętowo) przez USART5_RX. Kłopot polega na tym, że aby zrealizować funkcję *echo*, musimy przesyłać dane z rejestru UART5_DR do tegoż samego. UART5 jest układem peryferyjnym szyny APB1. Jak popatrzymy na schematy

240 jak Jarosław Psikuta z pewnego polskiego filmu

obrazujące do jakich układów mają dostęp poszczególne kontrolery DMA²⁴¹, to możemy zobaczyć²⁴², że DMA1 do układów z szyny APB1 może się dostać tylko poprzez swój *Peripheral Port. Memory Port* ma dostęp do różnych bloków, ale jak na złość, nie do APB1. Przesyły DMA zachodzą między dwoma portami (*Peripheral* i *Memory* - pamiętasz że dwa rejestrze DMA przechowujące adresy źródłowy i docelowy miały *Peripheral* i *Memory* w nazwie?). Żeby było możliwe przesłanie czegoś z rejestrów układowych szyny APB1 do tego samego układu, oba porty kontrolera DMA muszą mieć dostęp do tej szyny. Możliwość „dotarcia” do APB1 na obu portach ma kontroler DMA2. Niestety strumieni tego kontrolera nie może wyzwalać układem UART5. I kółko się zamknie.

W przykładowym kodzie wykorzystałem DMA2 (bo może przesyłać dane z UART5 do UART5), ale wyzwalanie następuje ręcznie w przerwaniu od odebrania danych przez UART. DMA jest ustawione w trybie *memory to memory* aby możliwe było programowe wyzwalanie transferu.

Przypominam że DMA w F429 ma taką miłą funkcję, że po tym jak zatrzyma się wskutek wyzerowania licznika transferów, wystarczy tylko ustawić ponownie bit EN aby uruchomić strumień ponownie z niezmienionymi ustawieniami. Czyli nie trzeba na nowo konfigurować np. licznika transferów tak jak miało to miejsce w F103. W przerwaniu od odebrania danych przez UART, następuje:

- skasowanie flag strumienia DMA
- wyzwolenie nowego przesyłu poprzez ustawienie bitu EN

I jakoś to działa... choć nie tak elegancko jakbym sobie tego życzył.

Co warto zapamiętać z tego rozdziału?

- to samo co z rozdziału 15.1

15.3. USART (F334)

Ten mikrokontroler tak już ma, że wszystko w nim musi być postawione na głowie :) Ale za to płytka Nucleo ma bardzo sympatyczną funkcję. Wyposażona jest bowiem w programator/debugger ST-Link V2-1. Ta wersja ST-Linka, poza programowaniem i debagiem, udostępnia jeszcze dwie funkcje. Jedna z nich jest nader upierdliwa (patrz dodatek

241 System implementation of the two DMA controllers w RMie

242 ale trzeba się wpatrzeć

7). Druga jest dla odmiany wielce wygodna i użyteczna. Jest to funkcja przejściówka USB ↔ UART. Dodatkowo, żebyśmy nie musieli się babrać w „hardwarze”, sygnały TxD i RxD z przejściówką są od razu połączone z wyprowadzeniami mikrokontrolera (można to zmienić rozlutowując dwa mostki na płytce):

- PA2 - USART2 TX
- PA3 - USART3 RX

Generalnie rzecz ujmując USART ma multum opcji, trybów i możliwości. Jednak jeśli odrzucimy wszystkie opcje związane z dedykowanymi, specjalnymi trybami (np. LIN, IrDA, SmartCard, ModBus) to resztę da się ogarnąć.

Co ciekawego spotka nas w F334? Ano między innymi:

- kilka ciekawych ustawień dotyczących próbkowania sygnału odbieranego
- oddzielne rejesty dla danych odbieranych (RDR) i wysyłanych (TDR)
- możliwość wybudzania mikrokontrolera ze stanu uśpienia *Stop Mode* po odebraniu danych
- funkcja automatycznej detekcji prędkości

Odnośnie pierwszej kropki, do wyboru mamy dwie opcje: częstotliwość próbkowania sygnału i ilość próbek na których podstawie określany jest poziom sygnału. Próbkowanie może odbywać się z wielokrotnością x8 lub x16 prędkości transmisji (baudrate). Ten parametr określa, ile razy linia odbiorcza będzie próbkowana w czasie odpowiadającym trwaniu jednego bitu w ramce danych. Wzrokowców odsyłam do RMa: *Universal synchronous asynchronous receiver transmitter (USART)* → rozdział *Receiver* → sekcja *Selecting the clock source and the proper oversampling method* → rysunki *Data sampling when oversampling by 16* i *Data sampling when oversampling by 8*. Według dokumentacji, próbkowanie x16 zwiększa tolerancję odbiornika na „dewiacje” częstotliwości zegara nadajnika. Wadą jest ograniczenie prędkości transmisji, nie może przekroczyć 1/16 częstotliwości zegara taktującego interfejs USART.

Druga opcja dotyczy ilości sprawdzanych próbek. Chodzi o to ile z tych 8 lub 16 próbek, będzie decydować o tym jaki bit zostanie odczytany. Mamy do wyboru dwa warianty:

- pojedyncza (środkowa) próbka decyduje o stanie bitu

- trzy (środkowe) próbki decydują o stanie bitu, przy czym jeśli nie są identyczne to zgłaszany jest błąd ramki czy coś takiego

Założmy, że wybraliśmy próbkowanie x8, czyli w czasie nadawania jednego bitu, odbiornik 8 razy sprawdza stan linii odbiorczej. Pierwsza kropka oznacza, że z tych ośmiu próbek wybieramy środkową i uznajemy, że odebraliśmy bit o takiej właśnie wartości jak ta próbka (0 lub 1). Druga kropka wygląda tak, że bierzemy trzy środkowe próbki i sprawdzamy czy są takie same. Jeśli są to zakładamy, że taki właśnie bit odebraliśmy. Jeśli próbki nie są jednakowe to znaczy, że coś jest nie halo (np. odbiornik i nadajnik pracują z różną częstotliwością lub pojawiły się jakieś magiczne zakłócenia) i zgłaszany jest błąd ramki danych. Jak nietrudno się domyślić, sprawdzanie trzech próbek zwiększa odporność transmisji na zakłócenia. Pojedyncza próbka może akurat trafić na jakiś „zakłóceniom szpilkę”. Z trzeba próbками jest to już mniej prawdopodobne.

Dosyć gadania, reszta wyjdzie w praniu.

Zadanie domowe 15.6: generalnie rzecz ujmując: echo. Im bardziej sprzętowo tym zacniej.

Przykładowe rozwiązanie (F334):

```

1. int main(void){
2.
3.     RCC->AHBENR = RCC_AHBENR_GPIOAEN | RCC_AHBENR_DMA1EN;
4.     RCC->APB1ENR = RCC_APB1ENR_USART2EN;
5.
6.     gpio_pin_cfg(GPIOA, PA5, gpio_mode_out_PP_LS);
7.     gpio_pin_cfg(GPIOA, PA2, gpio_mode_AF7_PP_HS);
8.     gpio_pin_cfg(GPIOA, PA3, gpio_mode_AF7_PP_HS);
9.
10.    DMA1_Channel6->CPAR = (uint32_t) &USART2->RDR;
11.    DMA1_Channel6->CMAR = (uint32_t) &USART2->TDR;
12.    DMA1_Channel6->CNDTR = 1;
13.    DMA1_Channel6->CCR = DMA_CCR_CIRC | DMA_CCR_EN;
14.
15.    uint16_t divider = 2*8000000/9600;
16.    USART2->BRR = (divider & 0xffff0u) | (divider & 0xfu) >> 1;
17.    USART2->CR3 = USART_CR3_DMAR;
18.    USART2->CR1 = USART_CR1_OVER8 | USART_CR1_TE | USART_CR1_RE | USART_CR1 UE;
19.
20.    SysTick_Config(8000000/4);
21.    while(1);
22. }
23.
24. void SysTick_Handler(void){
25.     GPIOA->ODR ^= PA5;
26. }
```

Chyba nie będziemy się bawić w omawianie tego linijka po linijce, bo nic odkrywczego tu nie ma :) Zresztą przypuszczam, że już całkiem nieźle radzisz sobie samodzielnie w dokumentacji STMa, więc nie będę przepisywał opisów bitów z RM, bo to nie ma sensu.

O rozdzielonych rejestrach (nadawczym i odbiorczym) dla przesyłanych danych już pisałem. W linijce 18 ustawiam bit o nazwie OVER8. Jest on odpowiedzialny za wybór

częstotliwości próbkowania (8 lub 16). Domyślnie wybrana jest opcja 16 próbek na bit. Na potrzeby przykładu zmieniłem to na 8 próbek, gdyż zmianie ulega wtedy wzór na wartość BRR. Przy domyślnej konfiguracji (16 próbek), wzór jest taki jaki poznaliśmy wcześniej, tj. częstotliwość taktowania podzielone przez prędkość transmisji. Jeżeli jednak zmienimy sposób próbkowania na 8 to wzór przyjmuje postać:

$$USARTDIV = \frac{2 \cdot f_{clk}}{baud}$$

Ponadto obliczona wartość dzielnika nie powinna być wpisana do rejestru BRR bezpośrednio. Należy jeszcze wykonać małe matematyczne hokus-pokus:

- cztery najmłodsze bity wartości USARTDIV należy przesunąć w prawo o jedną pozycję - to będzie wartość trzech najmłodszych bitów pola BRR
- czwarty bit pola BRR powinien być wyzerowany
- resztę wartości USARTDIV (bity [15:4]) przepisujemy bez zmian

Generalnie robi to linijka 16 listingu. Przy czym, jeszcze raz, żeby nie było wątpliwości - specjalnie wybrałem próbkowanie x8 żeby było weselej. Jeżeli nie będziesz ruszał próbkowania to sposób obliczania BRR nie ulega zmianie względem F103/F429.

Zadanie domowe 15.7: okiełznać automatyczną detekcję prędkości transmisji.

Przykładowe rozwiązanie (F334):

```
1. int main(void){
2.
3.     RCC->AHBENR = RCC_AHBENR_GPIOAEN;
4.     RCC->APB2ENR = RCC_APB2ENR_USART1EN;
5.
6.     gpio_pin_cfg(GPIOA, PA5, gpio_mode_out_PP_LS);
7.     gpio_pin_cfg(GPIOA, PA9, gpio_mode_AF7_PP_HS);
8.     gpio_pin_cfg(GPIOA, PA10, gpio_mode_AF7_PP_HS);
9.
10.    USART1->BRR = 8000000/9600;
11.    USART1->CR2 = USART_CR2_ABREN | USART_CR2_ABRMODE_0 | USART_CR2_ABRMODE_1;
12.    USART1->CR1 = USART_CR1_RXNEIE | USART_CR1_TE | USART_CR1_RE | USART_CR1 UE;
13.
14.    NVIC_EnableIRQ(USART1_IRQn);
15.    SysTick_Config(8000000/2);
16.    while(1);
17. }
18.
19. void USART1_IRQHandler(void){
20.
21.     if(USART1->ISR & USART_ISR_ABRE){
22.         USART1->RQR = USART_RQR_ABRRQ;
23.     }
24.
25.     if(USART1->ISR & USART_ISR_RXNE){
26.         char data = USART1->RDR;
27.         if(data == 'B') USART1->RQR = USART_RQR_ABRRQ;
28.         USART1->TDR = data;
29.     }
30. }
31. }
```

Uwaga! Niepostrzeżenie nastąpiła zmiana interfejsu na USART1. To dlatego, że nie wszystkie instancje peryferiala mają dostępne wszystkie fazy. Np. funkcję automatycznego wykrywania prędkości ma jedynie USART1 (odsyłam do RMa, rozdział *USART implementation*). W związku ze zmianą interfejsu wymagane jest dokonanie kilku zmian w programie i sprzęcie. W sprzęcie połączylem wyprowadzenia RX/TX ST-Linka (złącze CN3) z wyprowadzeniami mikrokontrolera PA9 i PA10 (RX i TX przy konfiguracji alternatywnej), oczywiście łączymy „na krzyż”. Trzeba pamiętać, że linie RX/TX ST-Linka w dalszym ciągu połączone są z wyprowadzeniami PA2 i PA3 mikrokontrolera (USART2)! Jednak jeśli nie będziemy z nich korzystać to nie będą przeszkadzać.

W programie natomiast trzeba zmodyfikować konfigurację portów oraz podmienić wszystkie USART2 na USART1. Dodatkowo, gdybyśmy dalej chcieli wykorzystywać DMA, to należałoby zmienić kanał DMA na taki, który może być wyzwalany przez interfejs USART1.

Działanie automatycznej detekcji prędkości jest dosyć proste. Po włączeniu funkcji (bit USART_CR2_ABREN), USART oczekuje na określony znak i na jego podstawie sam ustawia prędkość transmisji. Magiczny znak zależy od wybranego trybu:

- *mode0*: każdy znak z bitem 1 na początku, prędkość określana jest na podstawie długości bitu *start ramki danych*

- *mode1*: każdy znak z sekwencją 10xx na początku, prędkość określana jest na podstawie długości bitu *start* i pierwszego bitu danych ramki
- *mode2*: znak 0x7F (*delete*), prędkość określana jest dwuetapowo:
 - najpierw „wstępnie” na podstawie czasu trwania bitu *start* (ta prędkość jest wykorzystywana do dalszego próbkowania)
 - dokładna wartość jest uzyskiwana na podstawie pomiaru bitów 0 - 6 z ramki danych
- *mode3*: znak 0x55 (U), tym razem prędkość określana jest trzystopniowo

Jeżeli wszystko pójdzie dobrze to ustawiana jest flaga ABRF + RXNE oraz nowa wartość rejestru BRR. Jeżeli detekcja się nie powiedzie (np. zostanie odebrany inny znak niż oczekiwany) to nowa prędkość nie jest ustawiana, za to ustawiana jest flaga błędu ABRE. Można wtedy ponowić próbę automatycznej detekcji poprzez wpisanie jedynki do USARTx_RQR_ABRRQ.

Nie wdając się w szczegóły, przykładowy program wykorzystuje automatyczną detekcję zegara w trybie 3-cim (duże U). W przerwaniu odbiorczym program sprawdza czy odebrano znak 'B', jeżeli tak to aktywuje funkcję automatycznej detekcji prędkości. Dodatkowo, jeżeli pojawi się błąd (flaga ABRE) to również następuje ponowne włączenie funkcji. Po uruchomieniu programu USART oczekuje na literkę 'U', aby na jej podstawie rozpoznać prędkość komunikacji (wartość BRR nie ma znaczenia, ale musi być niezerowa). Jeżeli detekcja prędkości zakończy się powodzeniem, to program realizuje funkcję echo. Jeżeli się nie uda²⁴³, np. odebrany zostanie inny znak, to program ponownie aktywuje funkcję detekcji prędkości.

Włączenie funkcji następuje również po odebraniu znaku 'B'. Tym sposobem można zrobić taki myk: wysłać B (aktywacja automatycznej detekcji), wybrać inną prędkość transmisji, wysłać U (w celu synchronizacji) i dalej komunikować się już z wykorzystaniem nowej prędkości. Oczywiście to tylko zabawowy przykład.

Jeszcze jedna ciekawa rzecz mi przyszła do głowy i postanowiłem ją opisać. Każdy kto zetknął się z komunikacją wykorzystującą przesyłanie tekstowych poleceń - np. komend AT czy innych tokenowatych danych (typu: *AT+LED1=ON* itp...) - na pewno zastanawiał się, czy nie dałoby się zmusić USARTu do generowania przerwania po odebraniu określonego znaku (np. znaku końca linii CR czy LF). Wtedy można by pakować odebrane dane do bufora z wykorzystaniem DMA a procesor kłopotać dopiero po odebraniu całego polecenia. Na Elektrodzie było kiedyś podobne pytanie - czy można zmusić USART do generowania przerwania po odebraniu konkretnego znaku (znaku końca ramki), padła odpowiedź że niet. A, rzecz jasna, nie produkowałbym się, gdyby się nie dało. Da się uzyskać taki efekt, zaprzegając do tego mechanizm

²⁴³ „Uda zawsze są dwa - albo się uda albo się nie uda.”

„adresowania” wykorzystywany w trybie komunikacji wieloprocesorowej. Odsyłam do opisu pola bitowego USARTx_CR2_ADD (źródło RM):

This bit-field gives the address of the USART node or a character code to be recognized. This is used in multiprocessor communication during Mute mode or Stop mode, for wakeup with 7-bit address mark detection. The MSB of the character sent by the transmitter should be equal to 1. It may also be used for character detection during normal reception, Mute mode inactive (for example, end of block detection in ModBus protocol). In this case, the whole received character (8-bit) is compared to the ADD[7:0] value and CMF flag is set on match. This bit field can only be written when reception is disabled (RE = 0) or the USART is disabled (UE=0)

Jak dla mnie zabrzmiało to zachęcająco :] No i się nie pomyliłem - it's alive! Poniżej prosty przykład: jeżeli odebrany zostanie znak 13 (CR) to ustawiana jest flaga CMF (*Character Match Flag*), generowane jest przerwanie i program „printuje” napis „[CR]”

Ciekawostka (F334):

```
1. int main(void){
2.
3.     RCC->AHBENR = RCC_AHBENR_GPIOAEN;
4.     RCC->APB2ENR = RCC_APB2ENR_USART1EN;
5.
6.     gpio_pin_cfg(GPIOA, PA5, gpio_mode_out_PP_LS);
7.     gpio_pin_cfg(GPIOA, PA9, gpio_mode_AF7_PP_HS);
8.     gpio_pin_cfg(GPIOA, PA10, gpio_mode_AF7_PP_HS);
9.
10.    USART1->BRR = 8000000/9600;
11.    USART1->CR2 = 13 << 24;
12.    USART1->CR1 = USART_CR1_RXNEIE | USART_CR1_TE | USART_CR1_RE | USART_CR1_UE |
13.        USART_CR1_CMIE;
14.
15.    NVIC_EnableIRQ(USART1_IRQn);
16.    SysTick_Config(8000000/2);
17.    while(1);
18. }
19.
20. void USART1_IRQHandler(void){
21.
22.     if(USART1->ISR & USART_ISR_RXNE){
23.         char data = USART1->RDR;
24.         USART1->TDR = data;
25.     }
26.
27.     if(USART1->ISR & USART_ISR_CMF){
28.         USART1->ICR = USART_ICR_CMCF;
29.         const char *txt = "[CR]\n";
30.         while(*txt) {
31.             while(~USART1->ISR & USART_ISR_TXE);
32.             USART1->TDR = *txt++;
33.         }
34.     }
35. }
```

Co warto zapamiętać z tego rozdziału?

- nie każdy USART obsługuje wszystkie funkcje

16. PAMIĘĆ FLASH I BOOTOWANIE („*VARIATIO DELECTAT*”²⁴⁴)

16.1. Pamięć Flash

Pamięć Flash w STMach składa się z kilku kawałków. W przypadku F103 jest to:

- *Main Block* (0x0800 0000 - 0x0807 FFFF) - tutaj siedzi nasz program i dane tylko do odczytu
- *Information Block* - tutaj znajdują się:
 - *System Memory* (0x1FFF F000 - 0x1FFF F7FF) - firmowy *bootloader*
 - *Option Bytes* (0x1FFF F800 - 0x1FFF F80F) - bajty konfiguracyjne

W F429 pamięć Flash wygląda następująco:

- *Main Memory* - pamięć programu i danych:
 - *Bank 1* (0x0800 0000 - 0x080F FFFF)
 - *Bank 2* (0x0810 0000 - 0x081F FFFF)
- *System Memory* (0x1FFF 0000 - 0X1FFF 77FF) - firmowy *bootloader*
- *OTP* (0x1FFF 7800 - 0X1FFF7A0F) - pamięć jednokrotnego programowania (*one time programmable*), do użytku według własnego widzi-mi-się użytkownika mikrokontrolera (np. jakiś numer fabryczny)
- *Option Bytes* - bajty konfiguracyjne:
 - *Bank 1* (0x1FFF C000 - 0x1FFF C00F)
 - *Bank 2* (0x1FFE C000 - 0x1FFE C00F)

Zaś w F334 sprawy mają się następująco:

- *Main Memory* - pamięć programu i danych - 0x0800 0000 - 0x0800 FFFF
- *Information Block* - tutaj znajdują się:
 - *System Memory* (0x1FFF D800 - 0x1FFF F7FF) - firmowy *bootloader*
 - *Option Bytes* (0x1FFF F800 - 0x1FFF F80F) - bajty konfiguracyjne

W mikrokontrolerach AVR pamięć Flash po prostu... była. Nikt się nią specjalnie nie przejmował. W STM musimy pamiętać o kilku drobiazgach. Pamięć Flash, a właściwie bardziej kontroler tej pamięci, ma kilka opcji konfiguracji. Dobrać się do nich można poprzez rejestr FLASH_ACR. Część z nich dotyczy bajerów zwiększających wydajność systemu (wszelkie z

²⁴⁴ „*Odmiana sprawia przyjemność.*”

prefetch, cache w nazwie). Te opcje generalnie można włączyć i zapomnieć²⁴⁵. Bardziej interesujące są opcje związane z opóźnianiami: *latency (wait states)*. Chodzi o to, że pamięć Flash jest wolna w porównaniu z CPU. W związku z tym wprowadza się opóźnienia (*wait states*). Określają one co ile cykli CPU, może następować odczyt pamięci Flash. Brzmi zawile? W praktyce nie jest tak źle:

- dla STM32F103 i STM32F334 wielkość opóźnień jest związana z częstotliwością zegara systemowego SYSCLK²⁴⁶ ²⁴⁷:

Tabela 16.1 Wymagane opóźnienia przy dostępie do pamięci flash w F103 i F334

SYSCLK [MHz]		wymaga liczba WAITSTATES
od	do	
0	24	0
24	48	1
48	72	2

- dla STM32F429 wielkość opóźnień wynika z częstotliwości HCLK²⁴⁶ i napięcia zasilającego mikrokontroler

Tabela 16.2 Wymagane opóźnienia przy dostępie do pamięci flash w STM32F429 (częstotliwości HCLK podane w MHz)

wymagana liczba WAITSTATES	napięcie zasilania mikrokontrolera			
	1,8V - 2,1V	2,1V - 2,4V	2,4V - 2,7V	2,7V - 3,6V
0 WS (1 CPU cycle)	0 < HCLK ≤ 20	0 < HCLK ≤ 22	0 < HCLK ≤ 24	0 < HCLK ≤ 30
1 WS (2 CPU cycle)	20 < HCLK ≤ 40	22 < HCLK ≤ 44	24 < HCLK ≤ 48	30 < HCLK ≤ 60
2 WS (3 CPU cycle)	40 < HCLK ≤ 60	44 < HCLK ≤ 66	48 < HCLK ≤ 72	60 < HCLK ≤ 90
3 WS (4 CPU cycle)	60 < HCLK ≤ 80	66 < HCLK ≤ 88	72 < HCLK ≤ 96	90 < HCLK ≤ 120
4 WS (5 CPU cycle)	80 < HCLK ≤ 100	88 < HCLK ≤ 110	96 < HCLK ≤ 120	120 < HCLK ≤ 150
5 WS (6 CPU cycle)	100 < HCLK ≤ 120	110 < HCLK ≤ 132	120 < HCLK ≤ 144	150 < HCLK ≤ 180
6 WS (7 CPU cycle)	120 < HCLK ≤ 140	132 < HCLK ≤ 154	144 < HCLK ≤ 168	-
7 WS (8 CPU cycle)	140 < HCLK ≤ 160	154 < HCLK ≤ 176	168 < HCLK ≤ 180	-
8 WS (9 CPU cycle)	160 < HCLK ≤ 168	176 < HCLK ≤ 180	-	-

²⁴⁵ warto sobie o nich przypomnieć jeśli program modyfikuje pamięć Flash, wszelkie bufore należą wówczas wyczyścić tak aby nie zawierały starych danych

²⁴⁶ zaraz się wyjaśni co to :) w rozdziale 17

²⁴⁷ właściwie to nie wiem czemu SYSCLK a nie HCLK...

Co się stanie jeśli źle to skonfigurujemy? Nie mam pojęcia... może pojawią się przekłamania w odczycie pamięci i procesor będzie durniał?

Ponadto kontroler pamięci Flash pozwala:

- kasować i zapisywać pamięć flash (z poziomu programu)
- odczytywać stan *bajtów konfiguracyjnych*
- modyfikować *bajty konfiguracyjne*

Możliwość modyfikacji pamięci flash można wykorzystać do przechowywania danych nieulotnych. ST wydało notę aplikacyjną opisującą sposób emulowania pamięci EEPROM we Flashu (*AN2594 EEPROM emulation in STM32F10x microcontrollers*). Drugie zastosowanie tej funkcji to modyfikacja kodu programu przez program (np. bootloader).

Interfejs programowania pamięci Flash (FLITF), może być taktowany tylko z wewnętrznego źródła wysokiej częstotliwości (HSI). Z tego względu należy się upewnić, że źródło to jest włączone jeśli w programie korzystamy z tej opcji. Ponadto należy zwrócić szczególną uwagę na układy watchdog. Czas kasowania pamięci flash, może dochodzić do 40ms w przypadku F103/F334 i ponad 30s (!) w przypadku F429 (według datasheet), należy zadbać o to aby w tym czasie nie nastąpiło zadziałanie układu licznika nadzorującego. W czasie kasowania i programowania pamięci, należy również zapewnić stabilne zasilanie mikrokontrolera ze źródła o napięciu minimum:

- 2V dla F103
- od 1,7V do 2,7V (w zależności od szerokości zapisywanych danych) w F429

oraz o odpowiedniej wydajności prądowej (patrz datasheet).

Co warto zapamiętać z tego rozdziału?

- przy zwiększaniu częstotliwości sygnałów zegarowych w mikrokontrolerze (co zaraz zrobimy w rozdziale 17) trzeba pamiętać o ustawieniu opóźnień w dostępie do pamięci flash
- wszelkie buforowania można włączyć i zapomnieć
- interfejs programowania pamięci Flash korzysta z oscylatora HSI (szczegóły w rozdziale 17)

16.2. Tryby uruchamiania i bootloader

Jak zerkniesz do mapy pamięci to zobaczysz, że w przestrzeni adresowej, pamięć flash zaczyna się od adresu 0x0800 0000. No zaraz! Ale przecież rdzeń Cortex-M po resecie systemowym odczytuje wskaźnik stosu i adres kodu programu spod adresów odpowiednio: 0x0 i 0x4 (jeśli tego nie wiedziałeś to znaczy, że za mało czytasz w Internecie o STMachine!). Jak to więc może działać i co jest w tych 128MB „przed” pamięcią flash?

W STMachine zaimplementowano sprzętowy mechanizm, który pozwala wpływać na to, z jakiej pamięci uruchomi się procesor. Działa to w ten sposób, że tuż po resecie lub wybudzeniu z trybu standby (dokładnie 4 cykle zegarowe po) sprawdzany jest stan nóżek mikrokontrolera oznaczonych **BOOT0** i **BOOT1**²⁴⁸. W zależności od stanu tych nóżek, następuje zmapowanie pamięci²⁴⁹ pod adresem 0. Procesor zawsze po resecie zaczyna odczytywać zawartość pamięci od adresu 0. Tym sposobem, w zależności od sposobu mapowania pamięci, rdzeń startuje z innej pamięci.

Tabela 16.3 Tryby uruchamiania mikrokontrolerów

BOOT0	BOOT1 ²⁴⁸	uruchomienie z pamięci	
		F103 i F429	F334
0	x	pamięci flash	pamięci flash
1	0	bootloadera	pamięci SRAM
1	1	pamięci SRAM	bootloadera

Przykładowo jeśli **BOOT0** = 0, to pamięć zostanie zmapowana tak, że odczytując kolejne wartości od adresu 0 będziemy odczytywali wartości z pamięci flash (która fizycznie zaczyna się od adresu 0x0800 0000). To będzie po prostu ta sama zawartość pamięci, dostępna pod różnymi adresami.

Jeśli konfiguracja nóżek będzie inna, np. **BOOT0** = 1 i **BOOT1** = 0 (dotyczy F103 i F429), to odczytując pamięć od adresów 0 procesor będzie dostawał zawartość pamięci *System Memory*, w której znajduje się firmowy bootloader. Co do uruchamiania programu z pamięci SRAM, zapraszam do rozdziału 16.6 :)

Słówko o tym bootloaderze. Bootloader to (w skrócie) program, który poprzez jakiś interfejs komunikacyjny odczytuje nowy wsad dla mikrokontrolera i zapisuje go w pamięci flash. Powódów wykorzystania bootloadera można wymyślić dziesiątki. Przede wszystkim pozwala na podmianę oprogramowania bez specjalistycznych narzędzi (programatora i oprogramowania) czy dostępu do wnętrza urządzenia. Dzięki temu może to zrobić nawet laik (użytkownik produktu), np. wkładając do aparatu cyfrowego kartę pamięci z zapisanym nowym wsadem (aparat sam sobie rozpozna wsad

²⁴⁸ F334 nie posiada nóżki **BOOT1**, zamiast niej odczytywany jest stan bitu konfiguracyjnego n \overline{B} OOT1

²⁴⁹ tak to się fachowa nazywa?

i go „zainstaluje”). Koniec OT, wracamy do STMów! Firmowy bootloader jest wgrywany na etapie produkcji mikrokontrolera i jest nieusuwalny. Bootloader pozwala na wgrywanie wsadu za pomocą interfejsu:

- w F103: USART1
- w F334: USART1, USART2, I2C1
- w F429: USART1, USART3, CAN2 i USB (klasa DFU)

Potrzebny jest tylko specjalny program. Do F103 ST udostępnia program za darmo. Kiedyś był to *Flash Loader Demonstrator*. Teraz nie wiem, nie korzystam, być może coś się zmieniło. Bez problemu znalazłem też program działający na Linuksie (*stm32flash*). Nie wiem jak wygląda sprawa z F334 i F429 bo nigdy nawet nie włączyłem tam bootloadera. W zestawie HY-mini wbudowana jest przejściówka USART1 ↔ USB. I tak się miło składa, że współpracuje ona z interfejsem wykorzystywanym przez bootloader. Wystarczy więc podłączyć płytę przewodem USB do komputera i można wgrywać wsad poprzez bootloader. Ale proszę się nie cieszyć za bardzo. Na dłuższą metę to nie jest wygodne rozwiązańe ze względu na konieczność zabawy z nóżkami BOOT. No i debugować się nie da przez bootloader. Bootloader ma jedną dużą zaletę. Jeśli coś skopiemy w programie na tyle solidnie, że nie będziemy mogli się połączyć z mikrokontrolerem poprzez JTAG czy SWD, np. natychmiastowe usypianie mikrokontrolera po resecie czy wyłączenie interfejsów komunikacyjnych. To można uruchomić procek w trybie bootloadera. Nasz program nie jest wtedy w ogóle wykonywany, więc będziemy mogli połączyć się poprzez JTAG/SWD i zmienić wadliwy program.

Szczegółowe informacje o bootloaderach, protokołach komunikacji itd.:

- AN2606 *STM32 microcontroller system memory boot mode*
- AN3155: *USART protocol used in the STM32TM bootloader*
- AN3154: *CAN protocol used in the STM32 bootloader*
- AN3156: *USB DFU protocol used in the STM32 bootloader*
- AN3262: *Using the over-the-air bootloader with STM32W108 devices*
- AN4221: *I2C protocol used in the STM32 bootloader*
- AN4286: *SPI protocol used in the STM32 bootloader*

Co warto zapamiętać z tego rozdziału:

- bootloader może pomóc jeśli skopiemy program i np. wyłączymy JTAG

16.3. Bajty konfiguracyjne (F103)

Bajty konfiguracyjne (*option bytes*) jakoś nieodparcie kojarzą mi się z *fuse bitami* w AVR. Ale proszę nie bać - F103 generalnie nie da się zablokować na amen. W sumie AVRa też się nie da... mniejsza.

W skład bajtów konfiguracyjnych wchodzą:

- cztery bajty konfigurujące ochronę przed zapisem pamięci flash (WRP0...3)
- jeden bajt konfigurujący ochronę pamięci flash przed odczytem (RDP)
- dwa bajty do dowolnego zastosowania przez użytkownika (DATA0...1)
- jeden bajt ustawień użytkownika (USER)

Bajty WRPx umożliwiają zabezpieczenie pamięci flash przed zapisem. Można je wykorzystać np.:

- do ochrony kodu bootloadera (własnego) przed niechcianym nadpisaniem
- do ochrony danych przechowywanych w pamięci flash przed nadpisaniem

Każdy kolejny bit bajtów WRPx, odpowiada za ochronę dwóch stron pamięci flash. Tzn. że zerowy bit bajtu WRP0 umożliwia włączenie ochrony stron 0 i 1, kolejny bit stron 2 i 3, itd. W sumie cały bajt WRP0 odpowiada za strony 0-15, bajt WRP1 za strony 16-31, WRP2 strony 32-47, WRP3 strony 48-61... Ostatni bit bajtu WRP3 odpowiada za ochronę wszystkich pozostałych stron pamięci (62-255). W kwestii podziału pamięci na strony, odsyłam do dokumentacji. Włączenie ochrony następuje po **skasowaniu** danego bitu.

Bajt RDP odpowiada za ochronę pamięci przed odczytem. Jeżeli ochrona jest włączona:

- odczyt pamięci flash jest możliwy tylko przez kod programu uruchomiony z pamięci flash jeśli dodatkowo do mikrokontrolera nie jest podłączony debugger
- strony pamięci 0 i 1 automatycznie zostają objęte ochroną przed zapisem, reszta pamięci może być kasowana/zapisywana przed kod programu (z wyjątkiem kodu uruchomionego z pamięci SRAM)
- program uruchomiony z pamięci SRAM nie ma dostępu do zawartości pamięci flash (dotyczy również próby odczytania poprzez DMA)
- wszelkiej maści debuggery nie mogą odczytywać pamięci flash

- zdjęcie blokady przed odczytem powoduje (automatycznie) wyczyszczenie zawartości pamięci flash (*mass erase*) żeby uniemożliwić odczytanie programu

Ochrona przed odczytem **nie jest aktywna tylko wtedy**, gdy wartość bajtu RDP wynosi 0xA5.

Bajty *Data0* i *Data1* można dowolnie wykorzystać. Nie wpływają one w żaden sposób na działanie mikrokontrolera. Można do nich wpisać np. numer seryjny urządzenia, jakiś kod do szyfrowania, czy cokolwiek kto sobie wymyśli.

Bajt ustawień użytkownika (USER) odpowiada za następujące opcje:

- automatyczne resetowanie mikrokontrolera przy wejściu w tryb uśpienia (patrz: *low power management reset*, rozdział 11.1) - bity nRST_STOP i nRST_STDBY
- automatyczne włączanie watchdoga niezależnego przy uruchamianiu mikrokontrolera - bit WDG_SW

Warto powiedzieć kilka słów o zasadzie działania bajtów konfiguracyjnych. Są one zapisane w pamięci flash pod adresem 0x1FFF F800. Odczyt zawartości bajtów konfiguracyjnych następuje jednokrotnie podczas *resetu systemowego* mikrokontrolera. W przypadku zmiany ich wartości nowe nastawy zaczynają działać dopiero po zresetowaniu mikrokontrolera. W programie można odczytać wartości konfiguracyjne bezpośrednio z pamięci flash (z podanego trzy linijki wyżej adresu²⁵⁰) lub poprzez rejestrów bloku kontrolera pamięci flash.

Każdy bajt konfiguracyjny, zajmuje w pamięci flash 2B. Podwójna zajętość wynika z tego, że bajty konfiguracyjne zapisywane są podwójnie (masło maślano). Drugi zapis zawiera zanegowaną wartość podstawowego bajtu konfiguracyjnego i jest oznaczony prefiksem *n*. Np. bajt *nUSER* to zanegowana wartość bajtu konfiguracyjnego *USER*. Mechanizm ma na celu wykrywanie błędnych wartości konfiguracyjnych. Jeśli wartość podstawowa i zanegowana nie będą do siebie pasować, to wartości odczytane z pamięci flash zostaną zignorowane. W takim wypadku zamiast wartości odczytanej z pamięci flash, ładowana jest wartość konfiguracyjna 0xFF, ponadto ustawiana jest flaga OPTERR w rejestrze FLASH_OBR.

Zmiana zawartości bajtów konfiguracyjnych może być wykonana z poziomu komputera PC poprzez odpowiednie oprogramowanie (np. STLink Utility lub OpenOCD) lub z poziomu aplikacji. W przypadku drugiej metody, konieczne jest wykonanie szeregu upierdliwych czynności zabezpieczających wartości bajtów konfiguracyjnych przed przypadkową zmianą. Najlepiej, będzie to widoczne na przykładzie.

250 w pliku nagłówkowym są odpowiednie definicje (OB)

Zadanie domowe 16.1: napisać program, który sprawdza czy włączona jest funkcja resetowania mikrokontrolera przy wejściu w tryb uśpienia *stop mode*. Jeżeli nie, to program powinien włączać tą funkcję i sygnalizować to zapaleniem diody. Następnie program powinien (jeden raz) mignąć inną diodą i wprowadzić mikrokontroler w tryb uśpienia *stop mode*.

Przykładowe rozwiązanie (F103, diody na PB0 i PB1):

```

1. int main(void) {
2.
3.     RCC->APB2ENR = RCC_APB2ENR_IOPBEN;
4.     gpio_pin_cfg(GPIOB, PB0, gpio_mode_output_PP_2MHz);
5.     gpio_pin_cfg(GPIOB, PB1, gpio_mode_output_PP_2MHz);
6.
7.     if (FLASH->OBR & FLASH_OBR_nRST_STOP){
8.         uint16_t tmp_user = OB->USER;
9.         tmp_user &= ~(1<<1);
10.        GPIOB->ODR |= PB1;
11.
12.        RCC->AHBENR |= RCC_AHBENR_FLITFEN;
13.        while(FLASH->SR & FLASH_SR_BSY);
14.
15.        FLASH->KEYR = 0x45670123;
16.        FLASH->KEYR = 0xcdef89ab;
17.        while(FLASH->CR & FLASH_CR_LOCK);
18.
19.        FLASH->OPTKEYR = 0x45670123;
20.        FLASH->OPTKEYR = 0xcdef89ab;
21.        while(!(FLASH->CR & FLASH_CR_OPTWRE));
22.
23.        FLASH->CR |= FLASH_CR_OPTER;
24.        FLASH->CR |= FLASH_CR_STRT;
25.        while(FLASH->SR & FLASH_SR_BSY);
26.
27.        FLASH->CR &= ~FLASH_CR_OPTER;
28.        FLASH->CR |= FLASH_CR_OPTPG;
29.
30.        OB->USER = (uint16_t)tmp_user & 0xff;
31.        while(FLASH->SR & FLASH_SR_BSY);
32.
33.        OB->RDP = (uint16_t)0x00a5;
34.        while(FLASH->SR & FLASH_SR_BSY);
35.
36.        FLASH->CR &= ~FLASH_CR_OPTPG;
37.        FLASH->CR &= ~FLASH_CR_OPTWRE;
38.        FLASH->CR |= FLASH_CR_LOCK;
39.
40.    }
41.
42.    GPIOB->ODR |= PB0;
43.    for(volatile uint32_t delay = 500000; delay; delay--){};
44.    GPIOB->ODR &= ~PB0;
45.    for(volatile uint32_t delay = 100000; delay; delay--){};
46.
47.    SCB->SCR |= SCB_SCR_SLEEPDEEP_Msk;
48.    __WFI();
49.
50.    while (1){};
51.
52. } /* main */

```

7) odczytujemy wartość bitu konfiguracyjnego poprzez rejesty kontrolera pamięci flash (rejestr FLASH_OBR) i sprawdzamy czy jest włączona funkcja RST_STOP. Funkcja jest włączona jeśli związany z nią bit jest skasowany. Warunek będzie prawdziwy, jeśli funkcja nie jest włączona.

8) odczytuję wartość bajtu konfiguracyjnego USER do zmiennej pomocniczej. Tym razem (dla urozmaicenia) odczytuję wartość bezpośrednio z pamięci flash. Zwróć uwagę na wielkość zmiennej. Bajt USER ma 8b, ale OB_USER obejmuje wartość bajtu USER oraz wartość komplementarną (zanegowaną). Stąd rozmiar 16b.

9) kasuję pierwszy bit (nRST_STOP), co odpowiada włączeniu funkcji

10) zapalam diodę

12, 13) włączam taktowanie bloku programowania pamięci flash (FLITF), w następnej linijce sprawdzam czy nie trwa poprzednia operacja zapisy pamięci (flaga BSY)... oczywiście, że nie trwa bo to początek programu, ale kultura zobowiązuje :)

15 - 17) odblokowuję dostęp do rejestru FLAH_CR umożliwiającego kasowanie i programowanie pamięci. Wymaga to wpisania dwóch magicznych wartości kluczowych do rejestru FLASH_KEYR. Na końcu sprawdzam czy operacja zakończyła się powodzeniem (skasowanie flagi LOCK).

19 - 21) programowanie bitów konfiguracyjnych wymaga osobnego odblokowania

23 - 25) przed programowaniem nowych wartości, należy skasować pamięć (jak to flash). W tym celu ustawiam bit odpowiedzialny za kasowanie bajtów konfiguracyjnych (OPTER) i rozpoczynam kasowanie (STRT). Czekam na zakończenie operacji (flaga BSY). Jeżeli pamięć nie zostanie skasowana, to zapis nowych wartości nie zostanie przeprowadzony. Uwaga! Ta operacja spowoduje skasowanie (tzn. nadanie im wartości 0xFF) **wszystkich** bajtów konfiguracyjnych.

27, 28) kasuję bit powodujący kasowanie pamięci i ustawiam bit (OPTPG) włączający zapisywanie wartości konfiguracyjnych

30, 31) zapisuję nową wartość bajtu konfiguracyjnego USER. Dwie uwagi. Primo: zapis do pamięci flash musi być 16b. Secundo: kontroler programowania pamięci flash bierze pod uwagę tylko dolną połowę zapisywanej wartości. Górnego bajtu (bajt komplementarny) jest automatycznie obliczany przez kontroler. Czekam na zakończenie zapisu (flaga BSY).

33, 34) przypominam, że skasowaniu uległy wszystkie bajty. Zapisuję wartość bajtu RDP odpowiadającą za wyłączenie zabezpieczenia przed odczytem pamięci flash. Pozostałe bajty konfiguracyjne (np. WRP), zostawiam bez zmian (po kasowaniu mają wartość 0xFF). Czekam na zakończenie zapisu (flaga BSY).

36 - 38) wyłączam bit odpowiedzialny za programowanie bajtów konfiguracyjnych i blokuję dostęp do rejestru FLASH_CR

42 - 45) mignięcie diodą

47, 48) uśpienie mikrokontrolera (stop mode)

Uff. Wyszło długaśnie. Na szczęście bajty konfiguracyjne zmieniają się raczej rzadko. I zdecydowanie szybciej jest robić to z poziomu oprogramowania na PC. A jak wygląda efekt

działania programu? Zakładając, że funkcja resetowania mikrokontrolera przy próbie uśpienia nie była włączona:

- przy pierwszym uruchomieniu programu, warunek z linii 7 jest prawdziwy
- zapala się dioda na PB1
- kasowany jest bit odpowiedzialny za włączanie funkcji resetowania, ale funkcja nie jest jeszcze aktywna! przypominam, że bajty konfiguracyjne są odczytywane tylko raz, przy resecie mikrokontrolera
- program wykonuje „mignięcie diodą” i zasypia aż do resetu
- po resecie (drugie uruchomienie programu) warunek z linii 7 nie jest już prawdziwy
- program przeskakuje programowanie bajtów konfiguracyjnych i wykonuje mignięcie diodą
- na końcu następuje próba uśpienia mikrokontrolera, która kończy się jego zresetowaniem
- w efekcie można zaobserwować miganie diody

Szczegółowy opis programowania pamięci flash, w tym bajtów konfiguracyjnych, można znaleźć w dokumencie: PM0075 *STM32F10xxx Flash memory microcontrollers*.

Co warto zapamiętać z tego rozdziału:

- zmiana bajtów konfiguracyjnych jest najwygodniejsza z poziomu komputera PC
- bajty konfiguracyjne umożliwiają zabezpieczenie pamięci flash przed odczytem i zapisem oraz wymuszenie włączania IWDG

16.4. Bajty konfiguracyjne (F429)

Opis bajtów konfiguracyjnych tego mikrokontrolera można znaleźć w RMie. I tam odsyłam zainteresowanych. Mnie się nie chce dokładnie omawiać, bo to jest nad wyraz nudne. Pozwolę sobie tylko wypunktować kilka najważniejszych nowości:

- przybył nowy bit konfiguracyjny (w USER option bytes) odpowiedzialny za układ BOR
- ochrona pamięci przed odczytem (RDP) podzielona jest na trzy stopnie:
 - level 0 - *no protection* - brak ochrony

- level 1 - *read protection* - nie jest możliwy dostęp (odczyt, zapis, kasowanie) do pamięci flash i backup SRAM jeżeli jest podłączony debugger, uruchomiono program z pamięci SRAM lub uruchomiono firmowy bootloader. Zmniejszenie poziomu ochrony na 0 powoduje całkowite wykasowanie pamięci flash i backup SRAM
- level 2 - *chip protection* - tak jak na poziomie 1 plus dodatkowo: nie jest możliwe uruchomienie mikrokontrolera z pamięci SRAM lub z bootloadera; interfejsy JTAG, SWD, ETM zostają zablokowane; nie jest możliwa zmiana *user option bytes*; ten poziom ochrony jest **nieodwracalny** - nie jest możliwe cofnięcie do niższego levela! To jest jedyny, znany mi ficzer, którym można sobie na amen w pacierzu „zablokować” mikrokontroler F429. Tylko Rosjanie potrafią to cofnąć :)

Co warto zapamiętać z tego rozdziału:

- włączenie drugiego stopnia ochrony (RDP) pamięci, w mikrokontrolerze F429 jest nieodwracalne!

16.5. Bajty konfiguracyjne (F334)

Bajty konfiguracyjne mikrokontrolera F334 są zbliżone do tych z F429. Szczegółowy opis można znaleźć w RMie. Podobnie jak w przypadku F429, mikrokontroler F334 można „zablokować” na amen w pacierzy włączając mu drugi poziom ochrony pamięci. Na szczęście raczej trudno zrobić taki numer przypadkowo - to nie AVR :) Procedurka jest dosyć skomplikowana - odsyłam do przykładowych programów modyfikujących bajty konfiguracyjne w F103 (rozdział 16.3). A tak poza tym to nic tu po nas, są ciekawsze rzeczy.

Co warto zapamiętać z tego rozdziału:

- włączenie drugiego stopnia ochrony (RDP) pamięci, w mikrokontrolerze F344 jest nieodwracalne!

16.6. Kod w pamięci SRAM - po co?

W AVRach podział pamięci na flash, sram i pamięć układów peryferyjnych był, że tak to nazwę „sztywny, wyraźny... i czasem nieco frustrujący”. W STMacie, jak być może zdążyłeś się już

przekonać, granice te są mniej wyraziste. Czasem się wręcz zacierają... no przynajmniej ja tak to odbieram. Jedną z oznak owego zacierania jest dla mnie to, że rdzeń może wykonywać kod programu zapisany w pamięci sram mikrokontrolera. Na dodatek dotyczy to nie tylko wbudowanej pamięci operacyjnej. Zainteresowanych odsyłam do programming manuala rdzenia Cortex: rozdział *Memory model*, tabela *Memory access behavior*.

Podstawowe pytanie brzmi: a po kielę nam możliwość wykonywania kodu z pamięci, która traci zawartość po każdym resecie!? Zgadzasz się z tym pytaniem? Zastanów się dobrze zanim odpowiesz! Oczywiście pytanie jest perfidnie podchwytliwe i zawiera kardynalny błąd - pamięć sram nie traci zawartości przy byle resecie mikrokontrolera, dopiero przy zniku zasilania :)] Tak czy siak nie jest to pamięć „trwała”. No więc po co sobie w ogóle zwracać tym głowę? Najczęściej przewijające się, tu i tam, powody kombinowania z umieszczeniem kodu programu w pamięci operacyjnej to:

- chęć przedłużenia życia pamięci flash
- chęć skrócenia czasu wykonywania fragmentów kodu (np. skomplikowanych obliczeń)
- chęć aktualizacji kodu bootloadera lub umożliwienia wykonywania programu w czasie kasowania/programowania pamięci flash

No i oczywiście są też kosmiczne pomysły - np. system operacyjny, który umożliwi uruchamianie programów dynamicznie ładowanych z karty pamięci itp. Na razie pozostańmy jednak na ziemi i rozprawmy się z trzema powyższymi kropkami :)

Nie ma co ukrywać, że pamięć flash ma ograniczoną ilość zapisów. Dokładniej rzecz ujmując, za degradację pamięci odpowiada nie tyle zapis co kasowanie. Kasowanie jest wymagane przed zapisem, więc specjalnie nas to nie ratuje. Nie jeden programista mikrokontrolerowy zastanawiał się pewnie, czy nie warto byłoby rozwijać i debugować programów w pamięci sram²⁵¹ aby nie nabijać flashowi „licznika kasowań”? AVRRowcom też czasem takie pomysły wpadały do głowy, ale dosyć rzadko... no i tam sprawa była prosta, albowiem AVR może wykonywać kod tylko z pamięci flash więc nie ma nad czym dywagować. Obawy związane z możliwością wykończenia pamięci szczególnie często pojawiają się wśród początkujących (jak ja), którzy „czasem” zamiast przysiąć nad napotkanym problemem i rozwiązać go „analytycznie”, stosują metodę „brutal force” - czyli zmieniają coś w kodzie na chybiił trafił i szukają takiego ustalenia żeby program zaczął działać. Ubocznym skutkiem tej metody jest „pierdylion” programowań - po każdej malej zmianie w programie. Swoją drogą, żeby nie było, nie ma co ukrywać, że empiryczne sprawdzenie

²⁵¹ żeby nie było wątpliwości: pamięć sram się nie „zużywa” przy programowaniu

kilku wariantów konfiguracji może okazać się o wiele szybsze niż analiza „dzieł zebranych” zwanych dokumentacją²⁵². Ale wróćmy do głównego tematu, bo trochę odpłyneliśmy :)

Szybki skok do datasheeta (*Electrical characteristics → Operating conditions → Memory characteristics*) i już wiemy, że producent szacuje trwałość pamięci flash na minimum 10 tysięcy cykli kasowania/programowania. Zanim spróbujemy sobie odpowiedzieć czy to dużo czy mało, pomyślmy co ta liczba tak naprawdę oznacza? Dziesięć tysięcy programowań i nagle trach! Krew, pot, zły, niedowierzanie i uszkodzony mikrokontroler? Nie, to nie działa jak toner do drukarki z licznikiem, który po wydrukowaniu x stron „skończy się” nawet jeśli wszystkie wydrukowane kartki były puste. Nie ma tu żadnego licznika programowań (aż dziwne). Ta liczba (10 000) to maksymalna ilość programowań, dla której producent gwarantuje, że w określonych w datasheetie warunkach (dotyczy np. zakresu temperatur otoczenia) pamięć zachowią swoją zawartość przez podany w dokumentacji okres (np. 20 lat). Nie jest to dokładna wartość! Na pewno jest zaniżona i mikrokontroler wytrzyma dużo więcej. To jak z pastą do zębów, która „może zawierać śladowe ilości orzechów arachidowych”. Szansa jest niewielka, ale producent woli umieścić ostrzeżenie, żeby potem żaden alergik go po sądach nie ciągał.

No ale dobra, niech będzie... przyjmijmy że trzymamy się sztywno tej granicy 10 000 programowań. Ile razy dziennie jesteś w stanie zaprogramować mikrokontroler? Sto? No bez przesady... to wychodzi prawie jedno programowanie co 5 min przez 10 godzin. Nierealna wartość, większy projekt może się dłużej kompilować. A nawet gdyby się udało, to ile dni takiej zabawy wytrzymasz? Bo dojście do „ustawowego” limitu pamięci zajmie Ci, w tym tempie, ponad trzy miesiące.

Zejdźmy na ziemię :) Założmy cztery godziny zabawy dziennie, flashowanie co 15min, 5 dni w tygodniu - może być? Wydaje mi się, że to dosyć „pesymistyczne założenie”. Mnie po kilku tygodniach zabawy mikrokontrolerowej zaczyna „nosić” i muszą sobie zrobić przerwę na trochę. Tak czy inaczej, z tych założeń wychodzi 80 programowań w tygodniu. Czyli około 320 na miesiąc. „Limit” pamięci wyczerpie się po ca 31 miesiącach. Prawie trzy lata. Trzy lata programowania pamięci co 15min, cztery godziny dziennie bez weekendów... Wydaje mi się, że przedtem uszkodzi się płytę z STMem robiąc przypadkowe zwarcie niż zarzynając pamięć.

Jeżeli kogoś to jeszcze nie przekonuje, to niech się zastanowi co się stanie, gdy przekroczy limit programowań? Otóż w miarę „zużywania” się flasha, maleje czas przez jaki pamięć będzie przechowywała dane. W datasheetie jest mowa (z tego co pamiętam) o 20 latach w zakresie temperatur coś koło -40 - 85°C. Nieśmiało przypuszczam, że zakres zmian temperatur w miejscu gdzie programujesz jest nieco węższy? No i te 20 lat... skoro mówimy o tysiącach programowań to

252 swoją drogą: do wypróbowania różnych konfiguracji można wykorzystać debugger, tzn. zmieniać wartości rejestrów konfiguracyjnych „w locie”, bez konieczności ponownego programowania mikrokontrolera

nie jest to mikrokontroler użyty w „konkretnym” urządzeniu które ma działać latami, tylko jakiś układ „naukowy/prototypowy/rozwojowy”. Raczej nie ma potrzeby, aby trzymał program przez 20 lat, prawda? Szczególnie, że przed chwilą ustaliliśmy, że programujemy go prawie codziennie. Zmierzam do tego, że jeśli nie zależy nam na utrzymaniu tych katalogowych parametrów, to pewnie możemy zaprogramować flash kilkudziesiąt tysięcy razy i nic złego nas nie spotka. A takie 50 000 programowań to około rok codziennego programowania co 10min! Całą dobę, bez przerwy! Także ten...

Swoją drogą, popularne Atmelki też mają podany limit 10 000 programowań pamięci flash. Tak szczerze, z ręką na sercu, znasz kogokolwiek kto „zabił” flash w mikrokontrolerze? Ale tak na 100% - nie chodzi mi o jakieś legendy o koledze brata kuzyna sąsiada czy jakieś inne forumowe prawdy objawione w stylu „*zmienilem mikrokontroler i działa, pewnie mikrokontroler się zużył bo używam go ostro od miesiąca i w ogóle ostatnio dłużej się programował i często zawieszal*”. Ja nie znam takiego przypadku. A naprawdę sporo czasu spędzam na czytaniu for (forów?) i Internetu :)

Druga kropka „za” wykonywaniem kodu z ramu to była szybkość. Że niby wykonanie kodu umieszczonego w pamięci sram zajmie mniej czasu niż w przypadku tego samego kodu w pamięci flash, bo dostęp do pamięci sram jest szybszy. Generalnie sprawa wygląda tak, że w mikrokontrolerach z rdzeniami Cortex-M3, Cortex-M4, Cortex-M7²⁵³ jest wręcz odwrotnie. Program uruchomiony z pamięci flash działa szybciej. Chyba zgodzisz się ze mną, że biorąc pod uwagę iż pamięć flash działa wolniej niż sram, wydaje się to dosyć niecodzienne. Wymienione rdzenie opierają się o jakąś pochodną architektury Harvardzkiej. Oznacza to, że CPU może jednocześnie korzystać z pamięci programu (flash) i danych (sram). Jeżeli umieścimy zarówno kod jak i dane w jednej pamięci (sram) to o tej równoległości nie może być mowy (architektura von Neumanna). Stąd spadek wydajności. Oczywiście można się doktryzować: o ile pamięć flash jest wolniejsza od sramu, co będzie jak dojdą opóźnienia (waitstates - patrz rozdział 16.1), co jeśli program będzie rzadko odwoływał się do pamięci, itd. itd. Generalnie jednak wniosek do zapamiętania jest taki, że umieszczenie funkcji w pamięci operacyjnej nie przyspiesza jej wykonania. Koniec i klopka.

Dla niedowiarków wykonałem krótki test. Program testowy został napisany na kolanie, bez rozmyślań o dostępcach do pamięci, waitstatesach, optymalizacji itd.:

253 nie dotyczy Cortex-M0 i M0+ jako że to inna architektura (von Neumann)

Program testowy:

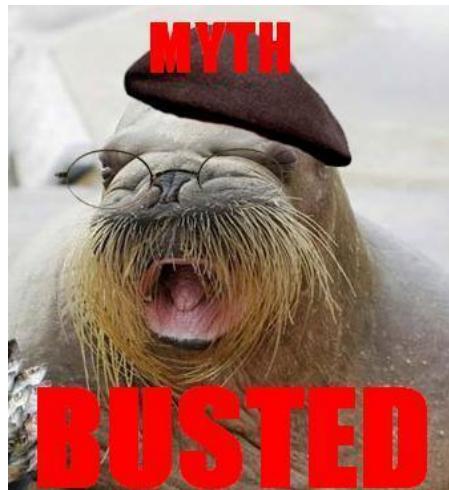
```
1. const double data[] = {1.3f, 1.4f, 1.6f, 1.7f, 1.8f};  
2. volatile double a,b,c;  
3. uint32_t i;  
4.  
5. for(i=0; i< 1000; i++){  
6.     a = log(b*c*0.5f);  
7.     b = sin(c * atan( pow(a, b) ) );  
8.     c = a/b;  
9.     a = data[(uint32_t)b%5] + c;  
10. }
```

Za pomocą funkcji opisanych przy okazji testowania bit bandingu (patrz dodatek 4) zmierzyłem czas wykonywania tego kodu z pamięci flash i sram. Mikrokontrolery pracowały z domyślną („po-resetową”) konfiguracją układów RCC i kontrolera flash²⁵⁴. Wyniki w poniższym array'u:

Tabela 16.4 Liczba cykli procesora przy wykonywaniu funkcji testowej z pamięci flash i sram

rodzaj pamięci	Cortex-M3	Cortex-M4 (bez FPU)
flash	758 007	759 010
sram	1 049 007	993 007

Także ten... podsumujmy teorię o szybkich funkcjach z ramu w formie rysunkowej :)



Rys. 16.1. Mit obalony (źródło: <http://i.imgur.com/hvPZPWR.jpg>)

Została jeszcze jedna kropka do omówienia. Dotyczy bootloaderów i pracy mikrokontrolera podczas programowania pamięci flash. I to jest jak dotąd jedyny sensowny powód aby wrzucić program do pamięci sram. W jednym z poprzednich rozdziałów wspominałem o tym, że kasowanie/programowanie pamięci flash jest dosyć czasochłonnym procesem (np. kasowanie

²⁵⁴ jak nie zapomnę to w wolnej chwili porobię testy z różnymi ustawieniami i jeśli coś ciekawego z tego wyjdzie to dorzucę :)

strony pamięci w F103 trwa do 40ms; kasowanie całej pamięci w F429 typowo trwa 7s!). Procesor nie jest wstrzymywany na ten czas, ale nie może odczytywać zawartości pamięci flash. Jeżeli nie możemy sobie pozwolić na wstrzymanie programu uruchomionego z flasha, bo np. korzystamy z interfejsu USB i nie możemy zerwać połączenia, to uratować może nas przeniesienie programu do pamięci sram.

Co warto zapamiętać z tego rozdziału?

- szansa na to, że wykończysz pamięć flash jest minimalna i nie warto się tym przejmować
- kod z pamięci sram generalnie nie wykonuje się szybciej niż z pamięci flash
- program z pamięci sram może się wykonywać w czasie programowania pamięci flash

16.7. Funkcja w pamięci sram - jak?

Skoro już z grubsza wiemy do czego może nam się przydać kod programu w pamięci sram, zastanówmy się nad drugą stroną kija - jak to zrealizować w praktyce.

Uwaga! Większość rozwiązań pokazanych w tym rozdziale (i następnym) jest specyficzna dla wykorzystywanych narzędzi (gcc, przykładowy projekt z www.freddiechopin.info). Jeżeli wykorzystujesz inne narzędzia to musisz przeprowadzić pokazane metody tak, aby działały z twoimi zabawkami :)

Uwaga dwa! Wydaje mi się, że ten (i następny) rozdział trochę wystaje poza obrys poradnika „dla początkujących”. Znaczna część zagadnień poruszanych w tych rozdziałach dotyczy bardziej wykorzystywanych narzędzi (gcc) niż samego STMa. Narzędzia nie są przedmiotem Poradnika, więc specjalnie się nad nimi nie rozwodziłem. Generalnie zmierzam do tego, że jeśli to Twój pierwszy kontakt z STMami to proponuję ten rozdział (i następny) zostawić sobie na deser. Zabawa w uruchamianie programów z pamięci sram jest szalenia ekscytująca (na pewno bardziej niż taki np. RTC :]) i nie będę ukrywał, że świetnie się bawiłem przy tym rozdziale, ale przypuszczam że w praktyce do niczego się ta wiedza na razie nie przyda :) Także spokojnie możesz ten rozdział „*step over*”, ewentualnie „*step into bez parcia na przyswojenie wszystkiego*” :)

Jak już wiemy, pamięć sram traci zawartość przy zaniku zasilania. Przy pierwszym uruchomieniu mikrokontrolera, po włączeniu zasilania, zawiera więc losowe śmieci. Rozwiązania pozwalające umieścić kod programu w sramie są dwa: albo zrobimy to „z komputera” (wgramy program analogicznie jak przy zwykłym flashowaniu tyle że do innej pamięci²⁵⁵), albo zrobi to kod

²⁵⁵ skoro wgrywanie programu do pamięci flash bywa nazywane „flashowaniem”, to wgrywanie do pamięci sram będzie „sramowaniem” mikrokontrolera? -_-

programu uruchomionego na mikrokontrolerze z pamięci flash. Np. własna funkcja kopiąca kawałek flasha do sramu. Można też wykorzystać już istniejące mechanizmy :} Jesteśmy leniwi, więc wybieramy bramkę numer dwa.

Pytanie za 100 punktów: z czym Ci się kojarzy kopiowanie fragmentu pamięci flash do sram? Oczywiście chodzi o sekcję *.data* i inicjalizację zmiennych w startupie²⁵⁶. Skoro program może kopować, w procedurze „rozbiegowej”, wartości zmiennych z flash do sram... to czemu nie mógłby też skopiować kodu jakiejś funkcji!? Przecież to takie same dane jak każde inne. Wystarczy tylko zbałamucić kompilator i wmówić mu, że funkcja jest „daną” a sam odwali resztę roboty :>

Specjalnie, na potrzeby nadchodzących zabaw, stworzyłem dzieło programistyczne - taki program „bazowy”. Przeanalizuj sobie jego działanie:

Program bazowy do dalszych zabaw (F103):

```
1. void fun1(void){  
2.     GPIOB->ODR ^= PB0;  
3. }  
4.  
5. void fun2(void){  
6.     GPIOB->ODR ^= PB1;  
7. }  
8.  
9. int main(void){  
10.    RCC->APB2ENR = RCC_APB2ENR_IOPBEN;  
11.  
12.    gpio_pin_cfg(GPIOB, PB0, gpio_mode_output_PP_2MHz);  
13.    gpio_pin_cfg(GPIOB, PB1, gpio_mode_output_PP_2MHz);  
14.  
15.    volatile uint32_t delay;  
16.  
17.    while(1){  
18.        fun1();  
19.        fun2();  
20.        for(delay = 500000; delay; delay--){};  
21.    } /* while(1) */  
22.  
23. } /* main */
```

Zadanie domowe 16.2: przerobić program bazowy tak aby funkcja *fun1()*, udając zmienną inicjalizowaną (z sekcji *.data*), wylądowała w pamięci sram mikrokontrolera

256 w razie potrzeby do doczytania we własnym zakresie

Przykładowe rozwiązań (F103, gcc):

```
1. void fun1(void) __attribute__((section(".data")));
2. void fun1(void){
3.     GPIOB->ODR ^= PB0;
4. }
5.
6. void fun2(void){
7.     GPIOB->ODR ^= PB1;
8. }
9.
10. int main(void){
11.     RCC->APB2ENR = RCC_APB2ENR_IOPBEN;
12.
13.     gpio_pin_cfg(GPIOB, PB0, gpio_mode_output_PP_2MHz);
14.     gpio_pin_cfg(GPIOB, PB1, gpio_mode_output_PP_2MHz);
15.
16.     volatile uint32_t delay;
17.
18.     while(1){
19.         fun1();
20.         fun2();
21.         for(delay = 500000; delay; delay--){};
22.     } /* while(1) */
23.
24. } /* main */
```

Jedynie co się zmieniło, to atrybut nadany funkcji (*section*). Powoduje on, że jest ona umieszczana w sekcji *.data* i, tak jak wszystkie inne obiekty z tej sekcji, jest ładowana do pamięci operacyjnej przy „rozruchu” mikrokontrolera. Dla pewności jeszcze wycinek z pliku *.dmp* (ewentualnie z tego co wypluwa program *nm*):

```
fun1          |20000400|   D  | FUNC|00000010|      | .data
```

W skrócie: obiekt *fun1* znajduje się pod adresem 0x2000 0400 (sram!), ma rozmiar 0x10B, znajduje się w sekcji *.data*. Banalne prawda :) Prawie! Jeśli przyjrzymy się liście symboli dokładniej, to zauważymy, że pojawił się jeszcze jeden nowy twór, który powinien nas zainteresować:

```
__fun1_veneer |08000220|   t  | FUNC|00000010|      | .text
```

Ki czort? Jakaś nowa, pomocnicza²⁵⁷, funkcja w pamięci flash. Zakładam, że frapuje Cię ona równie mocno jak i mnie! Zobaczmy co to jest na listingu assemblera. Szukamy kawałka programu związanego z wywołaniem funkcji *fun1()*:

257 *veneer* - z ang. fornir, okleina, pokost

```

1. fun1();
2. 80001ea: f000 f819 bl 8000220 <__fun1_veneer>
3. ...
4. 08000220 <__fun1_veneer>:
5. 8000220: b401 push {r0}
6. 8000222: 4802 ldr r0, [pc, #8] ; (800022c <__fun1_veneer+0xc>)
7. 8000224: 4684 mov ip, r0
8. 8000226: bc01 pop {r0}
9. 8000228: 4760 bx ip
10. 800022a: bf00 nop
11. 800022c: 20000401 .word 0x20000401

```

W pierwszej linijce jest wywołanie funkcji *fun1()*. Kolejna linijka zawiera wygenerowany „kompilat” odpowiadający temu wywołaniu. Jak widać, zamiast *fun1*, wołana jest nasza intrygująca okleina. A w niej dzieje się co następuje (linijki 5 - 11):

- rejestr r0 jest odkładany na stosie (żeby zapamiętać jego wartość, bo zaraz będzie modyfikowany)
- do rejestrów r0 ładowana jest wartość spod adresu PC+8 (0x0800 022c), czyli wartość 0x2000 0401
- wartość rejestrów r0 przenoszona jest do rejestrów *ip*
- stara wartość rejestrów r0 jest przywracana ze stosu
- wykonywany jest skok pod adres z rejestrów *ip*
- *nop* (z linii 10) nie robi nic mądrego, wyrównuje adres do 4B

Nasza funkcja *fornirowa* wykonała skok pod adres 0x2000 0401. Na pewno skojarzyłeś ten adres, z adresem w pamięci sram gdzie ulokowana została funkcja *fun1* (0x2000 0400). Jedynka na końcu jednego z adresów jest nieistotna, adres zawsze jest parzysty. Różnica jest związana z trybem pracy rdzenia, nie będziemy się w to zagłębiać (jeśli ktoś jest ciekawy to niech szuka haseł: *thumb state, arm state*). Czyli podsumowując sprawę: zamiast skoczyć bezpośrednio do funkcji w pamięci sram, program woła funkcję pośredniczącą, która wykonuje skok do funkcji w pamięci sram... Chyba naturalnym jest, że w tym momencie pojawia się pytanie: po kiego [...] cały ten cyrk i co to u diabła jest rejestr *ip*? I to jest bardzo dobre pytanie, musiałem trochę pogoogleać :]

Sprawa wygląda tak, że w „zwyczajnym” programie, do wykonywania skoków do funkcji wykorzystywany jest rozkaz *b* lub *bl*. Mają one jednak ograniczony zakres skoku do (w uproszczeniu) ±16 MB od aktualnego adresu. Skok z pamięci flash do sram jest „nieco” odleglejszy. Np. w naszym przypadku wywołanie funkcji *fun1()* znajduje się pod adresem 0x0800 01ea, zaś sama funkcja w pamięci sram - 0x2000 0400. Stąd też należy wykonać skok o

trochę ponad 384 MB. W takiej sytuacji należy wykorzystać inne rozkazy skoku - *bx*, *blx*. Szkopuł polega na tym, że w momencie generowania kodu (w procesie kompilacji) odpowiedzialny za tłumaczenie „C → asm” program (kompilator) nie wie, pod jakimi adresami znajdą się nasze funkcje.

Kompilator operuje symbolami, za rozmieszczenie ich w konkretnych miejscach pamięci odpowiada inny program - linker/konsolidator. Podczas linkowania zauważa on, że ten nasz skok do *fun1()* będzie dosyć daleki i wprowadza małą poprawkę do kodu. Linker generalnie nie jest stworzony do modyfikowania kodu i nie jest tak mądry jak kompilator. Dodaje prościutką funkcję pośredniczącą (nasz *veneer*), która jest wywoływana za każdym razem gdy skaczemy do *fun1()*. Ta短短ka funkcja pośrednicząca generuje właściwy „daleki” skok do funkcji w sram. Prawda, że proste? To jeszcze mały cytacik²⁵⁸:

“Machine-level B and BL instructions have restricted ranges from the address of the current instruction. However, you can use these instructions even if label is out of range. Often you do not know where the linker places label. When necessary, the linker adds code to enable longer branches. The added code is called a veneer.”

To jeszcze został nam ten rejestr *ip* do wyjaśnienia. Krótko i na temat: to po prostu inna (symboliczna) nazwa rejestru ogólnego *r12*. Zgodnie ze standardem wywoywania funkcji (*Procedure Call Standard for the ARM Architecture*) linker, w dodawanych przez siebie mini-funkcjach, może posługiwać się rejestrem *ip* i nie musi przywracać jego wartości.

Jeżeli komuś nie podoba się generowanie dodatkowych funkcji przez linker, to może podpowiedzieć kompilatorowi, że skok do danej funkcji będzie daleki. Służy do tego atrybut funkcji *long_call*. Mówią kompilatorowi, że wszelkie wołania tej funkcji będą „dalekie”. Dzięki temu, już na etapie kompilacji, używany jest właściwy rozkaz skoku:

```
1. fun1();
2. 80001f6: 4d0c          ldr    r5, [pc, #48]      ; (8000228 <main+0x60>)
3. ...
4. 80001fa: 47a8          blx    r5
5. ...
6. 8000228: 20000401     .word 0x20000401
```

Idąc jeszcze jeden krok dalej, można dodać do wywołania kompilatora opcję *mlong-calls*, która spowoduje, że wszystkie wywołania funkcji w programie będą dalekie. Kosztem wydajności.

258 źródło: <http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.dui0489e/Cihfddaf.html>

Niestety to nie koniec kłopotów. Problem z dalekim skokiem nie dotyczy tylko drogi *flash* → *sram*. Jeżeli w funkcji działającej w *sramie* (np. w *fun1()*) spróbujemy zwołać funkcję z pamięci *flash* (np. *fun2()*) to program się nawet nie skompiluje. Zamiast tego nakrzyczy na nas paskudnym błędem:

```
relocation truncated to fit: R_ARM_THM_CALL against symbol `fun2' defined in  
.text.fun2 section in out/main.o
```

Szczerze powiedziawszy nie do końca wiem, dlaczego tym razem linker nie dodał automatycznie funkcji pośredniczących. Przypuszczam, że nie potraktował obiektów w sekcji *.data* jako kodu programu (tylko jako dane)²⁵⁹ i dlatego nie zatroszczył się o nasze skoki. Potwierdzeniem mojej teorii może być to, że jeśli zmienimy nazwę sekcji do której wysyłamy *fun1* na np. *.data.code*, to następuje cudowne uzdrawienie i program linkuje się już bez błędu. Generowane są dwie funkcje pośredniczące: przy skoku z flash do ram (wywołanie *fun1* z *main*) i przy skoku z ram do flash (wywołanie *fun2* z *fun1*). Wadą takiego rozwiązania jest niestety to, że program *size* (chodzi o program, który po skończonej komplikacji wypisuje rozmiary sekcji *.text*, *.data*, *.bss*) zgłupiał i nie zalicza naszej funkcji do sekcji *.data*²⁶⁰.

Na powyższy błąd pomaga atrybut *long_call* (dodany do funkcji *fun2()* lub jako opcja wywołania kompilatora). Uwaga! Problem dalekich skoków dotyczy nie tylko „jawnych” wywołań funkcji w kodzie. Jeżeli kod umieszczony w pamięci *sram* zacznie np. operować na liczbach 64 bitowych i będzie konieczne wywołanie jakiejś funkcji matematycznej (te takie ze śmiesznymi nazwami zaczynającymi się od dwóch podkreśleń) umieszczonej domyślnie w pamięci flash, to też pojawi się błąd.

Cóż na to wszystko poradzić i jak to podsumować? AFAIK umieszczanie kodu w *.data* jest mało eleganckie, choć wymaga najmniej kombinowania. Lepszym rozwiązaniem wydaje się utworzenie nowej sekcji wewnątrz *.data* (w skrypcie linkera) i pakowanie w nią wszystkich funkcji sramowych za pomocą atrybutu *section*. Na przykład tak (*.ramcode*):

259 notabene krzyczy o tym podczas linkowania: *Warning: ignoring changed section attributes for .data*
260 dotyczy tylko wyników prezentowanych w formacie *sysv*, format *berkeley* działa poprawnie

```

1. .data :
2. {
3.     . = ALIGN(4);
4.     __data_init_start = LOADADDR (.data);
5.     PROVIDE(__data_init_start = __data_init_start);
6.     __data_start = .;
7.     PROVIDE(__data_start = __data_start);
8.
9.     . = ALIGN(4);
10.    *(.data .data.* .gnu.linkonce.d.*)
11.
12.    . = ALIGN(4);
13.    *(.ramcode)
14.
15.    . = ALIGN(4);
16.    __data_end = .;
17.    PROVIDE(__data_end = __data_end);
18. } > ram AT > rom

```

W tym układzie linker nie krzyczy ostrzeżeniami (patrz przypis 259) i poprawnie generuje funkcje okleinowe przy skokach we wszystkie strony :) Pozostaje kłopot z wyświetlaniem zajętości pamięci przez program *size*. Format *berkeley* działa dobrze, ale jest dla mnie nie strawny. Cóż... problem rozwiązałem trochę naokoło - napisałem sobie mały skrypt, który formatuje to co wypluwa *size* w formacie *berkeley* tak aby nie kłuło w oczy :)

Co warto zapamiętać z tego rozdziału?

- rozdział ten można ominąć, to raczej ciekawostka niż niezbędna wiedza
- umieszczenie funkcji w sekcji *.data* (za pomocą atrybutu *section*) spowoduje, że zostanie ona skopiowana do sramu przez kod rozbiegowy programu
- najwygodniej dodać do skryptu linkera nową sekcję przeznaczoną na funkcje (patrz przykład z *.ramcode*)
- rozkazy *b* i *bl* mają ograniczony zasięg skoku, przy dłuższych skokach linker dodaje funkcje pośredniczące (*veefer*)
- za pomocą atrybutu (lub opcji) *long_call* można zmusić kompilator do generowania rozkazów umożliwiających dalekie skoki

16.8. Cały program w pamięci sram - jak?

W poprzednim rozdziale walczyliśmy z metodą, pozwalającą umieścić w pamięci sram jedną lub kilka, wybranych funkcji programu. Były one kopowane do pamięci operacyjnej z pamięci flash przy starcie programu. A co jeśli uprzemy się, aby w ogóle nie wykorzystywać

pamięci flash? Ano musimy wgrać wsad do pamięci sram z komputera PC, analogicznie jak przy zwyczajnym flashowaniu mikrokontrolera. Z tą drobną różnicą, że pamięć inna i nietrwała :)

Na potrzeby tego rozdziału powstał nowy program bazowy migający diodą:

Program bazowy, *second-edition*:

```
1. int main(void) {
2.
3.     RCC->APB2ENR = RCC_APB2ENR_IOPBEN;
4.     gpio_pin_cfg(GPIOB, PB0, gpio_mode_output_PP_2MHz);
5.     SysTick_Config(8000000/2);
6.
7.     while(1);
8.
9. } /* main */
10.
11. __attribute__((interrupt)) void SysTick_Handler(void){
12.     GPIOB->ODR ^= PB0;
13. }
```

Do rozpatrzenia mamy trzy główne zagadnienia:

- umieszczenie kodu w pamięci sram mikrokontrolera
- przygotowanie mikrokontrolera do wykonywania kodu z pamięci sram
- uruchomienie kodu z pamięci sram

Zajmijmy się pierwszym z nich. Sprawa jest jak zawsze prosta. Za rozmieszczenie obiektów w pamięci odpowiada linker. Wystarczy więc poinformować go, że wszystkie elementy programu mają wylądować w pamięci sram. Pracą linkera steruje skrypt linkera, to w nim należy dokonać kilku zmian. A te zmiany to:

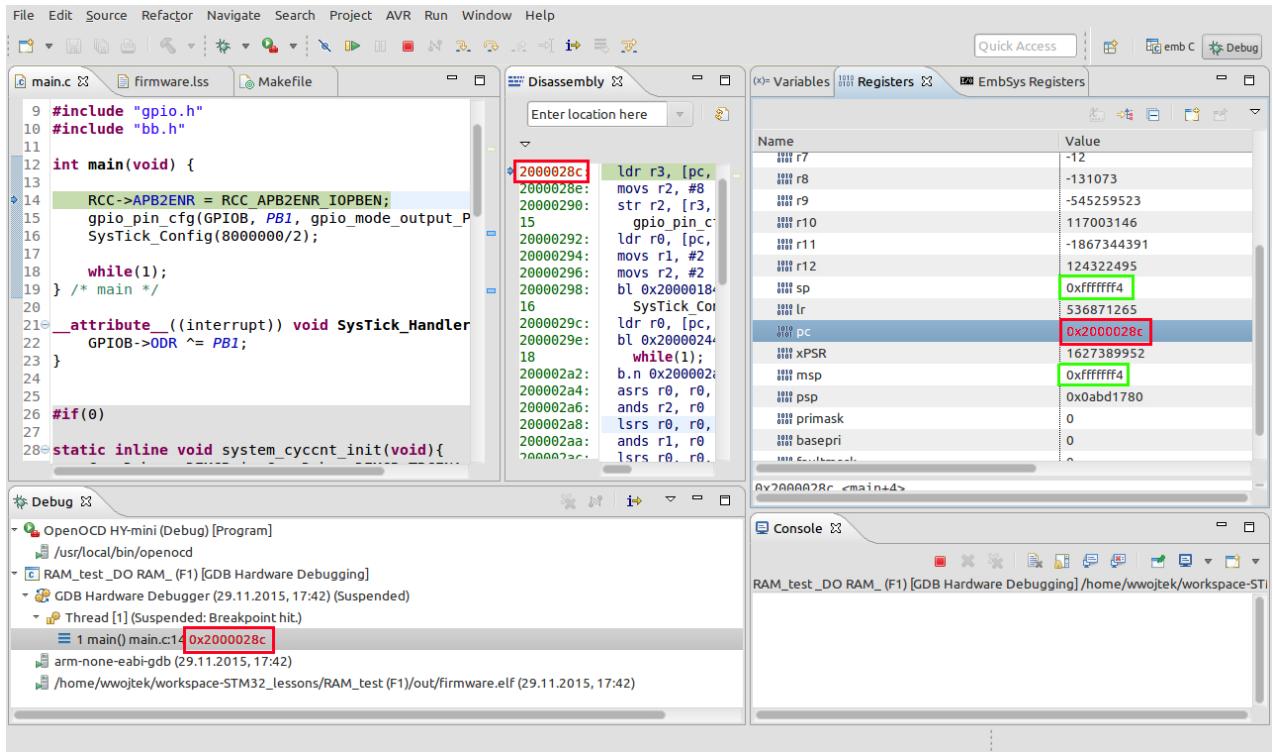
- wyrzucenie regionu pamięci *rom*
- zamiana wszystkim „*rom*” na „*ram*” w atrybutach sekcji pamięci

i to wszystko... 10s roboty. Gotowe? No to zwyczajnie wgrywamy program do mikrokontrolera. A! Przedtem proponuję upewnić się, że w pamięci flash nie ma podobnego programu. Najlepiej żeby w ogóle flash był skasowany. Dzięki temu nie będziemy mieli wątpliwości czy mikrokontroler wykonuje program z flasha czy sramu :)

No więc wgrywamy program, uruchamiamy debugowanie i... jakby działa²⁶¹! W oknie debuggera widzę, że program się uruchomił i faktycznie jest w pamięci sram (hura). Świadczy o tym wartość rejestrów *pc* wskazująca na adres w sramie (0x2000 028c). Poniżej zrzut z Eclipse'a dla

²⁶¹ jak już nieraz wspomniałem, korzystam m.in. z oprogramowania OpenOCD + Eclipse według przepisu z www.freddiechopin.info; inne programy mogą się zachowywać inaczej!

wzrokowców. Zaznaczyłem na nim, na czerwono, trzy miejsca gdzie można sprawdzić wartość *pc* (zakładka *Registers*, zakładka *Debug*, zakładka *Disassembly*).



Rys. 16.2. Program uruchomiony z pamięci sram (kolor czerwony - wskaźnik programu *pc*=0x2000 028c; kolor zielony - wskaźnik głównego stosu *msp* = *sp* = 0xffff fff4)

Bomba! Z lekkim niepokojem stepujemy sobie program, żeby zobaczyć czy wszystko działa i... „*już był w ogrodku, już witał się z gąską*”, program wysypuje się znienacka w funkcji konfigurującej porty. Nie przedłużając, przyczyną jest źle ustawiony wskaźnik stosu.

Być może zwróciłeś uwagę na wskaźnik stosu na rysunku 16.2, jego wartość wynosi 0xFFFF FFF4 (w programie wykorzystuję tylko jeden stos). Zdecydowanie nie jest to poprawna wartość. Stos powinien leżeć gdzieś w pamięci sram. Spodziewamy się więc wskaźnika ustawionego na coś z 0x2000... na początku. Co się stało, kto zawiął? My... jak zawsze :)] Musimy sobie odpowiedzieć na jedno ważne pytanie: dlaczego właściwie nasz program, wgrany do pamięci sram, się uruchomił? Przecież mikrokontroler powinien po resecie wystartować z pamięci flash a nie sram. Ano! Za uruchomienie się programu w pamięci sram odpowiada oprogramowanie OpenOCD. Program odczytał sobie adres od którego rozpoczyna się kod programu (*Entry Point* ze skryptu linkera) i w taki czy inny sposób wymusił start procesora z tego miejsca. Szkopuł polega na tym, że OpenOCD nie zadbał o poprawne odczytanie wskaźnika stosu.

Cortex, po resecie, sprzętowo odczytuje adres wierzchołka stosu spod adresu 0 pamięci (pierwszy wpis tablicy wektorów). W rozdziale 16.2 wspomniałem co prawda o mechanizmie

mapowania pamięci pod adresem 0, ale w naszym przykładzie nic nie kombinowaliśmy, więc „domyślnie” pod adresem zero jest pamięć flash. I właśnie z tej pamięci procesor odczytał sobie adres wierzchołka stosu. Ponieważ jednak cały program (wraz z tablicą wektorów i kodem rozruchowym) wgraliśmy do pamięci sram, pamięć flash nie zawiera prawidłowego wskaźnika stosu! Już ja o to zadbałem upewniając się, że jest ona skasowana :) Skasowane komórki pamięci flash zawierają wartość 0xFFFF FFFF. I taki właśnie, bezsensowny, wskaźnik stosu został ustawiony przy starcie rdzenia. Niższa wartość widoczna na zrzucie z rysunku 16.2 (0xFFFF FFF4) wynika z tego, że kilka rejestrów zostało już odłożonych na stos na początku funkcji main.

No to spróbujmy obejść ten błąd i zobaczyć co będzie dalej. Prawidłową wartość wskaźnika stosu odczytamy np. z pliku *.lss*. To jest wartość pierwszego słowa wsadu, czyli wartość spod adresu 0x2000 0000. W moim programie wynosi ona 0x2000 06e8. Ustawimy ją ręcznie w oknie debuggera (np. w zakładce *Registers*) i zobaczymy co będzie dalej :) Eureka! Program dał się „przekrokać” aż do pętli nieskończonej. Puścmy go więc samopas i zobaczymy czy dioda w przerwaniu SysTicka migła... „ciemność, widzę ciemność, ciemność widzę”!

No i znowu klapa! Program się ponownie wysypał. Szybki rzut oka na zakładkę *Registers* i widać, że wskaźnik programu (*pc*) pokazuje na bzdurny adres: 0xFFFF FFFE. Jak nietrudno zgadnąć, program wysypał się przy próbie obsługi przerwania. Tablica wektorów domyślnie zaczyna się od adresu 0 (wskaźnik stosu jest jej pierwszym elementem). My wgraliśmy cały program do pamięci sram i nie poinformowaliśmy mikrokontrolera, że to właśnie tam ma szukać tablicy. Gdy pojawiło się przerwanie, nieświadomu niczego Cortex, odczytał sobie adres ISR z pamięci flash zamiast sram. I tu jest pies pogrzebany.

Do poinformowanie procesora o tym, aby raczył szukać tablicy wektorów pod konkretnym adresem, służy register *SCB_VTOR*. Ustawia się w nim położenie tablicy, przy czym musi być ono wyrównane do ilości pozycji w tablicy. Ponadto minimalne wyrównanie wynosi 128B.

No to już wszystko wiemy :) Czas wprowadzić małe poprawki. Jeśli założymy, że przed funkcją główną main nie pojawi się żaden wyjątek, to przesunięcie tablicy możemy od biedy zrobić już w funkcji głównej. Oczywiście jakkolwiek wyjątek, który pojawi się wcześniej, wysypie program. Co do stosu to sprawa wygląda gorzej, zdecydowanie zależy nam na tym, aby był ustawiony jak najwcześniej - przed skokiem do funkcji głównej - bo już na początku *main* program odkłada coś tam na stosie. Skoro i tak musimy grzebać w procedurze rozbiegowej to wrzucimy do niej od razu i konfigurację stosu i przesunięcie tablicy wektorów.

W przykładowym projekcie Freddiego Chopina²⁶² kod rozruchowy jest napisany w assemblerze i umieszczony w osobnym pliku *startup.S*. Toteż i tam umieścimy nasz kawałek kodu

262 www.freddiechopin.info jakby ktoś jeszcze nie wiedział

odpowiedzialny za ustawienie wskaźnika stosu i przesunięcia tablicy przerwań. Na samym początku procedury rozruchowej dopisuję co następuje

Ustawienie wskaźnika stosu i przesunięcia tablicy przerwań:

-
1. ldr r0, =__main_stack_end
 2. msr MSP, r0
 - 3.
 4. ldr r0, =text_start
 5. ldr r1, =0xe000ed08
 6. str r0, [r1]

Kod korzysta z dwóch symboli zdefiniowanych w skrypcie linkera. Myślę, że jest na tyle prosty, że każdy sam poradzi sobie z jego analizą. Podpowiem tylko, że 0xe000 ed08 to adres rejestru SCB_VTOR. Kompilujemy, wgrywamy²⁶³ i... działa! Miga! It's alive!

Ale może być jeszcze lepiej :] Zresetuj mikrokontroler... i co? Kicha. Tym razem OpenOCD nie wymusił uruchomienia naszego programu w sramie - robi to tylko raz, po wgraniu programu. Po resecie mikrokontroler startuje z adresu 0 (pamięć flash) i się wysypuje. Co możemy z tym zrobić? Opcje są dwie:

- wrzucić do pamięci flash krótki programik, który uruchomi program w pamięci sram
- wymusić sprzętowo uruchomienie mikrokontrolera z pamięci sram (patrz rozdział 16.2)

Zacznijmy od pierwszej kropki. Nasz krótki programik będzie trochę podobny do tego co niedawno robiliśmy w startupie programu w sramie. Będzie robił trzy, proste, rzeczy:

- ustawiał prawidłowy wskaźnik stosu
- konfigurował przesunięcie tablicy wektorów do pamięci sram
- wykonywał skok do programu w sramie

Dzięki trzem powyższym kropkom, program wgrany do sramu nie będzie musiał już sam ustawiać sobie wskaźnika stosu i przesunięcie tablicy przerwań. Możemy więc cofnąć zmiany jakie wykonaliśmy przed chwilą w jego startupie. Nie przedłużając, przykładowy program realizujący trzy powyższe zadania przedstawiam poniżej. To jest cały „program”:

263 i nie musimy się martwić o zużywanie pamięci flash bo cały program wgrywamy do sram :] żarcik taki...

Program uruchamiający program z pamięci sram:

```
1. #define Sram_Start 0x20000000
2.
3. .word Sram_Start
4. .word Reset_Handler
5.
6. Reset_Handler:
7.
8.     /* SP */
9.     ldr r0, =Sram_Start
10.    ldr r1, [r0]
11.    msr MSP, r1
12.
13.    /* VTOR */
14.    ldr r1, =0xe000ed08
15.    str r0, [r1]
16.
17.    /* Hop */
18.    ldr r0, [r0, #4]
19.    bx r0
```

Linijka 1 to definicja początku pamięci sram, dla wygody. Linijki 3 i 4 to mikro tablica wektorów. Zawiera tylko dwie, niezbędne, pozycje. Pierwsza wartość to adres wierzchołka stosu. Ustawiam na początek pamięci operacyjnej. Nie ma to teraz większego znaczenia, ale coś ustawić trzeba. Druga wartość to adres początku kodu (procedury obsługi wyjątku *reset*). Reszta programu to realizacja trzech kropek o których wcześniej była mowa. Prościzna :) Wgrywamy program do pamięci flash. Potem wgrywamy program (np. migającą diodę) do pamięci sram i paczamy czy działa. Przypominam, że program wgrywany do pamięci sram nie musi już przestawiać sobie wskaźnika stosu i przerwań, powyższy „loader” robi to za niego.

Wgrywamy, odpalamy, resetujemy mikrokontroler i zauważamy co następuje: po wgraniu programu do sramu i uruchomieniu go przez OpenOCD - nie działa. Po zresetowaniu mikrokontrolera - działa. Zastanów się dlaczego tak, potem czytaj dalej. To, że dioda migła po zresetowaniu mikrokontrolera wynika z tego, że nasz program działa i robi to co ma robić. To, że nie działało po uruchomieniu w debuggerze wynika z tego, że OpenOCD wymusza start programu w sram (tak jak poprzednio) z pominięciem naszego krótkiego kodu w pamięci flash. Czyli znowu nie mamy ustawionego stosu i położenia wektorów. Inteligencji zawsze wiatr w oczy.

Został jeszcze jeden, finalny, punkt programu - zmuszenie mikrokontrolera, żeby sprzętowo uruchomił się z pamięci sram. Tak jak opisywałem w rozdziale 16.2, za pomocą nóżek BOOTT²⁶⁴, można zmienić mapowanie pamięci tak, aby od adresu 0 dostępna była pamięć sram. To by było dla nas genialne rozwiązanie! Wgrywamy zwyczajny program do pamięci sram i prawie o nic się nie martwimy. Sprzęt odwala za nas całą (!) robotę. Procesor odczytujący pamięć od adresu zero, czyta nasz program w pamięci sram.

264 lub bitu konfiguracyjnego nBOOT1 w przypadku mikrokontrolera F334

Robimy szybki test na F429²⁶⁵: kasuję pamięć flash, wgrywam program do sram, podciągam nóżki BOOT0 i BOOT1 do zasilania. Chwila prawdy - restart - i działa! *We are the champions!* Są jednak dwa „ale”:

- dokumentacja (RM) zaleca, aby w przypadku bootowania z pamięci sram ustawić przesunięcie tablicy wektorów w rejestrze SCB_VTOR na adres w pamięci sram. Nie wiem czemu: przecież pamięć jest zmapowana... no i test empiryczny pokazuje, że wszystko działa prawidłowo (o_0)
- to nie zadziała w F103 :]

Z F103 jest taki „wałek”, że bootowanie z pamięci sram działa dosyć osobliwie. Żeby było śmieszniej, nie udało mi się znaleźć żadnego dokumentu, który jasno opisywałby co się właściwie dzieje... i dlaczego. Internet też wiele nie pomaga. Ale przynajmniej strzępki znalezionych informacji są zgodne z tym, do czego sam doszedłem z pomocą debuggera. A to już coś.

Mikrokontroler F103, przy uruchamianiu z pamięci sram, robi dwie dziwne rzeczy. Po pierwsze wskaźnik stosu jest ustawiany na wartość 0x2000 5000. Uprzedzając wszelkie pytania z tym związane: nie wiem :] Druga, jeszcze dziwniejsza rzecz, to to że procesor rozpoczyna wykonywanie kodu nie od początku sramu, a od adresu 0x2000 01e0. Mapowanie sramu w ogóle jest jakieś dziwne. Po „zbootowaniu” ze sramu, mogę odczytać debuggerem tylko 4 pierwsze słowa pamięci od adresu 0. Ich wartości to kolejno:

- 0x2000 5000 (adres wierzchołka stosu)
- 0x2000 01e1 (adres początku kodu)
- 0x2000 0004
- 0x2000 0004

Próba odczytu dalszych kawałków pamięci wywala błędy... Próba zmiany wartości powyższych słów też nie przynosi rezultatu. Generalnie to jakieś dziwne jest. W Internecie udało mi się w sumie wygrzebać tylko jeden wątek na forum ST ([BootLoader from the RAM](#)). Kilka ciekawych cytatów z tego wątku (autorstwa użytkownika *clive1*):

„STM32 RAM booting doing some odd things with the PC vector, STM32F1 parts jump to very specific locations”

„There was some conjecture that it was executing some other code first”

265 nie na F103! Uzasadnienie za chwilę :)

O co by nie chodziło, czyni to bootowanie ze sram w F103 dosyć kłopotliwym. Program musi zaczynać się od adresu 0x2000 01e0, wskaźnik stosu trzeba sobie samemu przestawić, no i trzeba skonfigurować położenie tablicy wektorów. Dwie ostatnie rzeczy już nieraz ćwiczyliśmy. Co do położenia programu, mój pomysł jest taki: tablica wektorów zostaje na początku sramu, zaś uchwyt wyjątku Reset leci pod adres 0x2000 01e0. Wymaga to drobnej korekty w skrypcie linkera w sekcji `.text`:

Modyfikacja skryptu linkera:

```
1. .text :
2. {
3.     . = ALIGN(4);
4.     __text_start = .;
5.     PROVIDE(__text_start = __text_start);
6.
7.     . = ALIGN(4);
8.     KEEP(*(.vectors));
9.
10.    . = 0x1e0;
11.
12.    . = ALIGN(4);
13.    *(.text .text.* .gnu.linkonce.t.*);
14.    . = ALIGN(4);
15.    *(.glue_7t .glue_7);
16.    . = ALIGN(4);
17.    *(.rodata .rodata.* .gnu.linkonce.r.*);
18.    ...
```

A co z F334? Tutaj mapowanie pamięci działa bez niespodzianek, tak jak w F429. Problem natomiast występuje z wymuszeniem bootowania z pamięci SRAM. W tym mikrokontrolerze nie ma nóżki BOOT1. Zamiast niej jest bit konfiguracyjny nBOOT1. Ewentualnie sposób mapowania pamięci można zmienić w programie poprzez bity SYSCFG_CFGR1_MEM_MODE. I tyle w temacie :)

Co warto zapamiętać z tego rozdziału?

- ten rozdział można ominąć, to raczej ciekawostka niż niezbędna wiedza
- jeżeli nie chcemy w ogóle korzystać z pamięci flash, to program musi być wgrany do sramu przez komputer PC (takie... sramowanie mikrokontrolera)
- aby program wylądował w pamięci sram należy pozamieniać w skrypcie linkera wszystkie „rom” na „ram”, a region „rom” najlepiej w ogóle wyrzucić
- uruchamiając program z pamięci sram musimy w szczególności zadbać o dwie sprawy:
 - prawidłowy adres wierzchołka stosu
 - prawidłową konfigurację położenia tablicy wektorów

- są trzy opcje uruchomienia programu w pamięci sram:
 - można wykorzystać program OpenOCD, który wymusza start wgrywanego programu (ale sami musimy zadbać o stos i tablicę przerwań)
 - można wgrać do pamięci flash prosty *loader*, który: ustawi stos, skonfiguruje położenie tablicy wektorów i skoczy do kodu w sram
 - można wymusić sprzętowo (nóżki BOOT0 i BOOT1 w stanie wysokim lub bit nBOOT1 w stanie niskim w przypadku F334) bootowanie z pamięci sram:
 - bootowanie z pamięci sram w F429 i F334 jest cacy - nie trzeba się martwić o stos i przerwania (przy czym, nie wiadomo po co, RM zaleca jednak ustawienie SCB_VTOR)
 - bootowanie z pamięci sram w F103 jest be - procesor startuje od dziwnego adresu (trzeba ustawić stos, położenie tablicy i odpowiednio umiejszczyć program w pamięci)

17. SYSTEM ZEGAROWY („*FINITA EST COMOEDIA*”²⁶⁶)

17.1. Wstęp

Tak! To już! Za chwilę przekroczyłeś Rubikon STMów! Wreszcie nadszedł moment wtajemniczenia - zegary! Mityczna inicjalizacja mikrokontrolera, którą straszy się początkujących. „Dioda nie migła? A jak skonfigurowałaś zegary?”, „Przerwanie nie działa? Pokaż *inicjalizację*²⁶⁷ procesora!”, „Do ustawienia zegarów potrzebna jest biblioteka!”, „Pierwsza rzecz to konfiguracja zegarów!”, „Żle ustawiłem PLL i zablokowałem procesor”... brednie, farmazony, bujdy i niedorzeczności :)

Nie da się ukryć, że w przerobionych przykładach *ocieraliśmy się*²⁶⁸ konfiguracją systemu zegarowego w STMie. Pojawiały się jakieś akronimy typu HSI, LSI, zegary APB1 itd. coś tam włączaliśmy w bloku RCC. Ale! Uruchomiliśmy wspólnie sporą część układów peryferyjnych a żadnych skomplikowanych konfiguracji zegarów nie robiliśmy, prawda? Owszem, przyznaję, w kilku miejscach trochę brakowało szczegółowych informacji o bloku zegarowym (RCC), ale zdecydowałem się jednak zostawić ten rozdział na koniec. Dlategoż iż:

- żeby nie straszyć na początku i nie utwierdzać początkujących Czytelników w przekonaniu, że w STMacie to trzeba jakieś czary odprawiać, żeby mikrokontroler w ogóle ruszył
- żeby było inaczej, w większości poradników nt. STMów opis RCC jest na samym początku
- bo to nudny rozdział i nie wnosi nic spektakularnego :)

Możesz mieć do mnie odrobinę żalu. Bo teraz kilka rzeczy które *nad wyraz intelligentnie* przemilczałem wcześniej, stanie się jasnymi. Cóż... uprzedzałem: bez pracy własnej nic z tego nie będzie :] Jeśli gdzieś brakowało Ci jakiś informacji to trzeba było sobie poszukać!

Trochę informacji wstępnych na rozgrzewkę. Taktowanie w STM32 jest bardzo.... elastyczne :) W AVRach źródło zegara i ewentualne preskalery konfigurowało się głównie poprzez *fuse bity*. Rodziło to niebezpieczeństwo, że po dokonaniu błędnych zmian w konfiguracji, początkujący AVRmator straci możliwość komunikacji z mikrokontrolerem²⁶⁹. W kilku miejscach zauważylem, że te obawy przed zmianami w systemie zegarowym są potem przenoszone na inne mikrokontrolery. Nie potrzebnie! W STMie sygnał zegarowy jest konfigurowany w programie. Nawet jeśli jakimś cudem doprowadzimy do sytuacji, w której mikrokontroler nie będzie chciał działać, to wystarczy uruchomić go w trybie bootloadera żeby nie wykonywał błędnej konfiguracji i

266 „Komedii skończona.”

267 przynajmniej dobrze, że nie *inicjację*

268 ta „o siebie”? ja Ci dam że „o siebie” :)

269 tylko błagam, nie „zablokowana” czy „zepsuta” atmega. Błędną konfiguracją Fuse Bitów nie da się zablokować czy zepsuć atmela na amen!

wgrać nowy wsad. Dodatkowo STM potrafi się całkiem solidnie bronić przed naszymi próbami unieruchomienia go, ale o tym za chwilę.

STM może współpracować z czterema źródłami sygnału zegarowego (oczywiście nie wszystkimi naraz).

- **HSI** (*High Speed Internal*) to wbudowany oscylator dużej częstotliwości. To coś jak wewnętrzne źródło sygnału zegarowego w AVRach. Zaletą jest to, że jest wbudowany i szybko startuje, wadą niska stabilność. Częstotliwość zegara HSI w rodzinie STM32 nie jest stała, zależy od konkretnego modelu mikrokontrolera (F103 - 8MHz, F429 - 16MHz).
- **HSE** (*High Speed External*) to zewnętrzny odpowiednik HSI, czyli oscylator stabilizowany przez zewnętrzny rezonator (ceramiczny, kwarcowy) lub zewnętrzne źródło sygnału (może być kwadrat, trójkąt lub sinus)
- **LSI** (*Low Speed Internal*) to wewnętrzne źródło niskiej częstotliwości (~30...60kHz). Wadą jest mała dokładność i stabilność częstotliwości oraz spory pobór prądu (w porównaniu z LSE).
- **LSE** (*Low Speed External*) to zewnętrzne źródło niskiej częstotliwości (32,768kHz). Ceramiczny lub kwarcowy rezonator. Możliwe jest również podanie zewnętrznego sygnału zegarowego o częstotliwości do 1MHz.

Układy dużej częstotliwości służą do taktowania rdzenia i bloków peryferyjnych mikrokontrolera. Po resecie mikrokontrolera, jest on taktowany z źródła HSI. Ponadto jest ono wykorzystywane w czasie programowania pamięci flash i w przypadku awarii zewnętrznego źródła HSE. Układy niskiej częstotliwości taktują przede wszystkim niezależny *watchdog*. Blok RTC może być taktowany z obu grup źródeł (dużej i małej częstotliwości).

PLL (*phase locked loop*) to jest czad :) Póki nie wiedziałem co to jest PLL to się trochę przeraziłem – kupiłem płytę startową z mikrokontrolerem który może być rozbijany do 72MHz a na płytce siedzi kwarc 8MHz... myślę sobie – ale jaja! Teraz, gdy już wiem (ale jestem mądry...)... wątek mi się urwał. Tak czy siak, PLL to taki „mnożnik” częstotliwości. Jak to działa fizycznie nie chcę wiedzieć. Liczy się efekt, a efekt jest taki, że jeśli na wejście pętli PLL podamy sygnał 8MHz i rozbijamy pętlę PLL tak aby mnożnik wynosił 9x to otrzymamy na wyjściu 72MHz, którymi możemy taktować mikrokontroler – czy to nie piękne!

Co warto zapamiętać z tego rozdziału:

- co to jest HSI, HSE, LSI, LSE
- błędą konfiguracją systemu zegarowego nie zepsujesz mikrokontrolera
- konfigurowanie systemu zegarowego nie jest zawsze konieczne, mikrokontroler domyślnie działa na wbudowanym oscylatorze HSE
- pętla PLL mnoży częstotliwość

17.2. System zegarowy (F103)

W tym miejscu przerywamy lekturę, odpalamy sobie datasheet posiadanego mikrokontrolera, znajdujemy tam rysunek pod tytułem *clock tree* (gdzieś na początku) i drukujemy w możliwie najlepszej jakości! To drzewko będzie nam pomocne... jak schemat blokowy przy licznikach. Wiem że w pierwszej chwili wygląda to skomplikowanie, ale... przecież ogarnęliśmy liczniki i ADC - nic gorszego nie może nas spotkać!

Przelećmy ogólnie ten schemat. Po lewej stronie, na górze, znajduje się wewnętrzne źródło HSI. Sygnał z tego źródła może być doprowadzony do:

- FLITFCLK - to coś z programowaniem Flasha (kontroler czy jakiś inszy czort)
- multipleksera z którego wychodzi zegar systemowy SYSCLK (ten to jest ważny!)
- do dzielnika /2 z którego wchodzi na multiplekser i do bloku PLL

Proste. Nad niektórymi blokami mamy od razu podane bity konfiguracyjne. Np. PLLMUL nad blokiem PLL, czy PLLSRC nad multiplekserem PLL.

Jak zjedziemy trochę niżej to znajdziemy wejście sygnału zewnętrznego HSE. Sygnał HSE może być wykorzystany:

- jako zegar systemowy SYSCLK
- jako źródło dla pętli PLL (przy czym tu mamy dodatkową możliwość podziału /2 - patrz multiplekser sterowany bitami PLLXTPRE)
- do taktowania układu RTC (HSE/128)

Niżej, po lewej stronie na drzewku, widać jeszcze wejście LSE oraz układ LSI. Na samym dole zaznaczone jest wyjście sygnału zegarowego (MCO *Microcontroller Clock Output*) i multiplekser pozwalający wybrać sygnał wyjściowy. Do wyboru są:

- zegar wychodzący z PLL podzielony przez 2 (PLLCLK/2)
- sygnał z wewnętrznego źródła zegarowego (HSI)
- sygnał z zewnętrznego źródła zegarowego (HSE)
- zegar systemowy SYSCLK

Wyjście MCO można użyć np. aby dostarczyć sygnał zegarowy z mikrokontrolera do innego układu cyfrowego zastosowanego w budowanym urządzeniu.

Wracamy w okolice PLL i multipleksera wybierającego źródło sygnału zegara systemowego SYSCLK. Jest tam bloczek CSS (*Clock Security System*). Układ ten zabezpiecza mikrokontroler przed unieruchomieniem w przypadku awarii zewnętrznego źródła sygnału (HSE). W takiej sytuacji układ CSS przestawia multiplekser tak, aby sygnał SYSCLK pochodził z wewnętrznego źródła HSI. Miło z jego strony, prawda? :)

Po prawo robi się trochę gęściej. SYSCLK (maksymalnie 72MHz) leci do góry do jakichś interfejsów I2S oraz wchodzi na preskaler szyny AHB (sygnał zegarowy tej szyny nazywa się HCLK, maks 72MHz). Pamiętasz jak np. przy korzystaniu z DMA włączaliśmy taktowanie bloku w rejestrze AHBENR? To dlatego że układ DMA jest „zasilany” z szyny AHB. Podobnie jak kilka innych układów (patrz drzewko). Oprócz AHB mamy jeszcze dwie szyny: APB1 (sygnał zegarowy PCLK1, 36MHz maks.) i APB2 (sygnał zegarowy PCLK2, 72MHz maks.). Sygnały z tych szyn taktują podłączone do nich układy peryferyjne. Jak łatwo się domyśleć, taktowanie tych układów włącza się w rejestrach APB1ENR i APB2ENR, dedukcja mój drogi Watsonie :) Każda z szyn ma swój preskaler.

Na koniec zwróć uwagę na bloczki z „warunkowym” podziałem częstotliwości liczników. Znajdź np. wyjście sygnału zegarowego dla liczników TIM1 i TIM8. Przechodzi to to przez taki dziwaczny bloczek:

```
If (APB2 prescaler = 1) x1  
else x2
```

O co chodzi? Dokładnie o to co jest napisane: jeśli preskaler zegara dla szyny APB2 jest równy 1 to bloczek robi „x1” czyli nie zmienia częstotliwości sygnału. W przeciwnym wypadku, bloczek robi „x2” czyli podwaja częstotliwość sygnału zegarowego dochodzącego do liczników. Ot taki *trap for young players*.

To tak na rozgrzewkę, prześledźmy drogę sygnału zegarowego od zewnętrznego rezonatora do licznika TIM1 (z wykorzystaniem PLL):

- rezonator podłączony jest do nóżek OSC_IN, OSC_OUT - stąd startujemy
- dalej jest blok oscylatora zewnętrznego (HSE OSC)
- dalej sygnał się rozdziela i idzie do kilku bloków:
 - przez dzielnik częstotliwości (/128) idzie na multiplekser, gdzie za pomocą bitów RTCSEL można wybrać sygnał taktujący RTC (RTCCLK)
 - wchodzi do multipleksera PLLXTPRE (w zależności od konfiguracji multiplekser przepuszcza albo sygnał HSE albo HSE/2)
 - idzie do bloku CSS (*clock security system*)
 - wchodzi na multiplekser z którego brany jest sygnał zegarowy SYSCLK
- wybieramy opcję numer dwa (PLLXTPRE)
- za pomocą bitów PLLXTPRE wybieramy HSE bez podziału (potrzebny nam do czegoś podział?)
- za pomocą bitów PLLSRC wybieramy nasz sygnał jako źródło dla PLL
- za pomocą bitów PLLMULL konfigurujemy mnożnik PLL wedle uznania
- doszliśmy do SYSCLK, dalej mamy preskaler szyny AHB
- z szyny AHB idziemy przez kolejny preskaler na szynę APB2
- kolejny bloczek jest fajny – to ten warunkowy „antypreskaler” który podwaja częstotliwość APB2 jeśli preskaler APB2 jest różny od 1
- na końcu mamy bramkę włączającą sygnał zegarowy licznika (*Peripheral Clock Enable*) i wreszcie sygnał TIMxCLK dla liczników

W rozdziale poświęconym licznikom, rozwiązujeć zadania domowe, zakładaliśmy zawsze że częstotliwość taktowania licznika (z wewnętrznego źródła sygnału zegarowego) jest równa „domyślnej” częstotliwości pracy mikrokontrolera (8 lub 16MHz). To było oczywiście perfidne uproszczenie. Częstotliwość taktowania licznika jest taka, jak szyny do której jest on podłączony. To samo dotyczy również innych peryferiali. Pamiętasz może przykład z układem WWDG w F429, gdzie nie mogłem uzyskać opóźnień takich jakie chciałem i kombinowałem coś z zegarami uprzedzając, że wyjaśni się później (zadanie 10.2)? No to właśnie się wyjaśnia :) Częstotliwość taktowania układów peryferyjnych zależy od częstotliwości taktowania szyn, do których są one podłączone. Na przykład przetwornik ADC! Przypomnij sobie rozdział dotyczący czasu

próbkowania (rozdział 13.3), dosyć często pojawiała się tam informacja, że: „coś tam, coś tam... jeśli ADC jest taktowane z maksymalną częstotliwością równą 14MHz”. W ramach wprawki zobacz skąd się bierze (na drzewku zegarowym) sygnał zegarowy ADC (ADCCLK) :)

Generalnie nie ma w tym nic trudnego, prawda? To taka zabawa w „znajdź drogę przez labirynt aby Muminek trafił do Migotki²⁷⁰” i nic poza tym :)

Zadanie domowe 17.1: przestawić źródło zegarowe na HSE i rozburzyć mikrokontroler do maksymalnych możliwych częstotliwości. Tzn:

- SYSCLK = 72MHz
- HCLK = 72MHz
- PCLK1 = 36MHz
- PCLK2 = 72MHz

W celach testowych dorzucić miganie diodą w SysTicku (1Hz), aby upewnić się, że mikrokontroler rzeczywiście pracuje z założoną częstotliwością. Nie zapomnij o opóźnieniach w dostępie do flasha :)

Podpowiedź: w skrócie musisz zaliczyć następujące kroki (niekoniecznie muszą być w tej kolejności):

- włączyć oscylator zewnętrzny HSE
- ustawić opóźnienie w odczycie flasha
- doprowadzić sygnał z HSE do PLL
- dobrać mnożnik PLL i ją włączyć
- ustawić dzielniki szyn AHB, APB1, APB2
- przestawić źródło zegara systemowego na PLL

²⁷⁰ albo do Włóczykija jak kto woli... ja nie oceniam

Przykładowe rozwiązanie (F103, dioda na PB0, zewnętrzny rezonator 8MHz):

```
1. int main(void) {
2.
3.     RCC->CR |= RCC_CR_HSEON;
4.     RCC->CFGGR = RCC_CFGGR_PLLMULL9 | RCC_CFGGR_PLLSRC | RCC_CFGGR_ADCPRE_DIV6 |
5.                 RCC_CFGGR_PPREG1_DIV2 | RCC_CFGGR_USBPRE;
6.     while (!(RCC->CR & RCC_CR_HSERDY));
7.     RCC->CR |= RCC_CR_PLLON;
8.     FLASH->ACR |= FLASH_ACR_LATENCY_1;
9.     while (!(RCC->CR & RCC_CR_PLLRDY));
10.    RCC->CFGGR |= RCC_CFGGR_SW_PLL;
11.    while ((RCC->CFGGR & RCC_CFGGR_SWS) != RCC_CFGGR_SWS_PLL);
12.    RCC->CR &= ~RCC_CR_HSION;
13.
14.    RCC->APB2ENR = RCC_APB2ENR_IOPBEN;
15.    gpio_pin_cfg(GPIOB, PB0, gpio_mode_output_PP_2MHz);
16.    SysTick_Config(72000000/8/2);
17.    SysTick->CTRL &= ~SysTick_CTRL_CLKSOURCE_Msk;
18.
19.    while(1) __WFI();
20.
21. } /* main */
22.
23. __attribute__((interrupt)) void SysTick_Handler(void){
24.     BB(GPIOB->ODR, PB0) ^= 1;
25. }
```

Koszmarnie tru... nudne...

3) włączenie oscylatora HSE, gdy oscylator zacznie stabilnie pracować to zostanie ustawiona flaga HSERDY (patrz linia 6)

4) żeby nie marnować czasu, w oczekiwaniu na flagę HSERDY, możemy sobie ustawić parę drobiazgów:

- mnożnik pętli PLL ustawiam tak, aby uzyskać maksymalną częstotliwość zegara SYSCLK (72MHz), zewnętrzny kwarc ma 8MHz stąd $72/8 = 9$
- przestawiam źródło sygnału pętli PLL na HSE, na razie nie włączam pętli PLL bo nie jestem pewny czy HSE już działa
- pamiętasz rozdział o ADC? Była tam informacja, że aby ADC poprawnie działał, może być taktowany z maksymalną częstotliwością równą 14MHz. Jak zerkniesz na drzewko zegarowe to zobaczyysz, że ADC bierze zegar z szyny APB2 (72MHz) oraz ma swój osobisty preskaler. Obliczenie nastawy preskalera jest banalne jak zawsze :) $72/14 = \sim 5.14$ zaokrąglamy w górę do 6. Tutaj mała dygresja:

Czas pomiaru ADC jest zależny od częstotliwości taktowania przetwornika. Minimalny czas konwersji otrzymamy wtedy, kiedy częstotliwość pracy przetwornika będzie maksymalna (14MHz). Jeśli bardzo zależy nam na skróceniu czasu pomiaru to musimy tak dobrą częstotliwość pracy mikrokontrolera (a dokładniej zegara szyny APB2), aby udało się uzyskać te 14MHz na wejściu ADC.

Np. jeśli PCLK2 (zegar szyny APB2) jest równy 72MHz to, aby nie przekroczyć maksymalnej częstotliwości ADC, musimy ustawić preskaler na minimum 6 ($72\text{MHz}/6 = 12\text{MHz}$). To nie jest maksymalna częstotliwość pracy ADC, ale nie możemy zmniejszyć nastawy preskalera ADC do 5 bo przekroczymy graniczne 14MHz! Innymi słowy: dla PCLK2 = 72MHz nie mamy możliwości uzyskania minimalnego czasu konwersji ($1\mu\text{s}$)! Aby uzyskać minimalny czas pomiaru należy wybrać inną (niższą) częstotliwość zegara PCLK2. Np. 56MHz (preskaler ADC = 4). Taki paradoks: wolniej żeby krócej :) Koniec OT!

Ustawienie preskalera ADC w tym miejscu nie jest konieczne. Szczególnie, że z niego nie korzystamy w programie :) Ale co nam szkodzi! Dzięki temu będziemy mieli to już z głowy.

- kolejny bit (cały czas linia 4/5 listingu) to preskaler dla szyny APB1 (jej maksymalna częstotliwość to 36MHz)
- ostatni bit to preskaler USB. Interfejs USB jest wymagający i musi mieć zegar równy 48MHz. Na drzewku zegarowym widać, że ma swój osobisty preskaler (podobnie jak ADC). Ustawiamy podział przez 1,5 ($72\text{MHz} / 1,5 = 48\text{MHz}$). Podobnie jak w przypadku ADC, nie musimy ustawiać tego preskalera. Ale co nam szkodzi...

6) my się bawiliśmy z preskalерами, a HSE cały czas się roznikał. Czekamy na flagę gotowości HSE.

7) włączamy pętlę PLL, ona też potrzebuje chwili na rozbudzanie się

8) w czasie kiedy startuje PLL, żeby nie marnować czasu, konfiguruję opóźnienia w dostępie do pamięci flash

9) czekamy na gotowość pętli PLL

10, 11) za pomocą pola bitowego SW przestawiam źródło zegara systemowego (SYSCLK) na pętlę PLL. Następnie czekam na zakończenie tej operacji. Pole SWS to kilka bitów, stąd taki rozbudowany warunek w pętli.

12) wyłączam wewnętrzny oscylator HSI, po co ma pobierać energię

16, 17) konfiguracja SysTicka tak aby przerwanie było odpalane co 0,5s. Czemu tak dziwacznie? SysTick to licznik 24 bitowy. Czyli maksymalnie może zliczyć 0xFFFFFFFF impulsów zegarowych. Zegar HCLK (pełniący m.in. SysTick) ma teraz 72MHz. Czyli żeby uzyskać przerwanie co 0,5s, SysTick powinien zliczyć 36 000 000 cykli zegarowych ($36\ 000\ 000 = 0x2255100$). To jest więcej niż rozdzielcość licznika²⁷¹! W związku z tym wprowadzamy małą modyfikację. SysTick może być taktowany z HCLK lub HCLK/8 (patrz drzewko zegarowe). Jeśli wybierzemy drugą opcję to, dla założonej częstotliwości przerwań, SysTick będzie musiał zliczać

²⁷¹ funkcja SysTick_Config() zwraca 1 jeśli podana w argumencie ilość taktów zegara przekracza możliwości licznika

jedynie: $72\ 000\ 000 / 8 / 2 = 4\ 500\ 000$ cykli zegarowych (0x44AA20). I ta wartość zmieści się w liczniku :) Stąd taka konfiguracja pokrecona.

19) uśpienie rdzenia żeby nasz blinking był *eco* :)

Uruchamiamy program, bierzemy stoper i liczymy mignięcia (przez np. 30s.) aby sprawdzić czy na pewno działa :)

Zadanie domowe 17.2: Zobaczmy jak działa *Clock Security System*. W tym celu weźmiemy kod z poprzedniego zadania (17.1) i zasymulujemy układowi zawał rezonatora :) Awarię rezonatora upozorujemy zwierając mikrokontrolerowi nóżkę OSC_IN do masy przez rezystor $\sim 10\text{k}\Omega$. **Uwaga!** Szczerze powiedziawszy nie wiem czy to jest w 100% bezpieczna (dla mikrokontrolera) zabawa... także ten... żeby potem nie było na mnie :) Kto się boi, może bazować na moich obserwacjach²⁷². Po wykonaniu tego eksperymentu z kodem z poprzedniego zadania, ponawiamy próby uprzednio włączywszy układ CSS.

Obserwacje: po unieruchomieniu oscylatora migająca dioda się zatrzymała. To było do przewidzenia. Mikrokontroler stracił sygnał zegarowy więc się zatrzymał. Po włączeniu układu CSS (bit CSSON w RCC_CR) zachowanie uległo zmianie. Już samo dotknięcie ścieżki od rezonatora powoduje zadziałanie układu CSS. Objawia się to tym, że (obserwacje z debuggera):

- wyłączeniu ulega HSE i PLL (bity HSEON i PLLON się kasują)
- włączony zostaje oscylator wewnętrzny HSI (ustawienie bitu HSION)
- źródło zegara systemowego zostaje przestawione z PLL na HSI (bity SW się zerują)
- procesor wpada do „domysłnej” procedury obsługi wyjątku (*default handler*)

Pierwsze trzy kropki raczej nie powinny budzić zdziwienia. Układ CSS po prostu przełączył mikrokontroler na wewnętrzne źródło sygnału zegarowego. Zajmijmy się ostatnią kropką (zachowanie opisane w ostatniej kropce jest zależne od środowiska i ustawień projektu). *Default handler* wskazuje na to, że procesor próbował obsłużyć przerwanie dla którego nie ma napisanej ISR. Ale przecież w programie nie włączaliśmy żadnych przerwań, prawda? Prawda? Czyżby? Odsyłam oooo tu: rozdział 5.2, tabela 5.1, wyjątek NMI. W przypadku zadziałania CSS zgłaszane jest przerwanie NMI. Przerwanie nie maskowalne (po ludzku: nie wyłączalne - zawsze włączone). Wszystko pasuje! Procesor próbował obsłużyć przerwanie NMI, a my przecież nie napisaliśmy mu

²⁷² „Trust me, I'm an engineer!”

odpowiedniej ISR. Nadróbmy chybcikiem zaniedbania i dopiszmy taki o to arcydzieł do naszego kodu:

Procedura obsługi przerwania NMI (dioda świecąca na PB1):

```
1. __attribute__((interrupt)) void NMI_Handler(void){  
2.     gpio_pin_cfg(GPIOB, PB1, gpio_mode_output_PP_2MHz);  
3.     BB(GPIOB->ODR, PB1) = 1;  
4.     BB(RCC->CIR, RCC_CIR_CSSC) = 1;  
5. }
```

Arcydzieł robi dwie rzeczy: zapala diodę i zeruje flagę przerwania CSS. Dioda jest dla nas, żebyśmy wiedzieli co robi procesor. Flaga jest dla procesora, żeby mógł wyjść z przerwania. Efekt działania powyższego jest taki, że po uruchomieniu programu dioda (ta z SysTicka) migra co około sekundę. Zadziałanie układu CSS powoduje zapalenie diody z przerwania NMI i znaczne spowolnienie migania diody z SysTicka. Fanfary! Wszystko działa. CSS przełączył nam zegar na wewnętrzny, czyli częstotliwość taktowania mikrokontrolera spadła z 72MHz na „domyślne” 8MHz :) Przypominam że przerwanie NMI ma stały i bardzo wysoki priorytet, więc o żadnym wywłaszczeniu przez SysTicka nie ma nawet mowy! I jeszcze przypomnę na koniec, że zadziałanie bloku CSS z automatu aktywuje funkcję *break* liczników.

Uwagi różne różniste na koniec:

- po wybudzeniu mikrokontrolera z trybu standby lub stop, mikrokontroler przełącza się na źródło HSI
- przy zmianie zawartości pamięci flash (przez program) oscylator HSI musi być włączony
- mikrokontroler nie pozwoli wyłączyć źródła sygnału SYSCLK, nawet jeśli jest to źródło „pośrednie” - np. HSE przechodzące przez PLL
- mikrokontroler nie pozwoli wyłączyć źródła LSI, jeśli włączony został układ IWDG
- sygnał zegarowy układu peryferyjnego włącza się w rejestrach z sufiksem ENR (np. APB1ENR), w każdej chwili można wyłączyć taktowanie niepotrzebnych bloków, zawartość rejestrów konfiguracyjnych zostanie zachowana
- za pomocą rejestrów RCC_xxxRSTR można resetować bloki peryferyjne mikrokontrolera, po wpisaniu jedynki do odpowiedniego bitu zostają przywrócone domyślne wartości rejestrów konfiguracyjnych wybranego bloku (ustawiony bit nie kasuje się sprzętowo, trzeba to zrobić ręcznie)

Co warto zapamiętać z tego rozdziału:

- drzewko zegarowe Twoim przyjacielem tak jak schematy blokowe liczników
- konfiguracja zegarów wydaje się trudna tylko dlatego, że składa się z kilku etapów a początkujący boi się że coś zablokuje (jak w AVR)
- do niektórych liczników może dochodzić sygnał zegarowy o podwojonej częstotliwości („x2”)
- wszystkie układy peryferyjne mikrokontrolera dołączone są do szyn (AHB, APB1, APB2)
- rejestr RCC_xxxENR pozwalałą włączyć sygnał zegarowy peryferiala (xxx to nazwa szyny)
- sygnał zegarowy peryferiala można w każdej chwili wyłączyć aby np. ograniczyć pobór energii, nie powoduje to skasowania zawartości rejestrów konfiguracyjnych
- rejestr RCC_xxxRSTR umożliwiają zresetowanie peryferiala
- szyna APB1 ma częstotliwość ograniczoną do 36MHz, pozostałe do 72MHz
- układ CSS jest fajny i prosty a przerwania NMI zawsze włączone

17.3. System zegarowy (F429)

F429 generalnie rządzi się tymi samymi prawami co F103, tylko jest nieco bardziej rozbudowany. Popatrzmy na drzewko zegarowe F429. Dolna część jest związana z interfejsami USB OTG, Ethernet, LCD-TFT, Serial Audio (SAI), I2S. Tym się nie będziemy zajmować... bo to raczej tematy na osobne poradniki :) Więc cały dół drzewka możemy sobie odciąć i wywalić na razie... razem z dolnymi pętlami PLL (PLLI2S i PLLSAI). Od razu prościej prawda? Na górze nie ma nic nowego: LSI, LSE i wyjście sygnałów dla IWDG oraz RTC. No i dwa wyjścia sygnału zegarowego na zewnątrz mikrokontrolera (MCO1, MCO2). Został środek drzewka.

No to zacznijmy od HSI (16MHz) i HSE (4-26MHz). Oba sygnały są doprowadzone do multipleksera z którego wychodzi SYSCLK (180MHz maks.). Ponadto są doprowadzone do multipleksera z którego wychodzi sygnał wejściowy pętli PLL (nazwijmy go sobie f_M). Po drodze jest jeszcze dzielnik częstotliwości $[/M]$. Sygnał za tym dzielnicikiem częstotliwości ($f_{PLL\ in}$) musi mieć częstotliwość 1...2MHz. Bo tak i już! Przy czym w RMie zalecają aby trzymać się bardziej tych 2MHz bo to zmniejsza *jitter* PLL. Potem wchodzimy na PLL i tu się robi na chwilę fikuśnie. Będzie groźnie na pierwszy rzut oka, ale to jest tylko kilka prostych zasad... nie ma co panikować. Drzewko zegarowe przed nos i jedziemy:

- sygnał $f_{PLL\ in}$ (1...2MHz z dzielnika $[/M]$) przechodzi przez mnożnik $[xN]$ i bloczek VCO, wychodzi z tego sygnał f_{vco} o częstotliwości:

$$f_{vco} = f_{\text{PLL in}} \cdot N = f_m \cdot \frac{N}{M}$$

- pojawia się nowe ograniczenie: współczynniki N i M musimy dobrać tak aby:

$$192 \text{ MHz} \leq f_{vco} \leq 432 \text{ MHz}$$

- z pętli PLL wychodzą dwa sygnały:
 - PLLCLK (180MHz maks), który można wykorzystać jako zegar systemowy
 - PLL48CK (48MHz), który jest wykorzystywany przez bloki USB, SDIO, RNG²⁷³
- częstotliwości tych sygnałów są wyznaczane przez dwa preskalery (/P i /Q):

$$f_{\text{PLLCLK}} = f_{vco} \cdot \frac{1}{P}$$

$$f_{\text{PLL48CK}} = f_{vco} \cdot \frac{1}{Q}$$

I to cała filozofia. Dalej (za SYSCLK) jest już podobnie jak w F103.

W STM32F429 jest jeszcze jeden bajer o kosmicznej nazwie *Spread Spectrum Generator*. *Spread spectrum generator* to układ, który wprowadza drobne odchylenia częstotliwości wyjściowej PLL od domyślnej wartości. Powoduje to redukcję zakłóceń elektromagnetycznych... jakichś zakłóceń, sprzężeń czy czegoś tam takiego - jakichś niegodziwości. Do konfiguracji mamy trzy rzeczy:

- częstotliwość odchylania częstotliwości: f_{Mod} (do 10kHz)
- amplitudę zmian częstotliwości: md
- tryb zmian częstotliwości: *center* (symetryczne odchylanie częstotliwości w górę i w dół, dzięki czemu „średnio” nic się nie zmienia), *down* (odchylanie tylko w dół, żeby nie przekraczać wartości maksymalnej)

Częstotliwość oscylacji częstotliwości można sobie ustalić w RCC_SSCGR_MODPER. W datasheetie jest wzorek na wartość MODPER:

$$\text{MODPER} = \text{round} \left(\frac{f_{\text{PLL in}}}{4 \cdot f_{\text{Mod}}} \right)$$

gdzie:

²⁷³ USB wymaga zegara równo 48MHz zaś RNG i SDIO wymagają zegara o częstotliwości do 48MHz

- $f_{PLL\ in}$ - częstotliwość wejściowa PLL (za dzielnikiem M) [Hz]
- f_{Mod} - częstotliwość zaburzania częstotliwości zegara (do 10kHz) [Hz]

Drugi wzorek dotyczy amplitudy zmian częstotliwości²⁷⁴. Wartość jest konfigurowana w rejestrze RCC_SSCGR_INCSTEP:

$$INCSTEP = \text{round} \frac{(2^{15} - 1) \cdot md \cdot f_{VCO}}{100 \cdot 5 \cdot MODEPER}$$

gdzie:

- md - głębokość (amplituda) modulacji (od 0,25% do 2%) [%]
- f_{VCO} - częstotliwość zegara VCO (wewnątrz pętli PLL, patrz drzewo zegarowe) [MHz]

Przykład obliczeniowy:

- $f_{PLL\ in} = 1\text{MHz}$
- $f_{Mod} = 1\text{kHz}$
- $md = 2\%$
- $f_{VCO} = 240\text{MHz}$

$$MODEPER = \text{round} \frac{f_{PLL\ in}}{4 \cdot f_{Mod}} = \text{round} \frac{1\text{e}6}{4 \cdot 1\text{e}3} = \text{round} (250) = 250$$

$$INCSTEP = \text{round} \frac{(2^{15} - 1) \cdot md \cdot f_{VCO}}{100 \cdot 5 \cdot MODEPER} = \text{round} \frac{(2^{15} - 1) \cdot 2 \cdot 240}{500 \cdot 250} = \\ \text{round} (125,825) = 126$$

No to jedziemy:

Zadanie domowe 17.3: rozburzyć F429 na full (bez funkcji *over-drive* opisanej odrobinę w rozdziale 11.7). Tzn:

- HCLK - 168MHz
- PCLK1 - 42MHz
- PCLK2 - 84MHz

²⁷⁴ w datasheetcie, we wzorze, zamiast f_{VCO} jest PLLN czyli niby mnożnik PLL; ale potem w przykładzie obliczeniowym pod to PLLN jest podstawione 240MHz... więc chyba jednak chodzi o f_{VCO} ... także ten...

Nie zapomnieć o opóźnieniach flasha i ograniczeniach związanych z napięciem zasilania i *scalingiem*. (patrz rozdział 11.7). Standardowo: pełgający led na potwierdzenie :)

Przykładowe rozwiązanie (F429, dioda na PG13):

```

1. int main(void){
2.
3.     RCC->CR |= RCC_CR_HSEON | RCC_CR_HSEBYP;
4.     RCC->PLLCFGR = 4ul<<0 | 168ul<<6 | 7ul<<24 | RCC_PLLCFGR_PLLSRC_HSE | 1ul<<29;
5.     RCC->SSCGR = 500ul<<0 | 44ul<<13 | RCC_SSCGR_SSCEGEN;
6.     while (!(RCC->CR & RCC_CR_HSERDY));
7.     RCC->CR |= RCC_CR_PLLON;
8.     RCC->CFGGR = RCC_CFCGR_PPREG1_DIV4 | RCC_CFCGR_PPREG2_DIV2;
9.
10.    FLASH->ACR = FLASH_ACR_DCRST | FLASH_ACR_ICRST;
11.    FLASH->ACR = FLASH_ACR_DCEN | FLASH_ACR_ICEN | FLASH_ACR_PRFTEN | FLASH_ACR_LATENCY_5WS;
12.    while ((FLASH->ACR & FLASH_ACR_LATENCY) != FLASH_ACR_LATENCY_5WS);
13.
14.    while (!(RCC->CR & RCC_CR_PLLRDY));
15.    RCC->CFGGR |= RCC_CFCGR_SW_PLL;
16.    while ((RCC->CFGGR & RCC_CFCGR_SWS) != RCC_CFCGR_SWS_PLL);
17.    RCC->CR &= ~RCC_CR_HSION;
18.
19.    RCC->APB2ENR |= RCC_APB2ENR_SYSCFGEN;
20.    __DSB();
21.    SYSCFG->CMPCR = SYSCFG_CMPCR_CMP_PD;
22.    while (!(SYSCFG->CMPCR & SYSCFG_CMPCR_READY));
23.
24.    RCC->AHB1ENR |= RCC_AHB1ENR_GPIOGEN;
25.    __DSB();
26.    gpio_pin_cfg(GPIOG, PG13, gpio_mode_out_PP_LS);
27.
28.    SysTick_Config(168000000ul/8/2);
29.    SysTick->CTRL &= ~SysTick_CTRL_CLKSOURCE_Msk;
30.
31.    SCB->SCR |= SCB_SCR_SLEEPONEXIT_Msk;
32.    __WFI();
33.
34. }
35.
36. void SysTick_Handler(void){
37.     BB(GPIOG->ODR, PG13) ^= 1;
38. }
```

No to jedziemy od początku:

3) włączenie HSE. Uwaga! W domyślnej konfiguracji płytki Discovery, do głównego mikrokontrolera, doprowadzony jest sygnał zegarowy z programatora ST-Link. Mikrokontroler pracujący w ST-Linku ma włączone „wyprowadzenie” sygnału zegarowego (8MHz) na zewnątrz poprzez wyjście MCO (*Master Clock Output*). Sygnał ten przez rezystor R28 i mostek SB18 jest doprowadzony do wejścia HSE układu F429. Czyli, żeby nie było wątpliwości: oscylator HSE nie współpracuje z zamontowanym na płytce kwarcem (X3), tylko dostaje „gotowy” sygnał zegarowy z ST-Linka! W takiej sytuacji należy włączyć bit HSEBYP.

4) konfiguracja PLL:

- dzielnik M = 4 ($f_{PLL\ in} = HSE / M = 8MHz / 4 = 2MHz$)
- mnożnik N = 168 ($f_{VCO} = f_{PLL\ in} * N = 2MHz * 168 = 336MHz$)

- dzielnik Q = 7 ($f_{PLL48CK} = f_{VCO} / Q = 336\text{MHz} / 7 = 48\text{MHz}$)
- dzielnik P = 2^{275} ($f_{PLLCLK} = f_{VCO} / P = 336\text{MHz} / 2 = 168\text{MHz}$)
- ustawiam źródło sygnału PLL (HSE)
- ustawiam bit 29-ty rejestru... to są bity *Reserved* i jeśli już coś do nich wpisujemy to powinna być to wartość domyślna, tak się złożyło, że tu jest domyślnie jedynka (!)

5) konfiguracja *Rozmycia Częstotliwości*²⁷⁶ (*Spread Spectrum*), założylem sobie odchyłki o 1% z częstotliwością 1kHz²⁷⁷:

$$MODPER = \text{round} \frac{f_{PLL\text{ in}}}{4 \cdot f_{Mod}} = \text{round} \frac{2\text{e}6}{4 \cdot 1\text{e}3} = \text{round} (250) = 500$$

$$INCSTEP = \text{round} \frac{(2^{15} - 1) \cdot md \cdot f_{VCO}}{100 \cdot 5 \cdot MODEPER} = \text{round} \frac{(2^{15} - 1) \cdot 1 \cdot 336}{500 \cdot 500} = \text{round} (44,039) = 44$$

6) czekamy na HSE

7) włączam PLL, niech się rozpoczęda :)

8) konfiguracja preskalerów szyn

10) w F429, domyślnie, wszelkie bajery pamięci flash są wyłączone. Zaraz je włączymy przy okazji ustawiania opóźnień. Najpierw jednak (przed włączeniem) resetujemy bufory interfejsu pamięci flash bo nie wiadomo co w nich siedzi...

11) włączamy bajery flasha i ustawiamy opóźnienia zgodnie z tabelą z RMa (*Number of wait states according to CPU clock (HCLK) frequency...*)

12) RM zaleca odczytać nową wartość opóźnienia, żeby mieć pewność że konfiguracja zadziała... niech im będzie

14) czekamy na gotowość PLL

15, 16) zmieniamy źródło SYSCLK z HSI na PLL i czekamy na zakończenie tego procesu

17) wyłączamy HSI, żeby Nas jakiś Al Gore nie ścigał za niepotrzebne zużywanie energii

19 - 22) pamiętasz wzmiankę o *I/O Compensation Cell* (rozdział 3.8)? To dobry moment na włączenie tego mechanizmu.

275 to jest domyślna wartość, więc w rejestrze nic nie zmieniam

276 chodzi o rozmycie widma generowanych zakłóceń - odsyłam do wątku poświęconego Poradnikowi (<http://www.elektroda.pl/rtvforum/viewtopic.php?p=15126335>)

277 luz! nie przejmuj się, ja też nie mam bladego pojęcia jak się dobiera te wartości :)

Dalej już nie ma nic ciekawego... no może poza SLEEPONEXIT i brakiem pętelki głównej. Jak coś to odsyłam do rozdziału 11.7. Ok. Jest to trochę żmudne. Trzeba chwilę pokombinować żeby dobrać dzielniki itd. Ale czy jest w tym coś trudnego? No właśnie, też tak myślę.

Co warto zapamiętać z tego rozdziału:

- nic nie pamiętać tylko raz sobie to rozpracować a potem tylko Copy i Paste

17.4. System zegarowy (F334)

Standardowo otwieramy sobie drzewko zegarowe mikrokontrolera. F334 jest prawie tak prosty jak F103 jeśli chodzi o system zegarowy. Na pewno daleko mu do F429 :] Prześledzenie całego drzewka pozostawiam Ci, jako pracę domową. Na co warto zwrócić uwagę?

- sygnał zegarowy interfejsów USART jest dosyć elastyczny - wykorzystanie do taktowania interfejsu źródeł HSI lub LSE umożliwia pracę bloku (dotyczy tylko USART1) nawet gdy mikrokontroler jest uśpiony (USART1 może wybudzić układ z trybu *stop mode*)
- interfejs I2C1 również ma możliwość wybudzania mikrokontrolera z trybu uśpienia, w tym celu należy go „zasilić” ze źródła HSI
- ciekawą sprawą jest sygnał zegarowy dla przetworników ADC - sygnał może być pobierany sprzed bloku preskalera AHB, dzięki temu przetworniki mogą być taktowane z maksymalną prędkością (72MHz) niezależnie od ustawień reszty systemu (który można spowolnić np. w celu oszczędzania energii)
- zauważłeś już bloczek „x2” przed wyjściem zegara dla licznika TIM1 i HRTIM1? Tak tak, to nie pomyłka - te liczniki możemy zasilić z PLLCLK (72MHz max) x2, czyli mogą pracować z częstotliwością 144MHz!

No to wiadomo co :]

Zadanie domowe 17.4: przestawić źródło zegarowe na HSE²⁷⁸ i rozburzyć mikrokontroler do maksymalnych możliwych częstotliwości. Tzn:

- SYSCLK = 72MHz

278 w zestawie Nucleo nie ma zewnętrznego rezonatora (kwarca) dla „głównego” mikrokontrolera, do HSE dochodzi sygnał zegarowy (8MHz) z wyjścia MCO (*Master Clock Output*) mikrokontrolera ST-Linka

- HCLK = 72MHz
- PCLK1 = 36MHz
- PCLK2 = 72MHz

W celach testowych dorzucić miganie diodą w SysTicku (1Hz), aby upewnić się, że mikrokontroler rzeczywiście pracuje z założoną częstotliwością. Nie zapomnij o opóźnieniach w dostępie do flasha :)

Przykładowe rozwiążanie (F334, dioda na PA5):

```

1. int main(void){
2.
3.     RCC->CR |= RCC_CR_HSEON | RCC_CR_HSEBYP;
4.     while(~RCC->CR & RCC_CR_HSERDY);
5.
6.     RCC->CFGR = 0b0111 << 18 | RCC_CFGR_PLLSRC | RCC_CFGR_PPRE1_DIV2;
7.
8.     RCC->CR |= RCC_CR_PLLON;
9.     while(~RCC->CR & RCC_CR_PLLRDY);
10.
11.    FLASH->ACR = FLASH_ACR_PRFTBE | FLASH_ACR_LATENCY_1;
12.
13.    RCC->CFGR |= RCC_CFGR_SW_PLL;
14.    while((RCC->CFGR & RCC_CFGR_SWS) != RCC_CFGR_SWS_PLL);
15.
16.    RCC->CR &= ~RCC_CR_HSION;
17.
18.    RCC->AHBENR = RCC_AHBENR_GPIOAEN;
19.    gpio_pin_cfg(GPIOA, PA5, gpio_mode_out_PP_LS);
20.
21.    SysTick_Config(72000000/8);
22.    SysTick->CTRL &= ~SysTick_CTRL_CLKSOURCE_Msk;
23.    while(1);
24.
25. }
26.
27. void SysTick_Handler(void){
28.     GPIOA->ODR ^= PA5;
29. }
```

No i trzeci raz praktycznie to samo... co ja mam tu nowego napisać?

3, 4) włączam HSE i czekam na flagę gotowości, bit HSEBYP należy ustawić jeśli do oscylatora HSE doprowadzony jest „gotowy” sygnał zegarowy a nie kwarc - a tak właśnie jest w Nucleo

6) mnożnik PLL, dzielnik APB1, źródło taktowania PLL

8, 9) włączenie PLL i oczekiwanie na gotowość

11) jakiś tam bajer kontrolera flash (*prefetch*) i ustawienie opóźnień. O! Tu mam ciekawostkę - sprawdziłem co się stanie jeśli nie ustawię opóźnień flash - nic spektakularnego program się wykrzacał i tyle (po zatrzymaniu w debuggerze, wskaźnik programu był w malinowym chruśniaku).

13, 14) przestawiamy zegar systemowy na PLL i tak dalej...

Co warto zapamiętać z tego rozdziału:

- bezapelacyjnie wszystko!

18. HASHING RANDOMLY ENCRYPTED CRC (“*CONTRA FACTA NON VALENT ARGUMENTA*”²⁷⁹)

18.1. Wstęp z niespodzianką

Ten rozdział będzie dotyczył, mniej lub bardziej, czterech bloków mikrokontrolera:

- bloku szyfrującego - CRYP (*Cryptographic Processor*)
- bloku haszującego²⁸⁰ - HASH (*HASH Processor*)
- bloku obliczającego sumę kontrolną - CRC (*CRC Calculation Unit*)
- generatora liczb losowych - RNG (*Random Number Generator*)

W mikrokontrolerze F103, z wymienionych, występuje jedynie blok CRC. Dodatkowo działa on identycznie jak w F429. Z tego względu w tym rozdziale spuścimy na F103 zasłonę milczenia i skupimy się na bogatszym F429. No i na F334, gdyż w nim wszystko musi być inaczej. Przypominam, że informacje o tym jakie układy peryferyjne występują w danym mikrokontrolerze, można znaleźć w jego datasheetcie. Dla sportu sprawdź jak to jest z tytułowymi peryferialami w omawianych mikrokontrolerach (F103 i F429). Nie spiesz się ja poczekam, przecież nigdzie się nie wybieram -_-

Dlaczego wrzuciłem te układy do wspólnego worka? Zawsze wydawały mi się jakieś takie „zbliżone”, no i są dosyć proste - nie ma co mnożyć rozdziałów. Pierwsze dwa z nich (CRYP i HASH) to w ogóle przećwiczmy błyskawicznie. Wiesz dlaczego? Powinieneś wiedzieć!

Zadanie domowe 18.1: odpowiedz na pytanie: dlaczego przećwiczenie działania bloków CRYP i HASH pojedzie nam niezmiernie szybko?

Odpowiedź: Wbrew pozorom nie dlatego, że są proste a my jesteśmy genialni. Odpowiedź jest bardziej przyziemna i była już omawiana w poradniku (o [tu](#)). Mikrokontrolery, na których bazuje Poradnik, po prostu nie posiadają tych układów. Upewnić się o tym można przeglądać ich datasheets lub uważnie czytając reference manual. W RM0090 (dotyczy m.in. F429), na początku rozdziałów opisujących CRYP i HASH znajdują się uwagi podobnej treści:

„This section applies to STM32F415/417xx and STM32F43xxx devices.”

279 „Wobec faktów argumenty muszą ustąpić.”

280 jest w ogóle takie słowo?

Zresztą po co ja to mówię... na pewno to wiesz bo przecież przed chwilą miałeś, dla sportu, sprawdzić to samo samodzielnie :]

Nie ma i już, tyle w temacie. Przyznaję, sam się zdziwiłem. Nigdy nie wykorzystywałem tych peryferiali i jakoś nie przyszło mi do głowy, że taki wyczesany kontroler jak F429 może ich nie mieć na pokładzie. Po co w takim razie uwzględniłem je w tytule? A tak dla zmyłki. Żeby „boleśnie” przypomnieć o tym, o czym pisałem na początku Poradnika (o [tu](#) :) Jak się nie ma co się lubi, to się lubi co się ma. Pobawmy się tym co akurat mamy!

Co warto zapamiętać z tego rozdziału:

- rozdział w RMie o niczym nie świadczy, zawsze upewniaj się czy konkretny model mikrokontrolera posiada pożądany peryferial

18.2. Nic nie dzieje się przypadkiem (F429)

Nasz ulubiony mikrokontroler wyposażony jest w sprzętowy generator liczb *losowych* (o ile cokolwiek, w szczególności w elektronice, może być losowe). Czym to się różni od standardowej funkcji *rand()*? Tym, że ta funkcja generuje liczby *pseudo-losowe*, obliczone przez jakieś działanie matematyczne. Działa to tak, że na podstawie przyjętej wartości (zwanej *ziarnem*) funkcja wylicza kolejne liczby, które „udają” że są losowe. W rzeczywistości wynikają tylko i wyłącznie z przyjętego działania matematycznego i dla tej samej wartości ziarna, zawsze będą takie same. Także lipa a nie losowość :) Nie to co nasz peryferial.

Działanie bloku RNG opiera się o rejestr przesuwny ze sprzężeniem zwrotnym (LFSR²⁸¹, *Linear Feedback Shift Register*) do którego doprowadzany jest losowy śmiertnik generowany przez jakiś magiczny, szumiący układ analogowy (generator *ziarna*). ST w RMie chwali się, że układ przeszedł pewne testy, których nazwa pewnie mało komu cokolwiek mówi (FIPS PUB 140-2), i uzyskał przy tym wynik/wskaźnik/współczynnik (?) 99%... cokolwiek to jest. Mniejsza z tym.

Luźne marudzenie dla odprężenia: swoją drogą, jak można ocenić „losowość” generowanych liczb? Czy ciąg: 1, 5, 8, 0, 7, 2, 3, 5, 2, ... jest bardziej losowy od: 0, 0, 0, 0, 0, 0, 0, 0, ...? Skoro ma być losowy to chyba każda liczba ma jednakowe prawdopodobieństwa bycia wylosowaną. Hm? Mamy na sali probabilistyków/statystyków? Jak zmierzyć losowość? Jak matematycznie opisać to, co nie powinno wynikać z opisu matematycznego? </offtopic>

281 w Poradniku pojawiło się już to pojęcie - pamiętasz z jakiej okazji? :>

Z praktycznego (użytkowego) punktu widzenia sprawa przedstawia się następująco:

- układ RNG generuje losowe wartości o długości 32 bitów
- RNG taktowany jest swoim osobistym sygnałem zegarowym: RNG_CLK, sygnał ten wychodzi z układu PLL²⁸² (polecam zerknąć na drzewko zegarowe mikrokontrolera F429). Uwaga! To jest sygnał taktujący wewnętrzne bebechy bloku RNG a nie „interfejs” tego peryferiala. Za włączenie sygnału zegarowego dla interfejsu układu RNG, odpowiada bit RCC_AHB2ENR_RNGEN.
- układ powinien być taktowany sygnałem zegarowym o częstotliwości spełniającej zależność:

$$\frac{f_{HCLK}}{16} \leq f_{RNG_CLK} \leq 48 \text{ MHz}$$

- generowanie nowej liczby losowej zajmuje maksymalnie 40 cykli zegara RNG_CLK
- układ RNG ma możliwość generowania przerwania po wylosowaniu nowej liczby lub w przypadku wystąpienia błędu (brzmi groźnie)
- zgodnie z wymogami tego testu/normy/standardu (tego czegoś co mało komu cokolwiek mówi) pierwsza wylosowana liczba powinna zostać odrzucona, a każda kolejna powinna być porównywana z poprzednią. Jeżeli są równe to wynik testu (jakiego u diabła testu?) jest negatywny... i na tym kończą się dobre rady wujka eR-eM-aA. O tym co zrobić w przypadku oblania testu nie ma już ani słowa²⁸³.

Zatrzymajmy się na chwilę przy piątej kropce: „układ może generować przerwania w przypadku wystąpienia błędów”. Zapewne ciekawi Cię cóż to za paskudne błędy mogą nas niepokoić? Otóż blok RNG może wychwycić dwa rodzaje błędów:

- błąd związany z sygnałem zegarowym RNG_CLK - np. całkowity brak owego sygnału lub nader niska częstotliwość (patrz nierówność powyżej)
- błąd związany z generatorem ziarna (analogowym układem generującym szum podawany do rejestrów LFSR). Błąd ziarna sygnalizowany jest w dwóch przypadkach:

282 jak to dobrze, że rozdział o RCC mamy już za sobą i nie muszę kolejny raz pisać „wyjaśni się w rozdziale...”

283 to są rady z gatunku tych, po których człowiek wie jeszcze mniej niż przedtem. Tak mi się jakoś skojarzyło z dowcipem o zagubionych podróżnikach w balonie i matematyku, znasz?

- jeśli przynajmniej 64 kolejne bity ziarna będą miały tą samą wartość (0 boić 1)
- jeśli przynajmniej 32 kolejne bity ziarna będą miały przeciwnie wartości (0, 1, 0, 1, 0, ...)

W pierwszym przypadku ustawiana jest flaga RNG_SR_CECS (*Clock Error Current Status*). Jest to flaga tylko do odczytu, ustawiana i kasowana sprzętowo. Odzwierciedla ona aktualny stan błędu zegara (aktualny sygnał zegarowy jest poprawny lub nie). Każde ustawienie flagi CECS powoduje automatycznie ustawienie flagi przerwania CEIS (*Clock Error Interrupt Status*). Jak sama nazwa wskazuje, flaga ta jest odpowiedzialna za generowanie przerwania. Jest ona ustawiana sprzętowo (razem z CECS) i wywołuje przerwanie. Kasowanie flagi CEIS następuje ręcznie (w ISR sprawdzamy źródło przerwania i kasujemy flagę).

Po co ta druga flaga? Wyobraź sobie sytuację, w której błąd występuje tylko przez chwilę, np. wynika z przeprowadzanej rekonfiguracji systemu zegarowego. Flaga CECS zostaje ustawiona w momencie gdy sygnał zegarowy jest nieprawidłowy i jest automatycznie kasowana gdy sygnał wraca do „normy”. Jej automatyczne kasowanie powoduje, że nie zachowała się informacja o tym, że takowy błąd wystąpił. Jeśli przegapiliśmy to że flaga była ustawiona, to nie wiemy że jakikolwiek błąd wystąpił a wygenerowana liczba może nie być tak losowa jakbyśmy tego oczekiwali! Z pomocą przychodzi nam flaga CEIS która, po sprzętowym ustawieniu, wymaga ręcznego skasowania w programie.

W przypadku wystąpienia błędu zegara, praca bloku jest niemożliwa. Dokumentacja zaleca, aby w takiej sytuacji sprawdzić konfigurację układu zegarowego i skasować flagę CEIS. Błąd zegara nie ma wpływu na wygenerowane dotychczas liczby losowe, nie trzeba ich odrzucać.

Drugi rodzaj błędów (błędy ziarna) zachowuje się podobnie od strony użytkowej. Analogiczne jak przy błędzie zegara ustawiane są dwie flagi:

- SECS - *Seed Error Current Status (read only)*
- SEIS - *Seed Error Interrupt Status (read and clear by writing 0)*

Flaga SECS jest ustawiana i kasowana sprzętowo, odzwierciedla aktualny stan błędu. Flaga SEIS ustawiana jest razem z SECS, ale wymaga ręcznego skasowania w programie, np. w ISR.

Wykrycie błędu ziarna nie uniemożliwia dalszej pracy generatora. Rodzi jednak podejrzenia, że wygenerowane liczby nie są do końca losowe. Należy odrzucić wszystkie liczby wygenerowane gdy ustawiona była flaga SECS. Ponadto dokumentacja zaleca, w przypadku wykrycia tego rodzaju błędu, skasowanie flagi SEIS oraz ponowną inicjalizację (wyłączenie i ponowne włączenie) układu generatora liczb losowych.

Zadanie domowe 18.2: napisać prostą funkcję, która będzie zwracała losową liczbę 32b wygenerowaną przez RNG. Sprawdzić kilka wygenerowanych liczb, czy z grubsza „wyglądają losowo”. Na razie darujmy sobie obsługę błędów i porównywanie wygenerowanej wartości z poprzednią.

Przykładowe rozwiązanie (F429):

```
1. uint32_t rng(void){  
2.     while( !(RNG->SR & RNG_SR_DRDY) );  
3.     return RNG->DR;  
4. }  
5.  
6. int main(void){  
7.  
8.     RCC->CR |= RCC_CR_PLLON;  
9.     while( !(RCC->CR & RCC_CR_PLLRDY));  
10.  
11.    RCC->AHB2ENR = RCC_AHB2ENR_RNGEN;  
12.    RNG->CR = RNG_CR_RNGEN;  
13.  
14.    static uint32_t randoms[49000];  
15.    for (uint32_t i = 0; i < 49000; i++) randoms[i] = rng();  
16.  
17.    __BKPT();  
18.    while(1);  
19. }
```

Bułka z bananem, nieprawdaż?

8, 9) zegar bloku RNG_CLK wychodzi z pętli PLL, z tego względu wypadałoby włączyć pętlę. Po włączeniu pętli, analogicznie jak w poprzednim rozdziale, następuje oczekiwanie na jej gotowość (flaga RCC_CR_PLLRDY). A co z częstotliwością? W końcu układ RNG ma dosyć konkretne wymagania! A... Ty mi powiedz co z częstotliwością²⁸⁴?

11) włączenie sygnału zegarowego dla (interfejsu) układu RNG

12) włączenie generatora, w szczególności obejmuje to włączenie układu analogowego odpowiedzialnego za generowanie ziarna oraz rejestr LFSR

14) tablica na wylosowane liczby losowe

15) pętla zapełniająca tablicę; za generowanie liczb losowych odpowiada funkcja *uint32_t rng()*, patrz linijka 1

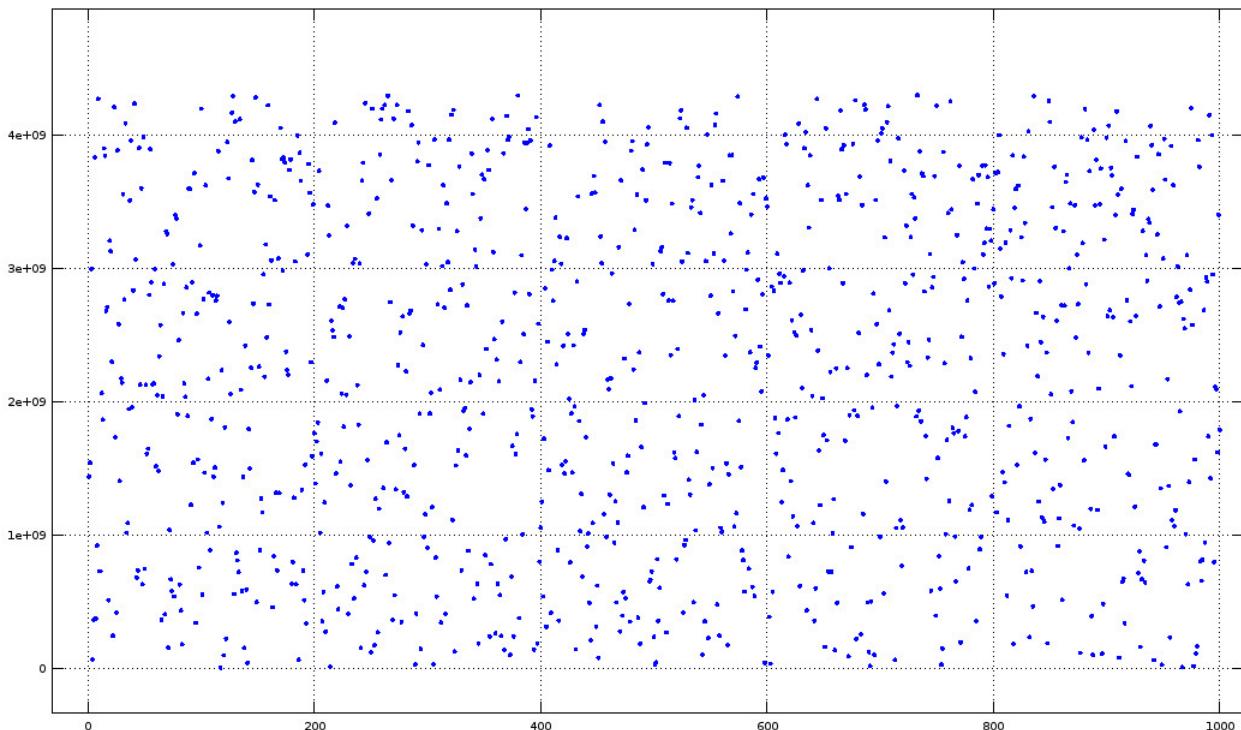
17) breakpoint przerywa działanie programu (np. w celu odczytania wygenerowanych liczb w debuggerze)

1 - 4) a oto i sedno tego przykładu - funkcja pobierająca wygenerowaną liczbę z bloku RNG. Za każdym razem, gdy peryferial wygeneruje liczbę losową, ustawiana jest flaga RNG_SR_DRDY. Flaga jest tylko do odczytu. Jest automatycznie kasowana przy odczytywaniu rejestrów danych RNG_DR. Po upewnieniu się, że flaga jest ustawiona (czyli że w rejestrze danych jest nowa liczba

²⁸⁴ i dlaczego wynosi ona 48MHz :)

losowa), odczytujemy wynik losowania z rejestru danych. Odczytanie rejestru DR powoduje automatyczne skasowanie flagi DRDY. Jeżeli kolejne wywołanie funkcji `rng()` nastąpi zbyt szybko (nie będzie jeszcze nowej liczby losowej), to procesor zatrzyma się na pętelce z linijki 2 i poczeka do końca najbliższego losowania.

Tak z ciekawości wrzuciłem kilka wyników losowania „na wykres”, jak uważasz są losowe?



Rys. 18.1 Wygenerowane wartości losowe

Trochę statystyki dla matematyków:

- liczba próbek: $N = 49000$
- wartość minimalna: $\min = 3537$
- wartość maksymalna: $\max = 4\ 294\ 921\ 572$
- średnia: $\mu = 2\ 157\ 849\ 288,59580$
- odchylenie standardowe: $\sigma = 1\ 238\ 416\ 768,22493$

no i jak? Jest losowo? Przyjmijmy że tak :]

Jeżeli ktoś chciałby wykorzystać tę funkcję do czegoś bardziej ważkiego aniżeli rysowanie śmieciowych wykresów, polecam dorobić kontrolę flag błędów zgodnie z zaleceniami dokumentacji.

Zadanie domowe 18.3: tym razem full-wypas. Sprawdzanie błędów, porównywanie wylosowanej wartości z poprzednią (powiedzmy, że będziemy powtarzać losowanie jeśli będą równe). Program ma zapisać tablicę 49000 losowo wygenerowanych liczb (32b). Ponadto niech będzie oparty na przerwaniach.

Przykładowe rozwiązanie (F429; wykorzystywane dwie diody: PG13 i PG14):

```

1. #define led1_bb BB(GPIOG->ODR, PG13)
2. #define led2_bb BB(GPIOG->ODR, PG14)
3.
4. static uint32_t randomsArray[49000];
5. static const uint32_t arraySize = sizeof(randomsArray) / sizeof(randomsArray[0]);
6.
7. int main(void){
8.
9.     RCC->CR |= RCC_CR_PLLON;
10.    while (!(RCC->CR & RCC_CR_PLLRDY));
11.
12.    RCC->AHB1ENR = RCC_AHB1ENR_CCMDATARAMEN | RCC_AHB1ENR_GPIOGEN;
13.    RCC->AHB2ENR = RCC_AHB2ENR_RNGEN;
14.
15.    gpio_pin_cfg(GPIOG, PG13, gpio_mode_out_PP_LS);
16.    gpio_pin_cfg(GPIOG, PG14, gpio_mode_out_PP_LS);
17.    led1_bb = 1;
18.
19.    NVIC_EnableIRQ(HASH_RNG_IRQn);
20.    RNG->CR = RNG_CR_IE | RNG_CR_RNGEN;
21.
22.    while(1);
23.
24. } /* main */
25.
26. void HASH_RNG_IRQHandler(void){
27.
28.     static uint32_t idx;
29.     static uint32_t rnd;
30.     static uint32_t matchFail, seedFail;
31.
32.     if(RNG->SR & RNG_SR_SEIS){
33.         seedFail++;
34.         RNG->SR = 0;
35.         RNG->CR = 0;
36.         RNG->CR = RNG_CR_IE | RNG_CR_RNGEN;
37.     } else if (RNG->SR & RNG_SR_CEIS){
38.         led1_bb = 1;
39.         led2_bb = 1;
40.         __BKPT();
41.         while(1);
42.     } else if(RNG->SR & RNG_SR_DRDY){
43.         rnd = RNG->DR;
44.         randomsArray[idx] = rnd;
45.
46.         if(idx && randomsArray[idx-1] == rnd) matchFail++;
47.         else idx++;
48.
49.         if(idx == arraySize) {
50.             led1_bb = 0;
51.             led2_bb = 1;
52.             __BKPT();
53.             while(1);
54.         }
55.     }
56. }
57. }
```

1, 2) dwie definicje bit bandowe do sterowania ledami (zawsze miło jak coś świeci i mruga)

4, 5) tablica na wygenerowane liczby i jej rozmiar w zmiennej pomocniczej dla wygody

7 - 24) funkcja główna jest bardzo podobna do poprzedniej. Na początku włączana jest pętla PLL, potem sygnały zegarowe wykorzystywanych bloków peryferyjnych (port GPIOG bo diody, generator RNG bo wiadomo, pamięć CCM RAM bo domyślnie jest włączona i nie chcemy jej wyłączać²⁸⁵). Następne linijki kodu to konfiguracja wyprowadzeń mikrokontrolera (nuda), zapalenie jednej diody (nuda), włączenie przerwania od układu RNG w kontrolerze NVIC (otwarcie NVICowych wrót... nuda) i na końcu włączenie generatora RNG tudzież generowania przezeń przerwań (dalej nuda). Na co w tej nudzie warto zwrócić uwagę?

Tym razem, po włączeniu zegara peryferiów nie ma rozkazu DSB (dla przypomnienia: [DSB w F4](#))! To tak dla odmiany. Przypominam, że DSB było stosowane tylko po to, aby uzyskać krótkie opóźnienie po włączeniu zegara. Rozkaz DSB jest wygodny, ale równie dobrze może to być jakiekolwiek inne działanie, które opóźni dostęp do rejestrów świeżo włączonego układu peryferyjnego. Pomiędzy włączeniem zegara interfejsu układu RNG (linijka 13) a pierwszym dostępem do rejestrów tego układu (linijka 20) upływa aż nadto czasu. Opóźnienie, wymagane po włączeniu peryferiala, wynosi tylko kilka cykli zegarowych.

Drugi wykorzystywany peryferial to port G. Od włączenie portu do dostępu do jego rejestrów, również upływa „sporo” czasu (włączenie zegara dla RNG, przygotowanie parametrów wywołania funkcji konfigurującej port, skok do funkcji). Co do pamięci CCM - była włączona już wcześniej (patrz stan początkowy rejestru RCC_AHB1ENR), więc o żadnych wymaganych opóźnieniach nie może być mowy :)

Kolejna ciekawostka to nazwa przerwania. Wektor przerwania jest wspólny dla układów RNG i HASH. Pomimo tego, że ten mikrokontroler nie ma układu HASH, nazwa przerwania uwzględnia obydwa bloki.

Bit RNG_CR_IE odpowiada za włączenie generowania przerwań przez układ RNG. Ustawienie tego bitu aktywuje wszystkie źródła przerwań (błąd zegara, błąd ziarna, wygenerowanie nowej wartości losowej). W ISR należy programowo zidentyfikować źródło przerwania.

26 - 57) ISR generatora RNG może wydawać się nieco skomplikowana, ale to tylko pozory. Cała procedura oparta jest o wyrażenie warunkowe (*if*), które sprawdza źródło przerwania.

Jeżeli przerwanie zostało wywołane przez błąd ziarna to spełniony jest warunek z linii 32. W takim wypadku, zgodnie z zaleceniami dokumentacji, kasujemy flagę przerwania (SEIS) i wyłączamy, a następnie ponownie włączamy, układ RNG. Dodatkowo w celach edukacyjnych dodałem licznik (*seedFail*) zliczający wystąpienia tego błędu.

Drugi z możliwych błędów to błąd sygnału zegarowego RNG_CLK. Jeżeli to on będzie źródłem przerwania, to spełniony będzie warunek z linii 37. W takim wypadku dalsza praca 285 jakby kogoś interesowało czemu zawsze tak pilnuję włączenia pamięci CCM: trzymam w niej stos

generatora nie jest możliwa. Program sygnalizuje stan awaryjny zapalając obie diody i przerywa pracę dzięki instrukcji *breakpoint*.

Ostatnie, i najbardziej pożądane, źródło przerwania to wygenerowanie nowej liczby losowej (flaga DRDY). Nowa liczba jest odczytywana i zapisywana w tablicy (*randomsArray*). Następnie jest porównywana z poprzednią. Jeżeli są równe to inkrementowany jest licznik *matchFail*. W przeciwnym wypadku inkrementowana jest zmienna przechowująca indeks nowego elementu tablicy. Dzięki temu, jeżeli wynik testu będzie negatywny, to następna wylosowana wartość nadpiszą aktualną która nie przeszła testu.

Warunek z linii 49 odpowiada za wykrycie zapełnienia całej tablicy. Jest to sygnalizowane jedną z diod. Ponadto wykonywanie programu jest przerywane instrukcją *breakpoint* - np. w celu odczytania zawartości tablicy w debuggerze.

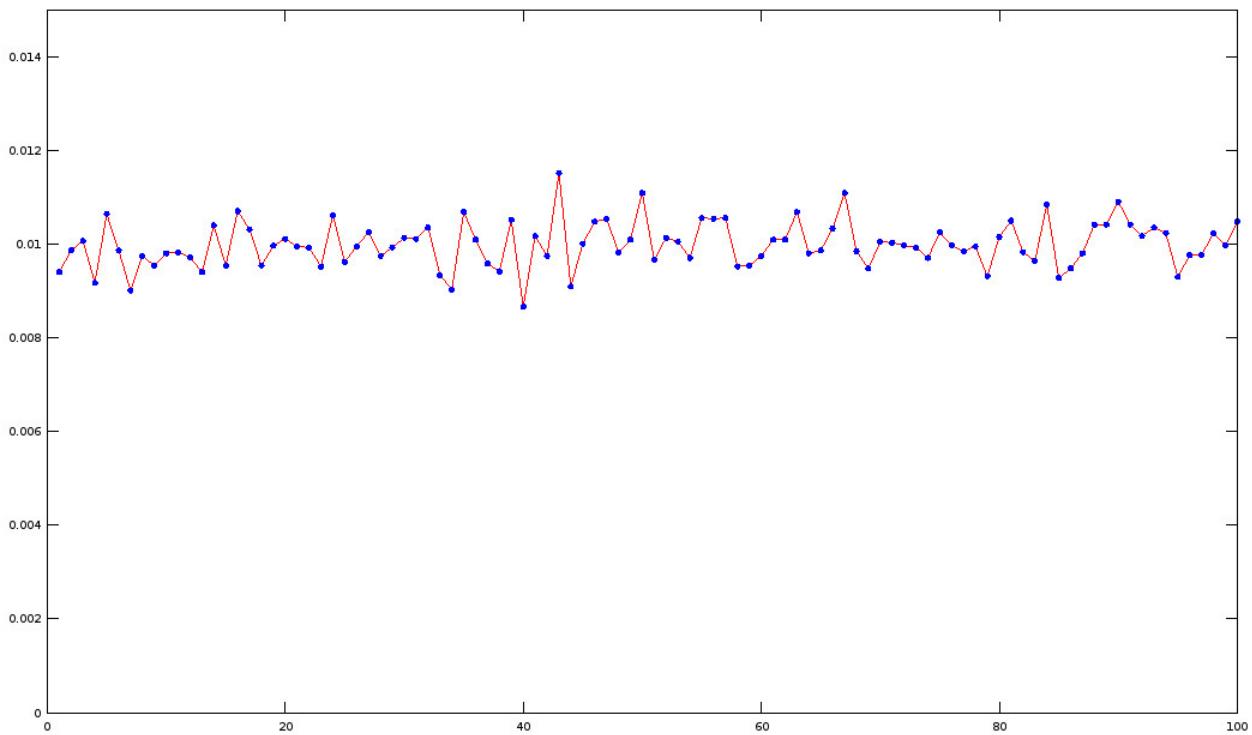
Pozostała jeszcze jedna mała sprawa na koniec. Być może zwróciłeś uwagę na to, że blok RNG generuje liczby 32 bitowe a nie zawsze takich potrzebujemy. A co jeśli potrzebna jest np. losowa liczba z zakresu np. od 1 do 100? Blok sprzętowy generuje tylko i wyłącznie liczby 32b, konwersji musimy więc dokonać na drodze programowej. I dobrze by było przy okazji nie popsuć „losowości” naszej liczby. Można wykorzystać do tego operację *modulo* (reszta z dzielenia, operator %). Miał być zakres od 1 do 100? No to proszę:

```
random_od_1_do_100 = (random_32b % 100) + 1;
```

i po bólu.

Tak w ramach zabaw około matematycznych pozwoliłem sobie wziąć wartości wygenerowane w zadaniu 18.2, „przeskalować je” do zakresu 1 - 100 (za pomocą *modulo*) i narysować rozkład prawdopodobieństwa²⁸⁶ wystąpienia każdej z wartości. Na chłopski rozum, skoro wygenerowana liczba jest losowa i z zakresu 1 - 100, to prawdopodobieństwo wystąpienia jakiejkolwiek wartości (całkowitej) z tego zakresu powinno wynosić 0,01... Uwaga, uwaga! Bęben maszyny losującej jest pusty. Następuje zwolnienie blokady... i:

286 chyba tak to się nazywa :)



Rys. 18.2. Rozkład prawdopodobieństwa wartości z zakresu od 1 do 100

Dziwnym trafem wyszło co miało wyjść :) I tym optymistycznym akcentem kończymy z RNG.

Co warto zapamiętać z tego rozdziału:

- mikrokontroler posiada wbudowany układ generujący 32 bitowe liczby losowe (a nie pseudo-losowe)
- jeżeli potrzebujemy „krótsze” liczby to *modulo*

18.3. Suma kontrolna CRC32 (F103 i F429)

Każdy chyba mniej więcej wie (lub czuje), czym jest CRC. A jak nie, to STFW. Tu i teraz liczy się tylko to, że nasze ulubione mikrokontrolery posiadają sprzętowy moduł, który po nakarmieniu danymi, *oddaje* (może w tym miejscu porzućmy analogię do układu pokarmowego) wartość kodu nadmiarowego obliczonego dla tych danych.

Oczywiście nie może być zbyt różowo. W mikrokontrolerach F103 i F429 układ CRC jest dosyć siermiążny. Możliwości jego konfiguracji ograniczono do niezbędnego minimum technicznego (nazywając rzecz po imieniu - do zera), ponadto cechuje się on kilkoma nowatorskimi rozwiązaniami. Efekt tego jest taki, że w Internecie można trafić na wypowiedzi programistów

słabej wiary, którzy zwańpili w jakąkolwiek przydatność tego układu i wolą liczyć CRC programowo. Ich wybór.

Najważniejsze cechy (... wady) układu CRC, to:

- układ przyjmuje tylko i wyłącznie dane o rozmiarze jednego słowa (32 bity)
- układ posiada jeden rejestr danych - służy on zarówno do podawania danych z których ma być policzone CRC jak i do odczytania wartości sumy kontrolnej²⁸⁷
- operacja zapisu do rejestru danych jest blokowana na czas obliczania nowej wartości CRC
- początkowa wartość rejestru danych, po resecie bloku, wynosi 0xFFFF FFFF
- wykorzystywany jest tylko i wyłącznie wielomian CRC-32 (0x4C11DB7)
- przeliczenie CRC po podaniu nowej wartości trwa 4 cykle zegara AHB
- układ posiada dodatkowy 8 bitowy rejestr (CRC_IDR) do dowolnego wykorzystania w programie (nie ma on żadnych „sprzętowych” podtekstów)²⁸⁸

Wykorzystanie bloku jest bardzo proste w teorii. Wystarczy go włączyć i zresetować odpowiednim bitem. Potem należy wrzucać kolejne porcje danych do rejestru CRC_DR, a na końcu odczytać z niego sumę kontrolną. Tyle. Łatwo prosto i przyjemnie. Wyjątkowo, jest to układ który nie może generować przerwań. Układ CRC można wykorzystać np. do kontroli poprawności transmisji między dwoma STMami. I generalnie wszystko będzie działać prawidłowo, bez żadnego cudowania. Z małym *ale*.

Kłopotki zaczynają się wtedy, kiedy spróbujemy wartość CRC-32 z STMa porównać z wartością kontrolną (dla tych samych danych) obliczoną przez cokolwiek innego... np. przez kalkulator online. Wartości będą różne! I tu jest właśnie pies pogrzebion. Wielu zacnych mężów poddało się, próbując doprowadzić do równości między tym co oblicza STM a tym co podaje 75% pozostałych programów liczących CRC-32. Ja z kolei mam problem pedagogiczny. W tym miejscu powinno być zadanie z gatunku „napisz program, który będzie liczył CRC tak jak powinien”. Kłopot w tym, że sam bym tego z głowy nie napisał, więc nie zamierzam wymagać od Ciebie. Nie raz wspominałem, że trzeba umieć poszukiwać informacji :] Działające rozwiązanie znalazłem gdzieś w Internecie, na jakimś forum. Tylko je trochę uporządkowałem po swojemu. Tak czy siak... zadanie musi być :)

287 jak rejestr danych UART który obsługuje zarówno dane wysypane jak i odbierane

288 bladego pojęcia nie mam po co tam ten rejestr, ni wypiął ni przypiął... przydatne jak WiFi w lodówce

Zadanie domowe 18.4: korzystając ze wszelkich dostępnych Ci źródeł napisz funkcję, która będzie zwracała wartość CRC-32 danych podanych w argumencie. Funkcja ma przyjmować dwa argumenty: wskaźnik na początek bufora z danymi i wielkość bufora (w bajtach!). Funkcja ma działać poprawnie jeśli wielkość bufora nie będzie wielokrotnością 4B. Oczywiście ma, generalnie, wykorzystywać sprzętowy układ CRC mikrokontrolera. Owocnych poszukiwań! Przypominam o „zasadzie trzydniowej” (trzy dni samodzielnych prób i poszukiwań przed obejrzeniem przykładowego rozwiązania). A! Poniżej lista kilku kalkulatorów CRC online. Naszym celem jest, aby funkcja uruchomiona w STMie zwracała to samo co one (patrz tabela 18.1)

- <http://www.lammertbies.nl/comm/info/crc-calculation.html>
- www.sunshine2k.de/coding/javascript/crc/crc_js.html
- https://www.tools4noobs.com/online_php_functions/crc32/
- <http://www.tahapaksu.com/crc/>
- <http://www.zorc.breitbandkatze.de/crc.html>

Tabela 18.1. Przykładowe wartości CRC-32 dla wybranych łańcuchów znakowych

Ciąg znaków	CRC-32
a	0xE8B7BE43
abcde	0x8587D865
<i>Lorem ipsum dolor sit amet, consectetur adipiscing elit.</i>	0x29BA2932
<i>Smile - they said - it could be worse. So i did and it was.</i>	0x749BDF80

Przykładowe rozwiążanie (F429)²⁸⁹:

```
1. static const char *data = "Smile - they said - it could be worse. So i did and it was.";
2.
3. uint32_t count_crc(const void *input, uint32_t bytes) {
4.     CRC->CR = CRC_CR_RESET;
5.     __DSB();
6.
7.     while (bytes >= 4) {
8.         CRC->DR = __RBIT(*((const uint32_t*) input));
9.         input += 4;
10.        bytes -= 4;
11.    }
12.
13.    uint32_t crc = __RBIT(CRC->DR);
14.
15.    while (bytes--) {
16.        crc ^= (uint32_t) *((const char*) input++);
17.
18.        for (uint32_t j = 0; j < 8; j++) {
19.            if (crc & 1) crc = (crc >> 1) ^ 0xEDB88320;
20.            else crc >>= 1;
21.        }
22.    }
23.
24.    return ~crc;
25. }
26.
27. int main(void) {
28.
29.     RCC->AHB1ENR = RCC_AHB1ENR_CCMDATARAMEN | RCC_AHB1ENR_CRCEN;
30.
31.     static uint32_t crc;
32.     crc = count_crc(data, strlen(data));
33.
34.     __BKPT();
35.     while (1);
36.
37. } /* main */
```

No i tak to właśnie wygląda. Funkcja główna (linijki 27 - 37) jest prościutka jak cała matematyka i nie ma czego w niej omawiać. Cały cymes siedzi w funkcji *count_crc()*. Przyjrzyjmy się tej funkcji.

4 - 5) pierwszy krok to zresetowanie kalkulatora CRC. Służy do tego bit CRC_CR_RESET. Jest to bit tylko do zapisu. Po zakończeniu resetowania jest kasowany sprzętowo. Profilaktycznie dorzucam krótkie opóźnienie oparte o rozkaz barierowy DSB.

7 - 11) ta pętla odpowiada za nakarmienie układu CRC danymi o długości 32b (4B, jedno słowo). Dopóki mamy odpowiednią ilość danych w buforze, to są one wpisywane kolejno do rejestru danych CRC_DR. Przypominam, że operacja zapisu jest blokowana aż nowe CRC zostanie policzone, w związku z tym nie są wymagane żadne opóźnienia po zapisie (nic się nie nadpisze). Funkcja *__RBIT()* to jedna z funkcji *intrinsic*²⁹⁰, czyli prosta wstawka asemblerowa umożliwiająca użycie konkretnego rozkazu procesora. W tym wypadku jest to rozkaz *rbit (reverse bit order)*, który odwraca kolejność bitów w słowie. Dlaczego należy odwracać bity danych bloku CRC? Dobre pytanie... nie mam pojęcia. W Internecie też nie znalazłem zadowalającej mnie odpowiedzi. Tak czy

289 silnie wzorowane na wątku [CRC computation](#) z forum ST, w szczególności na poście *clive1*

290 jest zdefiniowana w pliku *core_cmInstr.h*

siak - to rozwiązanie działa. Jakby ktoś wy-googlował coś ciekawego na ten temat to wiadomo gdzie mnie szukać :)

Po wyjściu z pętli zakończyliśmy wpisywanie do układu CRC danych o rozmiarze jednego słowa. I teraz możliwe są dwa scenariusze:

- ilość danych w buforze jest wielokrotnością 4B i nic więcej nie zostało
- ilość danych w buforze nie jest wielokrotnością 4B i coś jeszcze zostało (1, 2 lub 3 bajty)

Tak jak wspomniałem na początku rozdziału, układ CRC przyjmuje tylko i wyłączenie dane 32b. Jeżeli więc został nam np. 1B to jesteśmy w kropce. Wpisanie jednego bajtu do rejestru CRC_DR spowoduje, że policzona zostanie suma z tego bajtu dopełnionej zerami do wymaganych 4B. A nie koniecznie o to nam chodzi. W związku z tym ta „reszta” zostaje doliczona na piechotę (programowo) w pętli z linii 15 - 22. Na koniec wartość CRC jest zwracana po uprzednim odwróceniu wartości bitów (nie wiem, nie pytać; grunt, że działa).

W wyniku działania powyższego programu, obliczona zostaje wartość CRC-32 ciągu znaków „*Smile - they said - it could be worse. So i did and it was.*”, wynik wynosi (odczytany debuggerem): 0x749bdf80. Otrzymana wartość jest zgodna z tą z tabelki 18.1 (hura!). Jak ktoś nie wierzy tabelce to niech się pobawi kalkulatorami CRC online (np. z linków podanych dwie strony temu - te kalkulatory sprawdziłem).

CRC można w szczególności wykorzystać do weryfikowania poprawności danych w pamięci flash mikrokontrolera. Np. aby zmniejszyć ryzyko nieprawidłowego działania urządzenia w przypadku uszkodzenia zawartości pamięci programu lub w celu kontroli nowego wsadu wgrywanego poprzez bootloader. O ile policzenie sumy kontrolnej po stronie mikrokontrolera nie stanowi problemu, o tyle zautomatyzowanie procesu dodawania CRC do wsadu po stronie PC może okazać się nieco bardziej „tricky”.

Najprostsze rozwiązanie to obliczanie (zewnętrznym programem) wartości CRC z gotowego wsadu (pliku *.bin*) i bezczelne doklejanie jej na końcu tegoż wsadu. W programie STMowym wystarczy potem odczytać ostatnie zapisane słowo z pamięci flash i porównać z obliczoną wartością CRC. Wszystko fajnie, ale to zadziała tylko dla prostego pliku *.bin*. A co z *.hex* i *.elf*? Te formaty są bardziej złożone i nie można ot tak dokleić do nich wartości CRC. W przypadku pliku *.hex* można posłużyć się narzędziem *SRecord*. Jakiś sposób na plik *.elf* pewnie też się znajdzie. Ale to rozwiązanie wydaje mi się mało eleganckie :) Trzeba to zrobić jakoś mądrzej! Przedstawię to rozwiązanie, które mnie wydaje się najsympatyczniejsze.

Uwaga! Dalsza część tego rozdziału ma niewiele wspólnego z mikrokontrolerami STM32 jako takimi. To taki temat trochę poboczny. Koncentruje się bardziej na zagadnieniach związanych z wykorzystywanymi narzędziami (GCC), ich konfiguracją (szablon Freddiego Chopina) oraz systemem operacyjnym którego używam (pochodna Ubuntu). Ponadto przedstawione rozwiązanie zostało wymyślone na szybko, na potrzeby przykładu, na pewno można to zrobić bardziej elegancko i uniwersalnie. Tak czy siak zachęcam do lektury, może okaże się inspirująca :)

Proces budowania projektu wygląda (w uproszczeniu) tak, że najpierw komplikowane są poszczególne pliki źródłowe - powstają pliki obiektowe. Linker łączy pliki obiektowe tworząc plik wynikowy w formacie *.elf*. Następnie z pliku *.elf* tworzone są pliki *.bin* i *.hex*. Jeśli więc umieścimy wartość CRC już w pliku *.elf* (przy linkowaniu) to w naturalny sposób będzie ona obecna w plikach pochodnych *.bin* i *.hex*. O to nam właśnie chodzi.

Nie wdając się w szczegóły (jak ktoś jest zainteresowany to dokumentacja linkera jest ogólnie dostępna i wbrew pozorom nie gryzie bardzo :)) plan jest taki:

- w skrypcie linkera wprowadzimy następujące modyfikacje:
 - na początku sekcji *.text* dodamy nowy symbol, który będzie oznaczał początek danych objętych sumą CRC (*__crc_start*)
 - kontrolą CRC objęta będzie sekcja *.text* oraz *.data*²⁹¹
 - tuż za sekcją *.data* dodamy nową sekcję *.text.crc*²⁹²
 - w nowej sekcji zarezerwujemy 4B na naszą sumę CRC oraz zdefiniujemy symbol oznaczający koniec obszaru objętego kontrolą CRC (*__crc_end*)
 - dodamy symbol określający wielkość obszaru pamięci objętego kontrolą CRC (*__crc_size*)
- podczas linkowania, za pomocą opcji *defsym*, „zapiszemy wartość” sumy kontrolnej wsadu do zarezerwowanego wcześniej obszaru w sekcji *.text.crc*

Tak jak wspomniałem, wszystko zależy od wykorzystywanych narzędzi i ich konfiguracji. W moim przypadku, realizacja pierwszej kropki, przedstawia się następująco:

²⁹¹ zakładam, że sekcja *.data* znajduje się tuż po sekcji *.text*, jeżeli pomiędzy nimi leży np. stos to ten sposób nie zadziała!

²⁹² człon *.text* w nazwie sekcji powoduje, że program *size* będzie doliczał jej rozmiar do „normalnej” sekcji *.text*

Zmodyfikowany fragment skryptu linkera (dopisane nowości zaznaczono na czerwono):

```
1. .text :
2. {
3.     . = ALIGN(4);
4.     __text_start = .;
5.     PROVIDE(__text_start = __text_start);
6.     PROVIDE(__crc_start = __text_start);
7.
8.     [...]
9.
10. } > rom AT > rom
11.
12. [...]
13.
14. .data :
15. {
16.
17.     [...]
18.
19. } > ram AT > rom
20.
21. .text.crc :
22. {
23.     . = ALIGN(4);
24.     LONG(CRC32);
25.     . = ALIGN(4);
26.     __crc_end = .;
27.     PROVIDE(__crc_end = __crc_end);
28. } > rom AT > rom
29.
30. .bss :
31. {
32.
33.     [...]
34.
35. } > ram AT > ram
36.
37. [...]
38.
39. PROVIDE(__crc_size = __crc_end - __crc_start);
```

Wyrażenie *LONG* odpowiada za zarezerwowanie 4B pamięci na symbol *CRC32*. Prościzna.

Z drugą kropką jest nieco gorzej. Dlaczego? Ano: chcemy przy linkowaniu podać wartość CRC. Musimy więc ją wcześniej policzyć, to oczywiste. Kłopot polega na tym, że przed linkowaniem nie mamy pliku z wsadem. Jak więc policzyć CRC czegoś co jeszcze nie istnieje? Nie da się! Rozwiążanie ma dosyć mało polotu... Rozbijemy linkowanie na dwa razy. Najpierw zlinkujemy projekt bez podawania wartości CRC (tzn. w opcji *defsym* podamy jakąkolwiek wartość bo jakąś musimy). Potem wygenerujemy plik *.bin* z wsadem i policzymy CRC tegoż wsadu. A na koniec zlinkujemy całość jeszcze raz! Tym razem podając już prawidłową wartość CRC. Prawda, że proste :) Zmiany w kawałku *makefile'a*, odpowiedzialnym za linkowanie, przedstawiają się następująco:

Zmodyfikowany fragment pliku *makefile*:

```
1. CRC = ${OUT_DIR_F}${PROJECT}_crc.bin
2. ...
3. $(ELF) : $(OBJS)
4.     @echo 'Linking target: $(ELF)'
5.     $(CXX) -Wl,--defsym=CRC32=0 $(LD_FLAGS_F) $(OBJS) $(LIBS) -o $@
6.     $(OBJCOPY) -O binary $@ $(CRC)
7.     truncate --size=-4 $(CRC)
8.     $(CXX) -Wl,--defsym=CRC32=0x`crc32 $(CRC)` $(LD_FLAGS_F) $(OBJS) $(LIBS) -o $@
9.     @echo '
```

5) to jest pierwsze linkowanie, opcja *defsym* zapisuje wartość CRC32 - na razie równą 0, generowany jest plik *.elf*

6) za pomocą programu *objcopy* tworzony jest plik binarny z wsadem. Za ścieżkę i nazwę pliku odpowiada zmieniona *CRC*

7) tu jest mała ciekawostka. Utworzony powyżej plik z wsadem, zawiera na końcu 4B przeznaczone na wartość CRC. Aktualnie ten kawałek pamięci jest wypełniony zerami (patrz linkowanie z linii 5). Jeśli policzymy CRC z całego wsadu, to te zera wpłyną na otrzymaną wartość. A tego nie chcemy. W taki czy inny sposób trzeba policzyć CRC bez tych końcowych 4B. Ja wykorzystałem program *truncate*, który pozwala m.in. wywalić kawałek pliku (np. 4 ostatnie bajty).

8) drugie linkowanie, formułka przy *defsym* trochę się rozrosła. Wyrażenie między „apostrofami”²⁹³ to wywołanie programu *crc32* dla pliku ze zmiennej *CRC*. Jak łatwo odgadnąć, program liczy wartość CRC32 podanego pliku. Do tego doklejany jest prefiks „0x” oznaczający wartość wyrażoną w systemie szesnastkowym. I viola!

To linkowanie na dwie raty trochę mi się nie podoba, ale: „jeśli coś wygląda głupio, ale działa, to nie jest głupie”! A tak się składa, że to działa! Po skompilowaniu projektu możemy podejrzeć sobie plik *.map* i cieszyć się naszym dziełem:

Fragment pliku *.map* (wartość CRC to 0xE9E6 37F0, umieszczona pod adresem 0x0800 0300):

```
1. 0x08000300          . = ALIGN (0x4)
2. 0x08000300          __text_crc32 =
3. [!provide]          PROVIDE (__text_crc32, __text_crc32)
4. 0x08000300          0x4 LONG 0xe9e637f0 CRC32
```

A jak wykorzystać to w programie? Poniżej prościki przykładzik z wykorzystaniem funkcji z poprzedniego zadania:

293 uwaga! to nie są apostrofy, ten znak nazywa się *grawis* - to są te robaczki z dolnej części klawisza z tylda, a nie z klawisza z cudzysłowem (ma być ` a nie ')

Przykładowy program weryfikujący CRC wsadu:

```
1. int main(void) {
2.
3.     RCC->AHB1ENR = RCC_AHB1ENR_CCMDATARAMEN | RCC_AHB1ENR_CRCEN;
4.
5.     static uint32_t crc;
6.
7.     extern const uint32_t __crc_start;
8.     extern const uint32_t __crc_size;
9.
10.    crc = count_crc(&__crc_start, (uint32_t)&__crc_size);
11.
12.    if( crc == 0x2144df1c ){
13.        //crc ok
14.    } else {
15.        //crc nie ok
16.    }
17.
18. } /* main */
19.
```

W programie wykorzystywane są dwa, dodane przez nas, symbole ze skryptu linkera (`__crc_start` oraz `__crc_size`). Zwróć szczególną uwagę na sposób odwoływania się do nich w programie w C (operator `&`). Łatwo o pomyłkę. Obliczana jest CRC z obszaru pamięci od `__crc_start` (początek sekcji `.text`) o wielkości `__crc_size` (czyli do końca sekcji `.data` **wraz z wartością CRC** w naszej sekcji `.text.crc`).

Na końcu jest odrobina magii (nie mylić z przyprawą *maggi*). CRC ma taką sympatyczną właściwość, że: jeśli policzymy CRC jakichś danych a następnie dokleimy tą CRC do tych danych i jeszcze raz policzymy CRC z całości - to otrzymamy stałą „magiczną” wartość. W przypadku algorytmu CRC32 ta wartość wynosi 0x2144 DF1C. Nie musimy więc odczytywać wartości CRC z wsadu i porównywać z tą obliczoną przez mikrokontroler. Wystarczy policzyć CRC z całego wsadu (razem z zapisaną na końcu wartością CRC) i sprawdzić czy wynik wynosi magiczne 0x2144 DF1C.

Ponadto w programie można poprawić jeszcze kilka drobiazgów:

- wsad pamięci flash powinien być wielokrotnością 4B, więc można zrezygnować z kawałka funkcji liczącej CRC odpowiedzialnego za „doliczenie” pozostałego jednego, dwóch lub trzech bajtów w sposób programowy
- do karmienia bloku CRC można wykorzystać naszego cichego pomocnika - DMA (tylko, że wtedy nie będziemy mogli odwracać kolejności bitów, więc uzyskamy inną wartość CRC niż z większości kalkulatorów!)

Co warto zapamiętać z tego rozdziału:

- układ CRC w mikrokontrolerach F103 i F429, choć nieco upierdliwy, daje się zmusić do działania
- dodatkowe informacje nt. CRC można znaleźć w nocy aplikacyjnej: AN4187 *Using the CRC peripheral in the STM32 family*

18.4. Sprzętowe CRC8, CRC16 i dowolny wielomian (F103 i F429)

Patrz rozdział 19.5 :)

Co warto zapamiętać z tego rozdziału:

- bez żartów

18.5. Suma kontrolna CRC (F334)

Sprzętowe CRC w F3 ma wszystko to, czego nam wcześniej brakowało. Miłą niespodzianką jest to, że dodatkowe funkcje nie skomplikowały obsługi. A cóż ciekawego przybyło? Ano:

- układ można karmić danymi 8, 16 lub 32 bitowymi; o wielkości „danej” decyduje zapis do rejestru danych (zaraz się wyjaśni w praniu)
- za pomocą rejestru CRC_INIT możemy sobie ustawić dowolną wartość początkową kalkulatora CRC, domyślnie wynosi ona 0xFFFF FFFF
- za pomocą rejestru CRC_POL możemy ustawić dowolny wielomian CRC, tzn. prawie dowolny bo wielomian może mieć rozmiar 7, 8, 16 lub 32 bity (wybierany za pomocą CRC_CR_POLYSIZE); domyślnie wybrany jest wielomian 0x4C11DB7 (taki jak w F103 i F429)
- możemy skonfigurować odwracanie bitów danych wpisywanych i odbieranych z bloku CRC (pole CRC_CR_REV_IN i CRC_CR_REV_OUT) wedle potrzeb

No to tyle nowości. Tak na dobrą sprawę możemy wziąć funkcję liczącą CRC z poprzedniego rozdziału i będzie ona działać poprawnie w F334. Ale skoro mamy nowe możliwości, to warto ich użyć zamiast męczyć się na piechotę.

Zadanie domowe 18.5: wykorzystując nowe funkcje bloku CRC dostępne w F334, uprosić funkcję *count_crc()* z poprzedniego rozdziału.

Przykładowe rozwiązanie (F334):

```
1. uint32_t count_crc(const void *input, uint32_t bytes){  
2.  
3.     CRC->CR = CRC_CR_RESET | CRC_CR_REV_OUT | CRC_CR_REV_IN_0 | CRC_CR_REV_IN_1;  
4.     __DSB();  
5.  
6.     while (bytes >= 4) {  
7.         CRC->DR = *(const uint32_t*)input;  
8.         input += 4;  
9.         bytes -= 4;  
10.    }  
11.  
12.    while(bytes--) *(volatile uint8_t *)&CRC->DR = *(const uint8_t *)input++;  
13.    return ~CRC->DR;  
14.}  
15.  
16.  
17. static const char data[] = "Lorem ipsum dolor sit amet, consectetur adipiscing elit.xx";  
18.  
19. int main(void){  
20.     RCC->AHBENR = RCC_AHBENR_CRCEN;  
21.  
22.     static uint32_t crc;  
23.     crc = count_crc(data, strlen(data));  
24.  
25.     __BKPT();  
26.     while (1);  
27. }
```

3) w poprzedniej wersji funkcji, w tym miejscu był tylko reset kalkulatora CRC. Peryferial w F334 ma jednak możliwość sprzętowego odwracania bitów. Warto z tego skorzystać - dzięki temu odpadą nam konieczność programowego odwracania np. za pomocą rozkazu *rbit*

6 - 10) tutaj nic się nie zmieniło poza tym, że ubył *rbit*

12) w F103 i F429 nie było możliwość liczenia sumy z danych o rozmiarze różnym od 32 bitów. Z tego względu, jeśli ilość danych w buforze nie była wielokrotnością słowa, to resztę (1, 2 lub 3B) doliczaliśmy już na piechotę.

CRC w F334 pozbawione jest tego ograniczenia. Dane mogą mieć rozmiar 1B, 2B lub 4B. Nie ma więc potrzeby programowego doliczania końcówki, może to zrobić blok CRC. O rozmiarze danej decyduje zapis do rejestru CRC_DR. W powyższym przykładzie założyłem, że po wysłaniu wszystkich danych 4 bajtowych, resztę doliczę już po jednym bajcie. W związku z tym muszę zadbać o to, aby wygenerowany kod odwoływał się do CRC_DR z wykorzystaniem rozkazów operujących na danych 8-mio bitowych. O tym jaki rozkaz zostanie wykorzystywany przy dostępie do pamięci decyduje wielkość „miejsc docelowego”, z tego względu nie wystarczy rzutować

wpisywanych danych na typ *uint8_t*. Trzeba zastosować wątpliwe wizualnie rzutowanie rejestru jak w linii 12 listingu.

Dawno asemblera nie było, prawda? To w ramach zabawy skompiluj sobie coś takiego jak na poniższym listingu (koniecznie bez optymalizacji), i popatrz w komplikacie jakie rozkazy zostały użyte przy dostępie do rejestru CRC_DR. Oczywiście zachęcam do przerwania czytania i zabawy samodzielnej ;)

Kod zabawowy:

```
1. static volatile uint8_t z8;
2. static volatile uint16_t z16;
3. static volatile uint32_t z32;
4.
5. CRC->DR = z32;
6. CRC->DR = (uint8_t)z32;
7.
8. CRC->DR = z16;
9. CRC->DR = z8;
10.
11. *(volatile uint8_t*)&CRC->DR = z8;
12. *(volatile uint8_t*)&CRC->DR = z16;
13. *(volatile uint8_t*)&CRC->DR = z32;
14.
15. // (uint8_t)CRC->DR = z8;
```

Adres rejestru CRC_DR to 0x4002 3000. Zmienne zostały rozmieszczone w pamięci następująco:

- *z32* - pod adresem 0x2000 0000
- *z16* - pod adresem 0x2000 0004
- *z8* - pod adresem 0x2000 0006

Kod zabawowy (kompilat):

```

1.      CRC->DR = z32;
2. 800025c:    4a36      ldr    r2, [pc, #216] ; (8000338 <main+0xe0>)
3. 800025e:    4b37      ldr    r3, [pc, #220] ; (800033c <main+0xe4>)
4. 8000260:    681b      ldr    r3, [r3, #0]
5. 8000262:    6013      str    r3, [r2, #0]
6.      CRC->DR = (uint8_t)z32;
7. 8000264:    4a34      ldr    r2, [pc, #208] ; (8000338 <main+0xe0>)
8. 8000266:    4b35      ldr    r3, [pc, #212] ; (800033c <main+0xe4>)
9. 8000268:    681b      ldr    r3, [r3, #0]
10. 800026a:   b2db      uxtb   r3, r3
11. 800026c:    6013      str    r3, [r2, #0]
12.      CRC->DR = z16;
13. 800026e:    4b32      ldr    r3, [pc, #200] ; (8000338 <main+0xe0>)
14. 8000270:    4a33      ldr    r2, [pc, #204] ; (8000340 <main+0xe8>)
15. 8000272:   8812      ldrh   r2, [r2, #0]
16. 8000274:   b292      uxth   r2, r2
17. 8000276:   601a      str    r2, [r3, #0]
18.      CRC->DR = z8;
19. 8000278:    4b2f      ldr    r3, [pc, #188] ; (8000338 <main+0xe0>)
20. 800027a:    4a32      ldr    r2, [pc, #200] ; (8000344 <main+0xec>)
21. 800027c:   7812      ldrb   r2, [r2, #0]
22. 800027e:   b2d2      uxtb   r2, r2
23. 8000280:   601a      str    r2, [r3, #0]
24.
25.      (*(volatile uint8_t*)&CRC->DR) = z8;
26. 8000282:   4b2d      ldr    r3, [pc, #180] ; (8000338 <main+0xe0>)
27. 8000284:   4a2f      ldr    r2, [pc, #188] ; (8000344 <main+0xec>)
28. 8000286:   7812      ldrb   r2, [r2, #0]
29. 8000288:   b2d2      uxtb   r2, r2
30. 800028a:   701a      strb   r2, [r3, #0]
31.      (*(volatile uint8_t*)&CRC->DR) = z16;
32. 800028c:   4b2a      ldr    r3, [pc, #168] ; (8000338 <main+0xe0>)
33. 800028e:   4a2c      ldr    r2, [pc, #176] ; (8000340 <main+0xe8>)
34. 8000290:   8812      ldrh   r2, [r2, #0]
35. 8000292:   b292      uxth   r2, r2
36. 8000294:   b2d2      uxtb   r2, r2
37. 8000296:   701a      strb   r2, [r3, #0]
38.      (*(volatile uint8_t*)&CRC->DR) = z32;
39. 8000298:   4b27      ldr    r3, [pc, #156] ; (8000338 <main+0xe0>)
40. 800029a:   4a28      ldr    r2, [pc, #160] ; (800033c <main+0xe4>)
41. 800029c:   6812      ldr    r2, [r2, #0]
42. 800029e:   b2d2      uxtb   r2, r2
43. 80002a0:   701a      strb   r2, [r3, #0]
44.
45. ...
46.
47. 8000338: 40023000 .word 0x40023000
48. 800033c: 20000000 .word 0x20000000
49. 8000340: 20000004 .word 0x20000004
50. 8000344: 20000006 .word 0x20000006
51. 8000348: 40021000 .word 0x40021000
52. 800034c: 001d0400 .word 0x001d0400
53. 8000350: 40022000 .word 0x40022000
54. 8000354: 00895440 .word 0x00895440
55. 8000358: e000e010 .word 0xe000e010

```

Po operacji z pierwszej linijki kompilatu nie spodziewamy się niczego specjalnego. Obie strony przypisania są 32-bitowe. Do *r2* ładowana jest wartość spod adresu 0x0800 0338, czyli 0x4002 3000 - adres rejestru CRC_DR. Jak nietrudno się domyśleć w *r3* ląduje adres zmiennej *z32*. Linijka 4 to odczytanie, do *r3*, zawartości pamięci spod adresu zapisanego w *r3*, to znaczy zawartości zmiennej *z32*. Na końcu ta wartość (z *r3*) jest zapisywana pod adresem przechowywanym w *r2*, czyli w CRC_DR. Wykorzystany jest do tego rozkaz *str*. Jeśli wczytasz się w opis tego rozkazu w *Programming Manualu* to zobaczysz, że samo *str* (bez sufiksu określającego wielkość danych) to rozkaz operujący na słowie. Czyli wszystko się zgadza.

Druga wersja (linijka 6 komplatu) zawiera rzutowanie wartości wpisywanej do CRC_DR na typ 8-mio bitowy. Zobaczmy co to zmieniło w wygenerowanym kodzie. Przypominam, że dążyliśmy do tego, aby zapis do CRC_DR był wykonany z użyciem rozkazu operującego danymi o rozmiarze 1B. Pierwsze trzy linijki są takie same jak poprzednio: w r2 ląduje adres CRC_DR, w r3 zawartość zmiennej z32. W ostatniej też nic się nie zmieniło, dalej wykorzystany jest zapis 32-bitowy. Doszedł za to tajemniczy rozkaz *uxtb*. Szybki skok do ściągawki (PM):

UXTB Zero extends an 8-bit value to a 32-bit value.

These instructions do the following:

1. Rotate the value from Rm right by 0, 8, 16 or 24 bits.

2. Extract bits from the resulting value:

UXTB extracts bits[7:0] and zero extends to 32 bits.

Czyli po polsku, ten rozkaz wyłuskuje 8 najmłodszych bitów z wartości i dopełnia je zerami do wartości 32 bitowej. Ma to sens. Rzutowaliśmy zmienną 32 bitową na 8 bitów - zmienna została więc obcięta i dopełniona zerami. No ale dalej zapis do CRC_DR jest 32 bitowy.

Zobaczmy wersje z linijk 12 i 18. Zmieniły się adresy zmiennych, no ale to oczywiste. Wartości zmiennych są ładowane za pomocą innych rozkazów, odpowiednio *ldrh* i *ldrb* dla zmiennej o rozmiarze pół słowa i jednego bajtu. Sufiksy oznaczają rozmiar ładowanych danych: *h* - *halfword*, *b* - *byte*. Nieźle, jest postęp, ale to zapis do CRC_DR miał być o rozmiarze bajt, a nie odczyt zmiennej :) Swoją drogą, uczciwie przyznaję, że nie wiem po co kompilator dołożył rozkazy *uxth* i *uxtb*. Po co dodatkowo skracać wartości? Nie wystarczyło, że zostały odczytane za pomocą *ldr* z sufiksem określającym rozmiar? Daleki jestem od zrzucania własnych błędów i braków w wiedzy na „błędy kompilatora”. Jednak tym razem nieśmiało świta mi myśl, że te rozkazy są jednak zbędne.

Jedziemy z operacją z linii 25. Początek jest identyczny jak w przypisaniu z linii 18. Ale na końcu mamy nasz upragniony święty gral! Zapis do CRC_DR (*r3*) jest wykonany z wykorzystaniem rozkazu operującego na danych 8 bitowych - *strb*. Ten przykład był łatwy, bo i zapisywana dana był 8 bitowa. Ale w przypisaniach z linijk 31 i 38 już tak prosto nie jest :]

Podsumowując - dopiero to paskudne rzutowanie rejestru spowodowało wygenerowanie pożądanego kodu, z 8-mio bitowym zapisem do CRC_DR. A! Żeby nie było wątpliwości. Na koniec odkomentuj sobie ostatnią wersję z „zabawowego kodu”. Rzutowanie może i optycznie ładniejsze, ale mało pozyteczne bo się nie skompiluje :)

Co warto zapamiętać z tego rozdziału:

- CRC w F334 jest o wiele bardziej elastyczne niż w F103 i F429
- kalkulator CRC można karmić danymi 1, 2 lub 4 bajtowymi (decyduje operacja zapisu do rejestru danych)
- o tym jaki rozkaz zostanie użyty do zapisu do pamięci, decyduje rozmiar celu

19. INTERFEJS SPI (“*POTIUS SERO QUAM NUMQUAM*”²⁹⁴)

19.1. Wstęp (F103 i F429)

Szeregowego interfejsu SPI (*Serial Peripheral Interface*) generalnie nie trzeba chyba nikomu przedstawiąć. To jeden z podstawowych środków komunikacji mikrokontrolera ze światem. Polska wikipedia²⁹⁵ twierdzi wręcz, że jest to „jeden z najczęściej używanych interfejsów”²⁹⁶ pomiędzy układami mikroprocesorowymi a peryferyjnymi. Szczególnie ukochany przez sympatyków AVRów, którzy często i gęsto wykorzystują SPI do ISP (*In System Programming*).

Mikrokontroler F103 ma oczywiście ISP, i to nie jedno. Szybki skok w bok do datasheetu - do tabelki w rozdziale *Device overview* - i już wiemy że F103 ma 3 układy SPI, a dwa z nich mogą pracować w trybie I²S. F429 nie pozostaje w tyle - sześć razy SPI. Powinno nam wystarczyć :) A! Nie pomył I²S z I²C! I²S to standard wykorzystywany do komunikacji z cyfrowymi układami audio („S” na końcu oznacza „sound”). Nim nie będziemy się zajmować, pozostaniemy przy zwykłym SPI.

SPI w STMach może pracować w kilku trybach:

- *full-duplex* - jednoczesna komunikacja dwukierunkowa z wykorzystaniem dwóch linii danych (MISO i MOSI) oraz linii zegarowej (SCK)
- *half-duplex* - komunikacja dwukierunkowa (ale nie jednocześnie) z wykorzystaniem jednej, dwukierunkowej linii danych oraz linii zegarowej
- *simplex* - komunikacja jednokierunkowa z wykorzystaniem jednej linii danych i linii zegarowej

W każdym z nich mikrokontroler może pełnić rolę układu nadzawanego (*master*) lub podrzędnego (*slave*). Ponadto możemy skonfigurować wedle potrzeby (z grubsza tak samo było w AVR):

- rozmiar ramki danych (8 lub 16 bitów)
- polaryzację i fazę sygnału zegarowego
- kolejność przesyłania bitów
- sposób „zarządzania” sygnałem wyboru układu podrzędnego SS (*Slave Select*)

Zatrzymajmy się na trochę przy sygnale wyboru układu (SS - *Slave Select*). Dla ujednolicenia dalszych rozważań przyjmijmy, że:

294 „Lepiej późno niż wcześnie.”

295 https://pl.wikipedia.org/wiki/Serial_Peripheral_Interface

296 zero konkretów, ale brzmi ładnie i marketingowo :)

- SS - oznacza sygnał wyboru układu, widziany wewnątrz bloku peryferyjnego SPI (niekoniecznie musi być powiązany z wyprowadzeniem mikrokontrolera)
- NSS - oznacza stan wyprowadzenia SPIx_NSS mikrokontrolera (to wyprowadzenie może być sprzętowo powiązane z sygnałem SS, ale nie musi)

Większość układów komunikujących się z wykorzystaniem magistrali SPI, korzysta z sygnału wyboru układu (*Slave Select*). Układ nadzędny (*master*) wymusza stan niski na linii wyboru układu podziemnego (*slave*), z którym chce aktualnie „rozmawiać”. Z kolei układ podziemny, wykrywając ten stan niski wie, że *master* właśnie z nich chce się komunikować. Tym sposobem jeden *master* może komunikować się z wieloma układami *slave*. Wystarczy aby każdy z nich miał osobną linię wyboru slave'a.

Za konfigurację sygnału SS i związanego z nim pinu SPIx_NSS odpowiadają trzy bity w rejestrach konfiguracyjnych bloku SPI: SPI_CR1_SSM, SPI_CR1_SSI, SPI_CR2_SSOE. Bit SSM pozwala wybrać sprzętowy lub programowy sposób generowania sygnału SS (przypominam - to jest wewnętrzny sygnał istniejący gdzieś w bloku SPI). Jeżeli bit SSM ma wartość zero, to sygnał SS jest sprzętowo powiązany z nóżką SPIx_NSS. W takiej konfiguracji (zalecane skupienie się):

- jeżeli mikrokontroler pracuje w roli *slave'a* to należy wymusić stan niski na nóżce NSS na czas transmisji danych
- jeżeli mikrokontroler pełni rolę *mastera* to zachowanie nóżki NSS zależy od bitu SSOE:
 - jeżeli SSOE ma wartość zero, to pin SPIx_NSS jest wejściem. Pojawienie się na tym wejściu stanu niskiego powoduje, że układ automatycznie przestaje być *masterem*. Takie rozwiązanie zabezpiecza przed kolizją, mogąą wystąpić jeśli na linii znajduje się kilka układów mogących pełnić funkcję układu nadziednego (praca w trybie *multi-master*).
 - jeżeli SSOE ma wartość jeden, to pin SPIx_NSS jest wyjściem. Jego stan zależy sprzętowo od stanu układu SPI. Na pinie wymuszany jest stan niski w momencie rozpoczęcia nadawania i utrzymuje się on aż do wyłączenia bloku SPI.

Programowe sterowanie sygnałem SS (SSM = 1) jest o wiele prostsze :) W tym przypadku stan sygnału SS zależy tylko i wyłącznie od bitu SSI. Natomiast wyprowadzenie SPIx_NSS mikrokontrolera, można wykorzystać do dowolnych celów jako GPIO - w żaden sposób nie wpływa ono na pracę bloku SPI. Jeżeli układ pracuje jako *master* to bit SSI należy ustawić (aby sygnał SS miał stan wysoki), zaś sygnały wyboru poszczególnych układów podziemnych należy generować

„ręcznie” z wykorzystaniem dowolnych GPIO. Jeżeli układ pracuje jako *slave* to bit SSI należy skasować na czas trwania transmisji (aby SS miał stan niski).

Uff. Wiem, że zakręcone. Ale to jest chyba najwyższy murek jaki musimy przeskoczyć w drodze do opanowania podstawowej obsługi SPI! Spróbujmy podsumować to w formie macierzowej:

Tabela 19.1. Konfiguracja sygnału SS

Sterowanie sygnałem SS	Stan sygnału SS zależy od	Tryb pracy układu	
		slave	master
programowe (SSM = 1)	bitu SSI	bit SSI musi być wyzerowany na czas transmisji	SSI = 1
sprzętowe (SSM = 0)	nóżki SPIx_NSS	nóżka SPIx_NSS musi mieć stan niski w czasie transmisji	SSOE = 0 stan niski na SPIx_NSS powoduje wyłączenie trybu master (<i>multi-master</i>) SSOE = 1 nóżka SPIx_NSS ma stan niski od chwili rozpoczęcia nadawania do wyłączenia SPI

Swoją drogą w ATmegach wyglądało to dosyć podobnie. Jeżeli AVR pracował jako *slave* to jakiś tam pin realizujący funkcję wyboru układu był wejściem i odpowiadał za sygnał *Slave Select*. To tak samo jak w STMie ze sprzętową obsługą NSS.

Z drugiej strony, jeśli Mega była *masterem* to można było wyłączyć sprzętową kontrolę SS (poprzez ustawienie pinu jako wyjście) lub używać pinu do kontroli kolizji przy pracy *multi-master*. Czyli znowu prawie tak samo jak w STM :) Starczy tej teorii.

Zadanie domowe 19.1: zacznijmy od czegoś prostego - zidentyfikujmy wyprowadzenia mikrokontrolerów F103 i F429 związane z układami SPI. Ćwiczeń nigdy za dużo! Do roboty!

Odpowiedź: według mnie, najszybsza metoda wyszukiwania wyprowadzeń w przypadku F103, to *CTRL+F* w datasheetcie i przeszukanie pod kątem hasła *SPI*. Tak czy siak wynik poszukiwań przedstawia się tak jak w tabeli 19.2. Koniecznie porównaj tabelkę z tym co sam ustaliłeś! Z F429 nie pomagam - jest prościej bo wystarczy przelecieć wzrokiem tabelę *STM32F427xx and STM32F429xx alternate function mapping*.

Tabela 19.2. Wyprowadzenia mikrokontrolera STM32F103 związane z układami SPI (w nawiasach podano wyprowadzenia po re-mapowaniu funkcji alternatywnych)

Interfejs	SPIx_NSS	SPIx_SCK	SPIx_MISO	SPIx莫斯I
SPI1	PA4 (PA15)	PA5 (PB3)	PA6 (PB4)	PA7 (PB5)
SPI2	PB12	PB13	PB14	PB15
SPI3	PA15	PB3	PB4	PB5

Szybki rzut oka na dokumentację zestawu HY-mini i już wiemy, że po odłączeniu wyświetlacza, piny związane z SPI1 i SPI2 będą wolne i gotowe do naszej zabawy :] Z SPI3 sprawa jest nieco bardziej skomplikowana, gdyż współdzieli on piny z JTAGiem. ST zresztą wyraźnie o tym ostrzega w RMie na początku rozdziału o SPI (taki wytluszczonej akapit z *warning* na początku).

We wstępie do Poradnika wspominałem, że warto wybierać płytka testowa z jak najmniejszą liczbą bajarów podłączonych na stałe do mikrokontrolera. W HY-mini można odłączyć cały moduł wyświetlacza. I chwała mu za to. W F429i-Disco wszystko jest podpięte na stałe i bez lutownicy nic nie zdziała. Smutny efekt jest taki, że z sześciu układów SPI jakie posiada F429, tylko jeden (!) ma wszystkie wyprowadzenia wolne... Tobie pozostawiam przyjemność sprawdzenia który :>

Co warto zapamiętać z tego rozdziału:

- SPI może pracować w trybach *full-duplex*, *half-duplex*, *simplex*
- peryferial oferuje „standardowe” możliwości konfiguracji (praktycznie takie same jak w AVR)
- tabela 19.1

19.2. Master ŚPI (F103 i F429)

Zróbmy coś ciekawszego!

Zadanie domowe 19.2: skonfigurujmy SPI do pracy jako master, full-duplex, 8b (to chyba najpopularniejsza konfiguracja). Niech program w regularnych odstępach czasu wysyła 4-ro bajtowe paczki jakichś danych. Do obsługi pojedynczego przesyłu napisz prostą funkcję, która wyśle to co dostała w argumencie i zwróci to co zostało odebrane z SPI²⁹⁷. Linia NSS ma być sterowana programowo: stan niski w czasie przesyłania paczki danych, potem na chwilę wysoki²⁹⁸ i zapetlij. Reszta konfiguracji (prędkość, ustawienia linii zegarowej, kolejność bitów) bez znaczenia.

297 SPI to ten śmiszny interfejs, w którym nadawanie i odbieranie przebiegają jednocześnie

298 nawet jeżeli do mikrokontrolera podłączony jest tylko jeden układ slave, to najczęściej nie można na stałe podpiąć jego linii SS do masy, gdyż większość układów wymaga „machania” linią SS w celu np. synchronizacji transmisji

My natomiast podepniemy się do SPI z analizatorem i będziemy podglądać cóż się tam dzieje ciekawego :) Czas start! Mała podpowiedź na koniec: zwróć szczególną uwagę na flagi - SPI_SR_TXE, SPI_SR_RXNE, SPI_SR_BSY - powinny się przydać :>

Przykładowe rozwiązanie (F103; GPIO - PA0, NSS - PA4, SCK - PA5, MISO - PA6, MOSI - PA7):

```

1. volatile uint32_t delayVar;
2.
3. void delay(uint32_t cnt){
4.     delayVar = cnt;
5.     while(delayVar);
6. }
7.
8. enum { deselect=0, select=1 };
9. void slaveSelect_ctrl(bool state){
10.    if(state){
11.        BB(GPIOA->ODR, PA0) = 0;
12.    }
13.    else {
14.        while(SPI1->SR & SPI_SR_BSY);
15.        BB(GPIOA->ODR, PA0) = 1;
16.    }
17. }
18.
19. uint16_t spi_rw(uint16_t data){
20.     while( !(SPI1->SR & SPI_SR_TXE) );
21.     SPI1->DR = data;
22.     while( !(SPI1->SR & SPI_SR_RXNE) );
23.     data = SPI1->DR;
24.     return data;
25. }
26.
27. int main(void) {
28.
29.     RCC->APB2ENR = RCC_APB2ENR_IOPAEN | RCC_APB2ENR_SPI1EN;
30.
31.     gpio_pin_cfg(GPIOA, PA0, gpio_mode_output_PP_10MHz); /* GPIO */
32.     gpio_pin_cfg(GPIOA, PA5, gpio_mode_alternate_PP_10MHz); /* SCK */
33.     gpio_pin_cfg(GPIOA, PA6, gpio_mode_input_pull); /* MISO */
34.     gpio_pin_cfg(GPIOA, PA7, gpio_mode_alternate_PP_10MHz); /* MOSI */
35.
36.     SysTick_Config(1000000/10);
37.
38.     SPI1->CR1 = SPI_CR1_SSM | SPI_CR1_SSI | SPI_CR1_SPE | SPI_CR1_MSTR;
39.
40.     static const uint16_t data[] = {0xa, 0xb, 0xc, 0x1234};
41.     uint32_t i = 0;
42.
43.     while(1){
44.         slaveSelect_ctrl(select);
45.         for(i=0;i<4;i++) spi_rw( data[i] );
46.         slaveSelect_ctrl(deselect);
47.         delay(3);
48.     }
49.
50. } /* main */
51.
52. _attribute_((interrupt)) void SysTick_Handler(void){
53.     if(delayVar) --delayVar;
54. }
```

Dla porządku zacznijmy od środka :)

27) na początku funkcji main nie ma nic nowego. Włączenie zegarów i konfiguracja portów. W przykładzie wykorzystuję 4 wyprowadzenia mikrokontrolera (PA0 jako zwykłe wyjście + 3 linie SPI - SCK, MISO, MOSI). Konfigurację wykorzystywanych linii SPI należy przeprowadzić

zgodnie z zaleceniami dokumentacji (jak zawsze). A dokładniej tabelki *SPI* w rozdziale *GPIO configurations for device peripherals* w RMie. Linia SPI1_NSS (PA4), czyli wyprowadzenie powiązane sprzętowo z funkcją wyboru układu podziemnego, nie jest wykorzystywane w programie z premedytacją. Szczegóły za chwilę.

36) konfiguracja SysTicka (przerwanie co 10ms), na SysTicku oparta jest wielce ambitna, pomocnicza, funkcja opóźniająca z linijk 3 - 6. Tym razem zapomniałem wrzucić do SysTicka migającą diodę :(Obiecuję poprawę.

38) konfiguracja SPI. Dwa ostatnie bity odpowiadają za włączenie interfejsu i tryb master. Po włączeniu interfejsu nie należy zmieniać niektórych z jego opcji konfiguracyjnych (patrz np. bit SPI_CR1_SPE). Warto więc, jeśli konfiguracja obejmuje kilka rejestrów, włączenie interfejsu zostawić sobie na sam koniec.

Dwa pierwsze bity (SSM, SSI) odpowiadają za sygnał SS. Nieśmiało przypuszczam, że niemało programistów straciło kupę czasu właśnie przez te bity. W programie oba są ustawiane (patrz tejbl 19.1). Czyli sygnał SS (wewnętrzny sygnał, który istnieje gdzieś w czeluściach bloku SPI) jest generowany programowo ($SSM = 1$) i jest zgodny z wartością bitu SSI (stan wysoki). Przy takiej konfiguracji interfejs SPI nie wykorzystuje pinu SPI_NSS! Można go dowolnie wykorzystać w programie, jak każdy inny nóżek. Sygnał wyboru układu podziemnego musimy sobie wygenerować na piechotę - wykorzystując dowolny GPIO. W programie wykorzystane jest PA0. Zapamiętaj ten akapit! Taka konfiguracja ($SSM = 1$, $SSI = 1$) jest praktycznie jedyną sensowną, jeśli STM ma być masterem SPI!

Możliwe jest jeszcze sprzętowe sterowanie pinem SPIx_NSS ($SSM = 0$, $SSOE = 1$) - zerknij na tabelkę 19.1. Tak jak już wcześniej wspominałem, pin SPIx_NSS przyjmie wtedy stan niski na początku komunikacji i będzie w nim trwał aż do wyłączenia SPI. Czyli jeśli potrzebujemy zmian na linii NSS (a większość układów podziemnych potrzebuje) to musielibyśmy co chwilą wyłączać SPI... bez sensu. Co więcej, jeśli podłączymy do mastera kilka układów slave, to będziemy potrzebowali wielu linii wyboru układu. Przy sprzętowej obsłudze mamy tylko jeden pin SPIx_NSS, więc... lipa. Podsumowując, prosty *rule of thumb* do zapamiętania:

Jeśli STM masteruje SPI, to ustaw bity SSM i SSI!

40) bufor z danymi do wysyłania. Edukacyjnie ma rozmiar 16b (a będziemy wysyłać dane 8b)

43 - 48) w pętli wysyłane są 4-ro bajtowe paczki danych i obsługiwana jest linia wyboru układu podziemnego. Pierwsza instrukcja z pętli odpowiada za ustawienie stanu niskiego na naszej linii wyboru układu. Funkcja *slaveSelect_ctrl* zostanie omówiona na końcu. Enum wprowadzony dla

zwiększenia czytelności. Drugi krok to mini-pętela, która cztery razy odpala funkcję *spi_rw()*. Odpowiada ona za wysyłane „bardzo ważnych danych” z naszego bufora. Trzeci krok to zdanie sygnału wyboru układu (znowu funkcja *slaveSelect_ctrl*) i na koniec krótkie opóźnienie. Przyjrzyjmy się funkcji *spi_rw()*.

19 - 25) funkcja przyjmuje (i zwraca) wartość 16 bitową. Dzięki temu jest bardziej uniwersalna. SPI może pracować z danymi 8 lub 16 bitowymi (patrz bit SPI_CR1_DFF). Jeżeli jest ustawiony format 8 bitowy, to w przypadku zapisu „większej” wartości (np. 16 lub 32 bity) do rejestru danych (SPI_DR), tylko najmniej znaczące 8 bitów jest brane pod uwagę. Czyli możemy bezkarnie wpisać wartość 16 bitową a SPI samo odrzuci górną połówkę. Tym sposobem mamy jedną funkcję, która może obsługiwać obie konfiguracje SPI (8 i 16 bitów). Miodzio.

Uwaga na przyszłość! Tym razem miodzio, ale nie zawsze! Czasem będzie „dziegio”. W niektórych mikrokontrolerach STM32 układ SPI działa nieco inaczej. Mianowicie wpisanie do rejestru danych wartości 16b, jeśli SPI działa w trybie 8b, nie spowoduje automatycznego obcięcia jednej połówki z wartości. Zamiast tego wykonane zostaną dwa osobne transfery - obie połówki wartości 16b zostaną wysłane jako dwie wartości 8b. Zjawisko to dotyczy mikrokontrolerów serii F0 i F3... i może jeszcze jakichś. Przypuszczam, że jest związane z tym, że posiadają one buforowanie rejestru danych SPI (kolejka FIFO). Tak czy siak, w ich przypadku, należy specjalnie zatroszczyć się o zapis do rejestru danych SPI. Przykładowo w konfiguracji 8b, nie wystarczy aby wpisywana wartość była typu ośmiobitowego. Konieczne jest słyśnięcie rzutowania rejestru danych (dotyczy zarówno odczytu jak i zapisu do DR):

```
*(volatile uint8_t*)&SPIx->DR
```

Mikrokontrolerów F103 i F429 „problem” nie dotyczy, ale już niedługo się nań nadziejmy w rozdziale 19.6. Koniec OT!

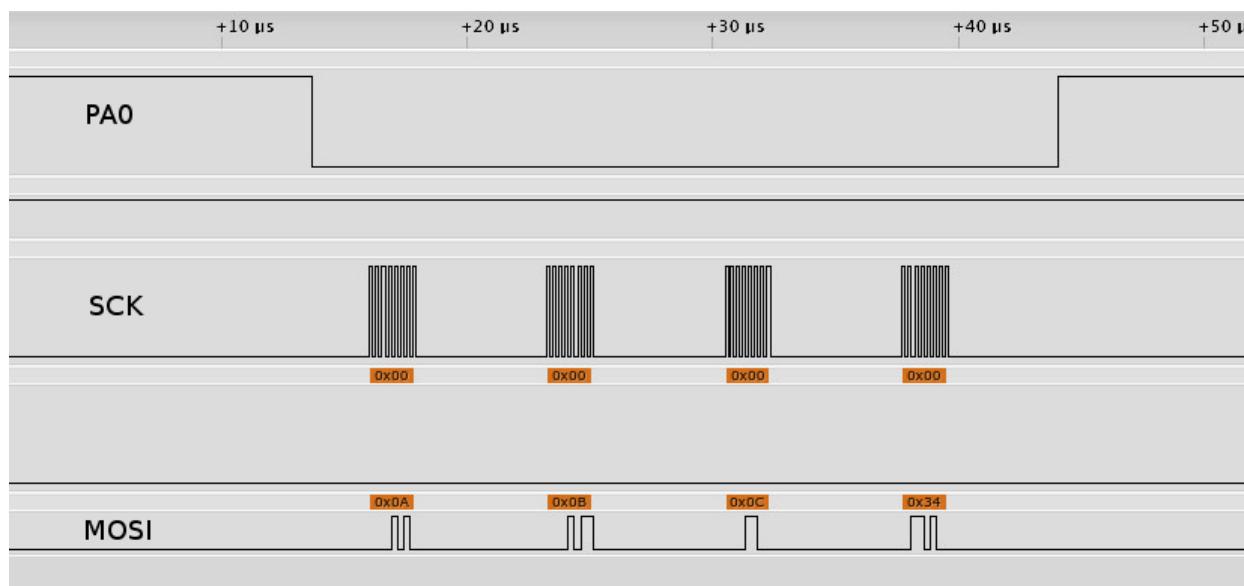
Przed wpisaniem wysyłanej wartości do rejestru danych (linijka 21) upewniamy się, że jest on pusty (flaga TXE). Flaga TXE oznacza, że rejestr/bufor nadawczy (SPI_DR) jest pusty i wpisując doń nową wartość, nie nadpiszemy poprzedniej. Oznacza to i tylko to! To, że flaga jest ustawiona nie znaczy, że SPI nic nie wysyła. Nowa wartość wpisana do bufora SPI_DR jest automatycznie przenoszona do jakiegoś wewnętrznego rejestru przesuwnego w układzie SPI, z którego kolejne bity tej wartości są „wysyłane” na linię MOSI. Flaga TXE jest ustawiana w

momencie zwolnienia bufora DR, czyli gdy zawartość zostaje przeniesiona do rejestru przesuwnego układu SPI. Fizyczne „wysyłanie danych w świat” dopiero się wtedy zaczyna. Flaga jest kasowana samoczynnie przy zapisie do rejestru/bufora DR.

SPI to ten śmieszny interfejs, w którym aby coś dostać - trzeba coś wysłać²⁹⁹. Odbieranie danych odbywa się równolegle z wysyłaniem. Bez względu na to czy układ podrzędny coś wysłał czy nie, master zawsze odbiera jakieś dane (np. same zera). Nasza funkcja `spi_rw()` ma możliwość zwracania odebranej wartości. W programie jej nie wykorzystujemy, ale przyda się w przyszłości. Przed odczytem odebranej wartości z rejestru danych DR, czekamy na flagę RXNE. Informuje ona o tym, że w rejestrze DR znajdują się jeszcze ciepłe „valid received data”. Flaga jest kasowana samoczynnie podczas odczytu rejestru DR.

9 - 17) wróćmy jeszcze na chwilę do funkcji `slaveSelect_ctrl`. Odpowiada ona za sterowanie pinem PA0, który generuje sygnał wyboru układu podzielnego. Stan niski oznacza „aktywację” układu, stan wysoki „de aktywację”. Pojawia się jeden haczyk. Stan niski na linii SS układu podzielnego musi utrzymywać się do zakończenia transmisji. Nie możemy „podnieść” linii SS dopóki trwa komunikacja po SPI, bo szlag ją trafi (komunikację). Musimy więc w jakiś sposób wykryć zakończenie transmisji. Właśnie po to jest flaga SPI_SR_BSY. Jeśli jest ona ustawiona to znaczy, że układ SPI jest zajęty komunikacją. Dopiero skasowanie flagi BSY (sprzętowe) wskazuje na zakończenie komunikacji - dopiero teraz możemy bezpiecznie zdjąć sygnał wyboru układu.

Trochę obrazków dla wzrokowców:

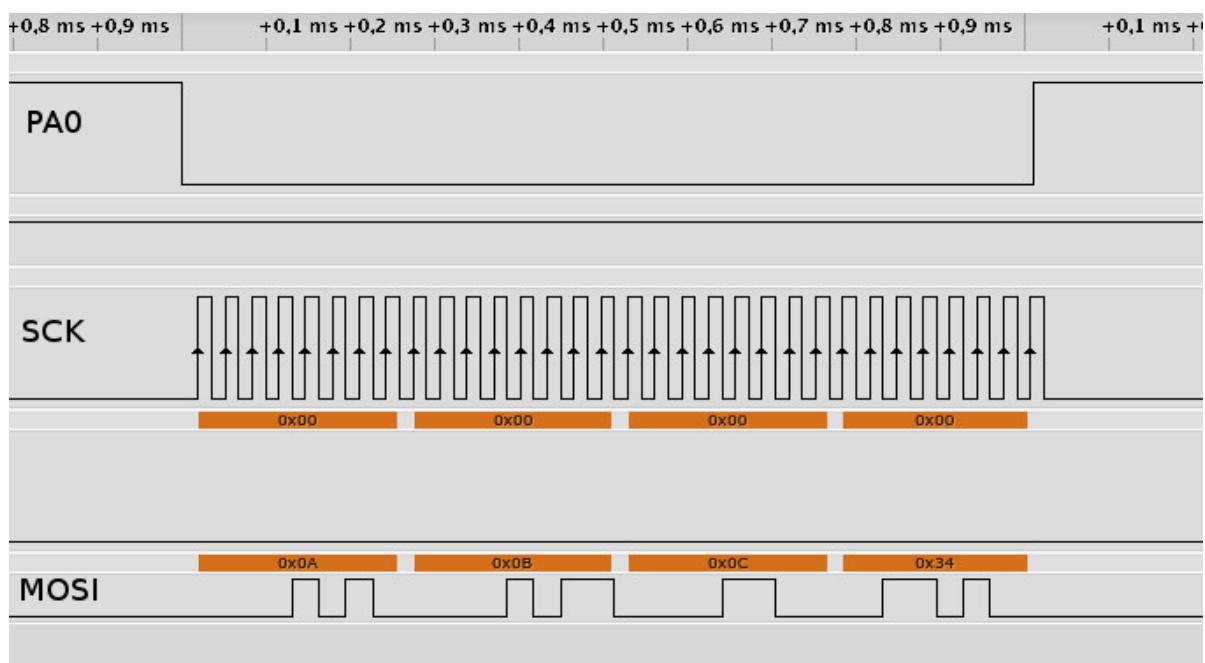


Rys. 19.1 Podglądarka SPI, wynik rozwiązania zadania 19.2

299 równość, sprawiedliwość i poprawność polityczna...

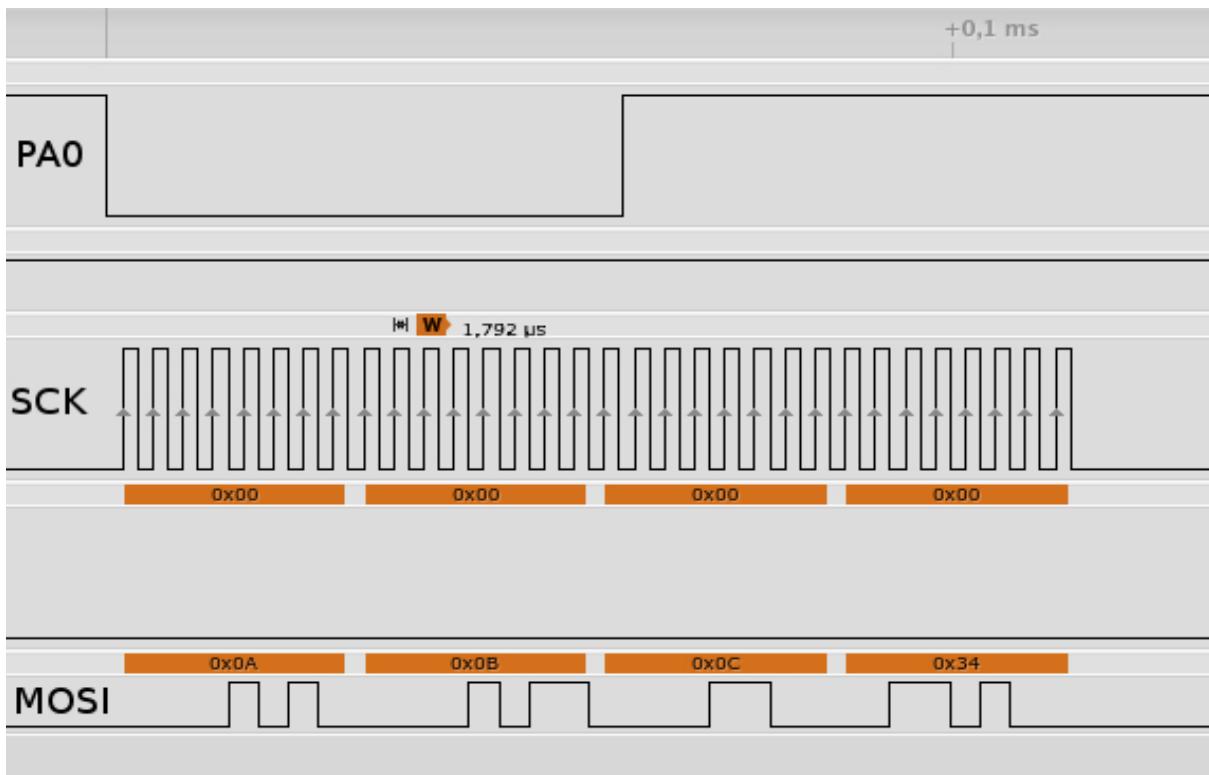
Na rysunku 19.1 pokazany jest wynik działania przykładowego rozwiązania zadania 19.2. Najpierw stan linii SS (PA0) zmienia się na niski, potem wysyłane są cztery bajty danych (0xA, 0xB, 0xC, 0x34). Na końcu, na linię SS powraca stan wysoki. Zwracam uwagę na wartość ostatniego wysłanego bajtu (0x34). Jest to „ucięta” dolna połówka ostatniej wartości z bufora z danymi do wysłania (0x1234, patrz linia 30 listingu). O „ucinaniu” danych przy konfiguracji 8 bitowej już pisałem, więc nie będę się powtarzał. Wszystko działa :)

Rysunek 19.2 to dokładnie to samo co wcześniej. Tylko tym razem (albo jak preferują niektórzy „tą razą” względnie „tom razom”³⁰⁰) program został skompilowany z włączoną optymalizacją. Kompilator spisał się bardzo ładnie. Program wykonuje się na tyle szybko, że SPI nie musi czekać na dane. Teraz to program czeka na SPI. Transmisja stała się ciągła - nie widać dziur w sygnale zegarowym.



Rys. 19.2 Podglądarkanie SPI, wynik rozwiązania zadania 19.2 (włączona optymalizacja)

300 przypuszcza, że w pewnych środowiskach obie formy są uznawane za równie poprawne :]



Rys. 19.3 Podglądarka SPI, wynik rozwiązań zadania 19.2 (bez sprawdzania flagi BSY i RXNE)

Rysunek 19.3 pokazuje co się dzieje, jeśli w programie zaniedba się sprawdzanie flag. Dla uzyskania „ładnego” przykładu zakomentowałem sprawdzanie flag RXNE i BSY. Zdecydowanie widać, że sygnał SS się pospieszył z powrotem do stanu wysokiego. Co się właściwie stało? Po kolei:

- nastąpił pierwszy zapis do bufora danych (SPI_DR), wpisana została wartość 0x0A
- zapis do DR skasował flagę TXE
- wartość z DR od razu została przeniesiona do rejestru przesuwnego układu SPI co zwolniło bufor, rozpoczęło się wysyłanie 0x0A
- zwolnienie bufora spowodowało ustawienie flagi TXE
- program wpisał drugą wartość (0x0B) do bufora danych SPI_DR, co skasowała flagę TXE
- program utknął w oczekiwaniu na TXE
- gdy pierwsza wartość (0x0A) została w całości wysłana, nowa wartość (0x0B) została przeniesiona do rejestru przesuwnego i rozpoczęło się jej wysyłanie
- przeniesienie wartości 0x0B zwolniło bufor danych (SPI_DR), flaga TXE się ustawiła
- program wpisał do rejestru SPI_DR trzecią wartość (0x0C) - flaga TXE uległa skasowaniu a program znowu utknął w oczekiwaniu na jej ustawienie

- zakończyło się wysyłanie wartości 0x0B, do rejestru przesuwnego została wpisana trzecia wartość (0x0C) - rozpoczęło się jej wysyłanie
- rejestr danych się zwolnił więc flaga TXE poszła w górę
- program wpisał do SPI_DR ostatnią wartość (0x34) i de aktywował linię SS
- układ SPI kontynuował wysyłanie wartości 0x0C i następnie 0x34

Ta dam! Wszystko się zgadza z rysunkiem 19.3. Na PA0 pojawił się stan wysoki praktycznie równocześnie z rozpoczęciem nadawania wartości 0x0C. SPI kontynuowało transmisję, ale podpięty do magistrali slave nie odebrałby dalszych danych z powodu zdjętego sygnału SS. Dlatego warto sprawdzać flagę BSY przed zmianą SS :)

Co warto zapamiętać z tego rozdziału:

- flagi TXE, RXNE, BSY
- „*Jeśli STM masteruje SPI, to ustaw bity SSM i SSI!*”
- rejestr danych (SPIx_DR) jest 16b, w konfiguracji 8b brana pod uwagę jest tylko dolna połowa wpisywanej do niego wartości

19.3. Master i Slave w jednym SP(al)I domu (F103 i F429)

Implementacja slave'a SPI jest dosyć niewdzięczna ze względu na specyfikę interfejsu (nadawanie przebiega równolegle z odbieraniem) i cechy STMowego SPI. Po włączeniu układu ustawiona jest flaga TXE (rejestr nadawczy SPI_DR jest pusty). Założymy, że wpiszemy do niego jakąś wartość X. Wartość X, niestety, nie zostanie od razu przeniesiona do rejestru przesuwnego układu (tak jak miało to miejsce w przypadku konfiguracji master). Zamiast tego będzie blokowała rejestr DR (flaga TXE pozostanie skasowana). Przeniesienie naszej wartości X z SPI_DR do wewnętrznego rejestru przesuwnego, nastąpi dopiero w momencie pojawienia się sygnału zegarowego na linii CLK interfejsu. Czyli gdy master zacznie nadawać. Spowoduje to ustawienie flagi TXE. Po odebraniu całej ramki danych, ustawiona zostanie ponadto flaga RXNE.

Najważniejsza rzecz, jaka wynika z tego w praktyce to to, że „dana” do wysłania (przez slave) musi być obecna w SPI_DR zanim rozpoczęcie się transmisja. Czyli zanim master zacznie nadawać - a na to slave nie ma wpływu. Staje się to szczególnie kłopotliwe jeśli odpowiedź układu slave zależy od poprzednio odebranych danych/komend. Układ musi zdążyć je przetworzyć i przygotować „odpowiedź” zanim master rozpoczęcie transmisję kolejnej ramki danych. Czyli w

najgorszym wypadku (przy ciągłej transmisji) ma na to jeden cykl zegara SPI. Jeżeli slave się nie wyrobi z wpisaniem nowej wartości do SPI_DR, to wysłana zostanie poprzednia wartość z tego rejestru. To może się masterowi nie spodobać.

Co do sygnału wyboru układu (SS), mamy dwie opcje konfiguracji: sterowanie programowe lub sprzętowe, patrz macierz 19.1. Sterowanie programowe jest proste - bit SPI_CR1_SSI ustala stan sygnału SS. Jeśli w implementowanym urządzeniu slave nie potrzebujemy tego sygnału to ustawiamy na stałe SSI = 0 (sygnał aktywny ma stan niski) i zapominamy o sprawie. Sterowanie sprzętowe też jest proste i działa zgodnie z oczekiwaniami (taka niespodzianka). Kiedy na nóżce SPIx_NSS jest stan niski to SPI pracujące w trybie slave „działa”, odbiera i wysyła dane po pojawienniu się sygnału zegarowego. Jeżeli natomiast na nóżce panuje stan wysoki to SPI ignoruje to co się dzieje na magistrali. I wszystko fajnie. Blok SPI nie dostarcza żadnych mechanizmów, które pozwalają programowi reagować na zmiany stanu sygnału SS. Do wykrywania zmian na linii wyboru układu należy wykorzystać np. przerwania zewnętrzne.

Nie ma co przedłużać :) Przystępujemy do działań operacyjnych!

Zadanie domowe 19.3: niech w F103 dwa SPI włączone będą, i kabelkami sprzętową się sprzeda. Jeden z nich niech masterem zostanie, drugi zaś niech odbiera co odeń dostanie. Dane niech master paczkami wypycha, po kilka bajtów w jedną upycha. Slave za to kiedy dane odbierze, niech je sumuje a wynik odeśle w następnym transferze.

SPI1 będzie masterem. Będzie wysyłał jakieś dane w paczkach po kilka bajtów, dokładnie tak samo jak w poprzednim przykładzie. Slave (SPI2) będzie te dane odbierał, na bieżąco je sumował i odsyłał wartość sumy masterowi. Master oczywiście ma weryfikować czy suma się zgadza. Suma ma być liczona tylko dla danych z danej paczki. Tzn. kiedy przesyłanie kilku bajtowej paczki się skończy, i master ustawi stan wysoki na linii wyboru układu podzielnego, to suma ma zostać wyzerowana. Tak aby przy kolejnej paczce była liczona od nowa. Powodzenia!

Przykładowe rozwiązań (F103; NSS: PA4 połączone z PB12, SCK: PA5 połączone z PB13, MISO: PA6 połączone z PB14, MOSI: PA7 połączone z PB15)

```
1. volatile uint32_t slaveSum;
2.
3. int main(void) {
4.
5.     RCC->APB2ENR = RCC_APB2ENR_IOPAEN | RCC_APB2ENR_IOPBEN | RCC_APB2ENR_SPI1EN |
6.             RCC_APB2ENR_AFIOEN;
7.     RCC->APB1ENR = RCC_APB1ENR_SPI2EN;
8.
9.     SysTick_Config(1000000/10);
10.    gpio_pin_cfg(GPIOB, PB0, gpio_mode_output_PP_2MHz);
11.
12.    /* Master SPI1 */
13.    gpio_pin_cfg(GPIOA, PA4, gpio_mode_output_PP_10MHz); /* NSS */
14.    gpio_pin_cfg(GPIOA, PA5, gpio_mode_alternate_PP_10MHz); /* SCK */
15.    gpio_pin_cfg(GPIOA, PA6, gpio_mode_input_pull); /* MISO */
16.    gpio_pin_cfg(GPIOA, PA7, gpio_mode_alternate_PP_10MHz); /* MOSI */
17.
18.    SPI1->CR1 = SPI_CR1_SSM | SPI_CR1_SSI | SPI_CR1_SPE | SPI_CR1_MSTR | (7<<3);
19.
20.    /* Slave SPI2 */
21.    gpio_pin_cfg(GPIOB, PB12, gpio_mode_input_floating); /* NSS */
22.    gpio_pin_cfg(GPIOB, PB13, gpio_mode_input_floating); /* SCK */
23.    gpio_pin_cfg(GPIOB, PB14, gpio_mode_alternate_PP_10MHz); /* MISO */
24.    gpio_pin_cfg(GPIOB, PB15, gpio_mode_input_floating); /* MOSI */
25.
26.    AFIO->EXTICR[3] = AFIO_EXTICR4 EXTI12_PB;
27.    EXTI->IMR = EXTI_IMR_MR12;
28.    EXTI->FTSR = EXTI_FTSR_TR12;
29.
30.    SPI2->DR = 0;
31.    SPI2->CR2 = SPI_CR2_RXNEIE;
32.    SPI2->CR1 = SPI_CR1_SPE;
33.
34.    NVIC_EnableIRQ(SPI2_IRQn);
35.    NVIC_EnableIRQ(EXTI15_10_IRQn);
36.
37.    static const uint8_t data[] = {10,3,12,67,54,23,29,43};
38.    uint32_t i, sum;
39.
40.    while(1){
41.        slaveSelect_ctrl(select);
42.        for(i=0, sum=0; i<8; i++){
43.            if( spi_rw(data[i]) != sum ) __BKPT();
44.            sum+= data[i];
45.        }
46.        slaveSelect_ctrl(deselect);
47.        delay(2);
48.        spi_rw(0x01);
49.        spi_rw(0x67);
50.        delay(2);
51.    }
52. } /* main */
53.
54. __attribute__((interrupt)) void SPI2_IRQHandler(void){
55.     if(SPI2->SR & SPI_SR_RXNE){
56.         slaveSum += SPI2->DR;
57.         SPI2->DR = slaveSum;
58.     }
59. }
60.
61. __attribute__((interrupt)) void EXTI15_10_IRQHandler(void){
62.     if(EXTI->PR & EXTI_PR_PR12){
63.         EXTI->PR = EXTI_PR_PR12;
64.         slaveSum = 0;
65.         SPI2->DR = 0;
66.     }
67. }
```

Trochę się rozrosło :) I fajnie. Na listingu nie uwzględniałem funkcji *spi_rw()*, *slaveSelect_ctrl()*, *delay()* i ISR SysTicka - są identyczne jak w poprzednim przykładzie. Z małym wyjątkiem. Ostatnio obiecałem dodać migającą diodę - dodałem ją do SysTicka :) Ale z migającą diodą, mam nadzieję, poradzisz sobie sam!

5 - 7) włączenie sygnałów zegarowych dla:

- portów A i B - linie SPI oraz migająca dioda
- dwóch układów SPI - tu ważna uwaga SPI1 jest podłączony do innej szyny niż SPI2 i SPI3, te szyny mają różne maksymalne częstotliwości sygnału zegarowego (APB1 - 36MHz, APB2 - 72MHz) może się więc okazać, że uzyskanie tych samych częstotliwości transmisji na SPI1 i SPI2/3 będzie wymagało konfiguracji interfejsów z różnymi preskalerami (patrz SPIx_CR1_BR)
- bloku AFIO - bo przerwania zewnętrzne

9) SysTick na 10ms - na SysTicku opiera się funkcja *delay()* i migająca dioda (nie pokazane na listingu bo nic nowego nie wnoszą)

10) konfiguracja pinu migającej ledy

13 - 16) wyprowadzenia związane z masterem (SPI1). Tym razem do generowania sygnału SS wykorzystałem wyprowadzenia SPI1_NSS (PA4). Przy czym tak jak w poprzednim przykładzie, sterowanie sygnałem następuje w pełni programowo. PA4 jest wykorzystywane jak każdy inny pin GPIO (pamiętasz *rule of thumb* dotyczącą konfiguracji SS w masterze?).

18) konfiguracja i włączenie SPI1, w porównaniu z poprzednim przykładem przybył tylko preskaler (ostatni „parametr”). Transmisję maksymalnie spowolniłem, żeby slave miał zapas czasu na przygotowanie danych do wysłania.

21 - 24) konfiguracja wyprowadzeń SPI2 (slave), ponownie posiliujemy się tabelką z rozdziału *GPIO configurations for device peripherals* (RM)

26 - 28) interfejs SPI (slave) nijak nie pomaga nam w zorientowaniu się, co się dzieje na linii wyboru układu. Z tego względu wspomagamy się przerwaniami zewnętrznyimi. Ustawiamy przerwanie aktywowane zboczem ↓ na PA4 (SPI2_NSS).

30) master może zacząć nadawać w każdej chwili i my (slave) musimy być na to gotowi (na razie zapomnijmy o tym, że oszukujemy system i nasz program będzie sam sobie slavem i masterem³⁰¹, wyobraźmy sobie że jesteśmy tylko slavem). W związku z tym jak najwcześniej podaję pierwszą

³⁰¹ „Sam sobie sterem, żeglarzem, okrętem”

wartość jaka zostanie odesłana do mastera, gdy ten rozpocznie komunikację. Slave ma sumować odebrane dane. Na razie nic nie odebrał, więc suma wynosi zero. Resetowa wartość rejestru SPI_DR wynosi zero, więc można by nic wpisywać - ale tak jest bardziej edukacyjnie!

31, 32) arcytrudna konfiguracja SPI slave (włączam przerwanie od odebrania danych i sam interfejs). Przy konfiguracji slave'a trzeba zwrócić uwagę na to aby ustawienia linii zegarowej (CPOL i CPHA) oraz kolejności bitów i wielkości ramki (LSBFIRST i DFF) były jednakowe po stronie mastera i slave'a. Ustawienie prędkości po stronie slave'a (SPI_CR1_BR) nie ma znaczenia, gdyż to master generuje sygnał zegarowy.

34, 35) włączenie przerwania zewnętrznego i przerwania od interfejsu SPI

37) losowe dane do wysyłania

38) zmienne pomocnicze

40 - 51) pętla główna. Pętla jest odpowiedzialna za działania mastera, w szczególności:

- sterowanie linią wyboru układu podległego (linie 41, 46)
- wysyłanie ośmiobajtowych paczek danych i sprawdzanie czy odebrana wartość jest równa sumie wysłanych danych (linijka 43)
- obliczanie sumy wysłanych danych (linijka 44)
- wysyłanie kilku bajtów danych gdy linia SS nie jest aktywna (linijki 48, 49) w celu sprawdzenia czy sprzętowe sterowanie slave'a sygnałem SS działa poprawnie (czy slave ignoruje dane gdy SS = 1). To jest np. symulacja sytuacji w której do jednego mastera podłączonych jest kilka slave'ów. Gdy master rozmawia z jednym z nich, pozostałe powinny ignorować ruch na magistrali.

54 - 59) przerwanie SPI2 jest wywoływanie po odebraniu danych (flaga RXNE), po sprawdzeniu flagi przerwania dodajemy odebraną wartość do zmiennej przechowującej sumę odebranych danych (*slaveSum*) i od razu ładujemy nową wartość sumy do rejestru nadawczego. Musimy zdążyć zanim master zacznie wysyłać kolejny bajt danych (bo równocześnie z wysyłaniem, rozpocznie się odbiór). Flagi przerwania nie kasujemy. Flaga RXNE kasuje się sama przy odczycie rejestru SPI_DR.

61 - 67) przerwanie zewnętrzne jest odpalane, gdy master „aktywuje” naszego slave'a. Po sprawdzeniu i skasowaniu flagi przerwania, zerujemy wartość sumy (zgodnie z założeniami) i zapisujemy nową wartość do rejestru nadawczego. Czemu? Dla uproszczenia założymy, że master wysyła trzy wartości: 1, 3, 5:

- pierwszy transfer: master wysyła „1” i równocześnie odbiera „0” (wartość wpisana przy konfiguracji SPI2); slave w przerwaniu oblicza sumę (1) i wpisuje ją do rejestru SPI2_DR
- drugi transfer: master wysyła „3” i równocześnie odbiera „1” (wartość wpisana do SPI2_DR w przerwaniu slave'a); slave w przerwaniu odbiera 3 i oblicza sumę ($1+3 = 4$) po czym wpisuje ją do SPI2_DR
- trzeci transfer: master wysyła 5 i równocześnie odbiera 4; slave w przerwaniu odbiera 5 i oblicza sumę ($4+5 = 9$) po czym wpisuje ją do SPI2_DR
- master wysłał już wszystko co miał więc zdejmuje sygnał SS
- po jakimś czasie rozpoczyna się kolejna transmisja - master aktywuje układ slave
- w układzie slave odpala się przerwanie zewnętrzne - jest w nim zerowana wartość zmiennej *slaveSum* (i założmy, że na tym przerwanie się kończy)
- master rozpoczyna przesył - coś tam sobie wysyła, natomiast w rejestrze nadawczym układu slave dalej siedzi wartość wpisana cztery kropki wyżej (9) a chcieliśmy aby suma liczona przez slave zerowała się po zmianie na linii SS. Dlatego właśnie w przerwaniu trzeba wpisać zero do SPI2_DR :) Nie ma co się przejmować tym, że rejestr DR nie jest pusty (wartość 9 nie została wysłana), po prostu nadpiszemy starą wartość i tyle

Co warto zapamiętać z tego rozdziału:

- implementacja slave'a SPI nie jest specjalnie sympatyczna
- slave musi mieć przygotowane dane zanim master rozpocznie transmisję
- stan linii wyboru układu należy badać np. wykorzystując przerwania zewnętrzne

19.4. Pół-puplex na dwa mikrokontrolery (F103 i F429)

Wszystkie przykłady z poprzednich rozdziałów zrealizowano w oparciu o F103 bez głębszych powodów. Układy SPI w F103 i F429 działają z grubsza tak samo. F429 ma dodatkowy bit (SPI_CR2_FRF) odpowiedzialny za obsługę formatu ramki TI. Wygooglanie różnic między „normalną” ramką (format Motorola) a formatem TI pozostawiam zainteresowanym. Ponadto przybyła flaga błędu ramki w formacie TI (SPI_SR_FRE) i związane z nią przerwanie.

Żeby F429 się nie obraził, przećwiczmy SPI jeszcze raz. W tym celu podrasujemy poprzedni przykład z sumą:

Zadanie domowe 19.4: niech mikrokontroler F429 (master) wysyła regularnie losowe dane w paczce o losowej długości z przedziału 100 - 1000B. Dane niech będą 16b. Układ podrzędny (F103) ma odbierać dane i na bieżąco odsyłać sumę odebranych danych. Podobnie jak poprzednio, suma ma być liczona dla każdej porcji danych od zera. No i teraz żeby było weselej: dwa warianty:

- wariant podstawowy - komunikacja w trybie full-duplex
- wariant rozszerzony - komunikacja w trybie half-duplex (jedna dwukierunkowa linia danych, linia SCK i ewentualnie linia wyboru układu podrzędnego)

Do roboty mości Czytelniku!

Przykładowe rozwiązanie (program dla slave'a - F103; wersja podstawowa - full-duplex):

```

1. #define __irq __attribute__((interrupt))
2. volatile uint32_t slaveSum;
3.
4. int main(void) {
5.
6.     RCC->APB2ENR = RCC_APB2ENR_IOPAEN | RCC_APB2ENR_IOPBEN | RCC_APB2ENR_SPI1EN |
7.             RCC_APB2ENR_AFIOEN;
8.
9.     SysTick_Config(1000000/10);
10.
11.    gpio_pin_cfg(GPIOA, PA4, gpio_mode_input_pull); /* NSS */
12.    gpio_pin_cfg(GPIOA, PA5, gpio_mode_input_pull); /* SCK */
13.    gpio_pin_cfg(GPIOA, PA6, gpio_mode_alternate_PP_10MHz); /* MISO */
14.    gpio_pin_cfg(GPIOA, PA7, gpio_mode_input_pull); /* MOSI */
15.
16.
17.    SPI1->CR2 = SPI_CR2_RXNEIE;
18.    SPI1->CR1 = SPI_CR1_DFF | SPI_CR1_SPE;
19.
20.    AFIO->EXTICR[1] = AFIO_EXTICR2 EXTI4_PA;
21.    EXTI->IMR = EXTI_IMR_MR4;
22.    EXTI->FTSR = EXTI_FTSR_TR4;
23.
24.    NVIC_EnableIRQ(SPI1_IRQn);
25.    NVIC_EnableIRQ(EXTI4_IRQn);
26.
27.    while(1);
28.
29. } /* main */
30.
31. __irq void SPI1_IRQHandler(void){
32.     if(SPI1->SR & SPI_SR_RXNE){
33.         slaveSum += SPI1->DR;
34.         SPI1->DR = slaveSum;
35.     }
36. }
37.
38. __irq void EXTI4_IRQHandler(void){
39.     if(EXTI->PR & EXTI_PR_PR4){
40.         EXTI->PR = EXTI_PR_PR4;
41.         slaveSum = 0;
42.         SPI1->DR = 0;
43.     }
44. }
```

Prawie nic nowego tu nie ma :) To jest praktycznie ten sam kod co w rozwiązaniu zadania 19.3, tylko wyrzuciłem wszystko co odpowiadało za realizację funkcji mastera. I zmieniłem konfigurację wyprowadzeń - wyłączyłem podciąganie, bo bez tego komunikacja nie była skora do współpracy.

Odpowiadające sobie sygnały slave'a (F103) i mastera (F429) połączone są przewodami (zgodnie z tabelką 19.3). Czas przejść do kodu mastera :)

Tabela 19.3. Wyprowadzenia układów master i slave (piny z każdej z kolumn należy połączyć)

układ	NSS	SCK	MISO	MOSI
slave (F103)	<i>PA4</i>	<i>PA5</i>	<i>PA6</i>	<i>PA7</i>
master (F429)	<i>PE4</i>	<i>PE2</i>	<i>PE5</i>	<i>PE6</i>

Przykładowe rozwiązanie (program dla mastera - F429; wersja podstawowa - full-duplex):

```
1. uint32_t rng(void){
2.     while ( !(RNG->SR & RNG_SR_DRDY) );
3.     if ( RNG->SR & RNG_SR_SEIS || RNG->SR & RNG_SR_CEIS ) __BKPT();
4.     return RNG->DR;
5. }
6.
7. enum { select = 0, deselect = 1 };
8. void slaveSelect_ctrl(bool state){
9.     while(SPI4->SR & SPI_SR_BSY);
10.    BB(GPIOE->ODR, PE4) = state;
11. }
12.
13. int main(void){
14.     RCC->CR |= RCC_CR_PLLON;
15.     while (!(RCC->CR & RCC_CR_PLLRDY));
16.
17.     RCC->AHB1ENR = RCC_AHB1ENR_CCMDATARAMEN | RCC_AHB1ENR_GPIOGEN | RCC_AHB1ENR_GPIOEEN;
18.     RCC->AHB2ENR = RCC_AHB2ENR_RNGEN;
19.     RCC->APB2ENR = RCC_APB2ENR_SPI4EN;
20.     __DSB();
21.
22.     RNG->CR = RNG_CR_RNGEN;
23.
24.     gpio_pin_cfg(GPIOE, PE2, gpio_mode_AF5_PP_MS); /* SCK */
25.     gpio_pin_cfg(GPIOE, PE4, gpio_mode_out_PP_MS); /* NSS-GPIO */
26.     gpio_pin_cfg(GPIOE, PE5, gpio_mode_AF5_PP_MS); /* MISO */
27.     gpio_pin_cfg(GPIOE, PE6, gpio_mode_AF5_PP_MS); /* MOSI */
28.
29.     SPI4->CR2 = SPI_CR2_RXNEIE;
30.     SPI4->CR1 = SPI_CR1_DFF | SPI_CR1_SPE | SPI_CR1_SSM | SPI_CR1_SSI | (7<<3) | SPI_CR1_MSTR;
31.     SPI4->DR = 0;
32.
33.     NVIC_EnableIRQ(SPI4_IRQn);
34.     while(1);
35.
36. } /* main */
37.
38. void SPI4_IRQHandler(void){
39.
40.     static uint32_t dataSum;
41.     static uint32_t dataCnt;
42.     static uint16_t rx, tx;
43.
44.     if(SPI4->SR & SPI_SR_RXNE){
45.
46.         rx = SPI4->DR;
47.         if ( rx != (uint16_t)dataSum ) __BKPT();
48.
49.         dataSum += tx;
50.
51.         if(dataCnt == 0){
52.             slaveSelect_ctrl(deselect);
53.             dataCnt = rng()%901 + 100;
54.             dataSum = 0;
55.             slaveSelect_ctrl(select);
56.         }
57.
58.         tx = rng() & 0xFFFF;
59.         SPI4->DR = tx;
60.         dataCnt--;
61.     }
62. }
```

1 - 5) tą funkcję to już znamy. Jest skopiowana z rozwiązania zadania 18.2. Rozrosła się jedynie o *breakpoint* w przypadku wykrycia błędu generatora liczb losowych. Nuda.

7 - 11) podobna funkcja do sterowania wyjściem wyboru układu podzadnego też już się przewijała. Znowu nuda.

24 - 27) konfiguracja wyprowadzeń. Przypominam, że konfiguracja funkcji alternatywnych w F429 wygląda zupełnie inaczej niż w F103! Numer funkcji alternatywnej odpowiadającej układowi SPI zidentyfikowany w oparciu o tabelę *STM32F427xx and STM32F429xx alternate function mapping* z datasheeta.

29) włączenie przerwań od odebrania nowej wartości przez SPI

30) konfiguracja SPI (16b, programowe sterowanie linią SS, maksymalna wartość preskalera sygnału zegarowego, master)

31) pierwsza wartość do wysłania (głównie po to, aby zainicjować pierwszy transfer)

40 - 64) generalnie cały program opiera się o przerwanie odbiorcze SPI. Odczytana wartość (slave odsyła sumę tego co dostanie) jest porównywana z sumą obliczoną w masterze (linijka 47). W razie wykrycia niezgodności program jest przerywany.

Zmienna *dataCnt* przechowuje ilość danych do wysłania (wielkość paczki danych). Jest dekrementowana przy wysyłaniu kolejnych losowych danych (linie 58 - 60). Gdy wartość *dataCnt* zjedzie do zera (czyli jest wysłana już cała paczka) to master (linijki 51 - 56):

- zdejmuje, na chwilę, sygnał SS z układu slave (czyli slave powinien wyzerować swoją sumę danych)
- losuje nową długość pakietu danych
- zeruje sumę wysłanych danych

Wysłanie nowej wartości (linijka 59) rozpoczyna kolejny transfer. Gdy nowa wartość zostanie odebrana, odpali się przerwanie. I tak w kółko. Generalnie jeśli chodzi o samo SPI, to nic odkrywczego tu nie ma :) Przejdźmy do drugiej wersji tego zadania.

No nie powiem... tym zadaniem „rozszerzonym” sam sobie zabiłem ćwieka. Oficjalnie uznaję, że SPI w *half-duplex* (jedna, dwukierunkowa linia danych) jest paskudne :)] Idea jest prościusia... w teorii. Aktualny kierunek transmisji jest ustalany przez bit SPI_CR1_BIDIOE. Układ „nadaje” gdy bit jest ustawiony lub odbiera, gdy bit jest skasowany. Zmianę kierunku musimy ogarnąć całkowicie programowo!

Jeżeli układ nadzędny ma nadawać (BIDIOE = 1) to transmisja rozpoczyna się w momencie wpisania nowej wartości do rejestru danych. Prawie „zwyczajnie”. Tyle tylko, że w konfiguracji *half-duplex* nadawaniu nie towarzyszy symultaniczne odbieranie! Jeżeli master chce coś odebrać to musi skasować swój bit BIDIOE. Wtedy pracuje jako odbiornik. Transmisja rozpoczyna się natychmiast po włączeniu interfejsu SPI. Kłopotliwe jest to, że po odebraniu ramki

danych, master będzie kontynuował „transmisje odbiorcze”, będzie odbierał kolejne ramki. Aby zatrzymać nadajnik, trzeba wyłączyć/przekonfigurować SPI.

Ze slave'em sprawa wygląda nieco prościej. Jeżeli układ ma coś odesłać to dane muszą być w rejestrze DR i musi być ustawiony bit BIDIOE zanim master rozpocznie nową transmisję. Jeżeli bit BIDIOE będzie skasowany to układ podrzędny będzie odbiornikiem. Proste? :> No to jedziemy!

Przykładowe rozwiązań (program dla slave'a - F103; wersja rozszerzona - half-duplex):

```
1. #define __irq __attribute__((interrupt))
2. volatile uint32_t slaveSum;
3.
4. int main(void) {
5.
6.     RCC->APB2ENR = RCC_APB2ENR_IOPAEN | RCC_APB2ENR_SPI1EN | RCC_APB2ENR_AFIOEN;
7.
8.     gpio_pin_cfg(GPIOA, PA4, gpio_mode_input_pull); /* NSS */
9.     gpio_pin_cfg(GPIOA, PA5, gpio_mode_input_pull); /* SCK */
10.    gpio_pin_cfg(GPIOA, PA6, gpio_mode_alternate_PP_10MHz); /* MISO */
11.
12.    SPI1->CR2 = SPI_CR2_RXNEIE;
13.    SPI1->CR1 = SPI_CR1_DFF | SPI_CR1_SPE | SPI_CR1_BIDIMODE;
14.
15.    AFIO->EXTICR[1] = AFIO_EXTICR2_EXTI4_PA;
16.    EXTI->IMR = EXTI_IMR_MR4;
17.    EXTI->FTSR = EXTI_FTSR_TR4;
18.
19.    NVIC_EnableIRQ(SPI1_IRQn);
20.    NVIC_EnableIRQ(EXTI4_IRQn);
21.
22.    while(1);
23.
24. } /* main */
25.
26. __irq void SPI1_IRQHandler(void){
27.     if(SPI1->SR & SPI_SR_RXNE){
28.         slaveSum += SPI1->DR;
29.
30.         SPI1->CR1 = SPI_CR1_DFF | SPI_CR1_SPE | SPI_CR1_BIDIMODE | SPI_CR1_BIDIOE;
31.         SPI1->DR = slaveSum;
32.
33.         while(~SPI1->SR & SPI_SR_TXE);
34.         while(SPI1->SR & SPI_SR_BSY);
35.
36.         SPI1->CR1 = SPI_CR1_DFF | SPI_CR1_SPE | SPI_CR1_BIDIMODE;
37.     }
38. }
39.
40. __irq void EXTI4_IRQHandler(void){
41.     if(EXTI->PR & EXTI_PR_PR4){
42.         EXTI->PR = EXTI_PR_PR4;
43.         slaveSum = 0;
44.         SPI1->DR = 0;
45.         SPI1->CR1 = SPI_CR1_DFF | SPI_CR1_SPE | SPI_CR1_BIDIMODE;
46.     }
47. }
```

1) mała definicje dla skrócenia późniejszego zapisu

8 - 10) konfiguracja wyprowadzeń, zwracam uwagę że wyleciało MOSI. SPI przy pracy z jedną, dwukierunkową linią danych wykorzystuje odpowiednio: MISO w układzie slave i MOSI w układzie master.

12, 13) włączenie przerwania odbiorczego SPI i konfiguracja interfejsu. Bit BIDIMODE odpowiada za konfigurację dwukierunkową (*bidirectional mode*). Nie ustawiam bitu BIDIOE, więc układ będzie odbiornikiem (będzie czekał na dane z mastera).

26) tu rozpoczyna się ISR od przerwania odbiorczego SPI. Pierwszy krok to oczywiście sprawdzenie flagi źródła przerwania. Flaga kasuje się sama przy odczycie danych z rejestru SPIx_DR.

28) zgodnie z założeniami zadania, slave ma sumować odebrane dane. No to sumujemy.

30) teraz robi się śmiesznie. Przyjąłem, że zmiana kierunku transmisji ma następować po każdej przesłanej wartości. Czyli master wysyła coś do slave'a, potem slave odpowiada masterowi i tak w kółko... Przed sekundą odebraliśmy nową wartość, czyli teraz to my (slave) będziemy wysyłać. Zgodnie z tym co pisałem przed chwilą, wymaga to prze-konfigurowania interfejsu SPI, należy ustawić bit BIDIOE. Zapisuję całą konfigurację rejestru SPI_CR1, żeby nie bawić się bez potrzeby w przypisanie z sumą bitową (`|=`). Zmieniamy konfigurację na „nadawczą” i w następnej linijce zapisujemy wartość do wysłania.

33, 34) ten kawałek nie jest zbyt finezyjny. Generalnie mamy teraz ustawiony tryb nadawczy i załadowaliśmy nową wartość do wysłania. Slave grzecznie czeka aż master rozpocznie odbieranie danych. Gdy zakończy się etap wysyłania, znowu powrócimy do roli odbiornika. I tu pojawia się mały problem - nie mam pomysłu jak „elegancko” rozwiązać do wracanie do roli odbiornika. W prezentowanym przykładzie czekam na flagi TXE i BSY (tak sugeruje dokumentacja). I to oczywiście działa, ale wolałbym jakieś rozwiązanie bez głupiego oczekiwania na flagę. Niestety flaga BSY nie może generować przerwania :(

No i w sumie tyleż nowości. Przełączmy się z powrotem na odbieranie i czekamy na wysłanie kolejnej wartości z mastera. No to teraz pora na kod mastera... siedzisz?

Przykładowe rozwiązań (program dla mastera - F429; wersja rozszerzona - half-duplex):

```
1. #define SPI_M_TX SPI_CR1_DFF | SPI_CR1_SPE | SPI_CR1_SSM | SPI_CR1_SSI | (7<<3) | SPI_CR1_MSTR
2. | SPI_CR1_BIDIMODE | SPI_CR1_BIDI0E
3. #define SPI_M_RX SPI_CR1_DFF | SPI_CR1_SPE | SPI_CR1_SSM | SPI_CR1_SSI | (7<<3) | SPI_CR1_MSTR
4. | SPI_CR1_BIDIMODE
5. #define SPI_M_RXN SPI_CR1_DFF | SPI_CR1_SSM | SPI_CR1_SSI | (7<<3) | SPI_CR1_MSTR |
6. SPI_CR1_BIDIMODE
7.
8. volatile uint32_t dataSum;
9.
10. uint32_t rng(void){
11.     while ( !(RNG->SR & RNG_SR_DRDY) );
12.     if ( RNG->SR & RNG_SR_SEIS || RNG->SR & RNG_SR_CEIS ) __BKPT();
13.     return RNG->DR;
14. }
15.
16. void delay(void){
17.     volatile uint32_t cnt = 50;
18.     while(cnt--) __DSB();
19. }
20.
21. enum { select = 0, deselect = 1 };
22. void slaveSelect_ctrl(bool state){
23.     BB(GPIOE->ODR, PE4) = state;
24. }
25.
26. void start_rx(void){
27.     SPI4->CR1 = SPI_M_RX;
28.     delay();
29.     SPI4->CR1 = SPI_M_RXN;
30. }
31.
32. int main(void){
33.     RCC->CR |= RCC_CR_PLLON;
34.     while ( !(RCC->CR & RCC_CR_PLLRDY));
35.
36.     RCC->AHB1ENR = RCC_AHB1ENR_CCMDATARAMEN | RCC_AHB1ENR_GPIOGEN | RCC_AHB1ENR_GPIOEEN;
37.     RCC->AHB2ENR = RCC_AHB2ENR_RNGEN;
38.     RCC->APB2ENR = RCC_APB2ENR_SPI4EN;
39.     __DSB();
40.
41.     RNG->CR = RNG_CR_RNGEN;
42.
43.     gpio_pin_cfg(GPIOE, PE2, gpio_mode_AF5_PP_FS); /* SCK */
44.     gpio_pin_cfg(GPIOE, PE4, gpio_mode_out_PP_MS); /* NSS-GPIO */
45.     gpio_pin_cfg(GPIOE, PE6, gpio_mode_AF5_PP_FS); /* MOSI */
46.
47.     SysTick_Config(16000000/500);
48.
49.     SPI4->CR2 = SPI_CR2_RXNEIE;
50.     NVIC_EnableIRQ(SPI4_IRQn);
51.
52.     while(1);
53.
54. } /* main */
55.
56. void SysTick_Handler(void){
57.
58.     static uint32_t dataCnt;
59.     static uint16_t tx;
60.
61.     enum { state_rx, state_tx };
62.     static uint32_t state = state_tx;
63.
64.     if(state == state_rx){
65.         start_rx();
66.         state = state_tx;
67.     } else if (state == state_tx){
68.
69.         if(dataCnt == 0){
70.             slaveSelect_ctrl(deselect);
71.             dataCnt = rng()%901 + 100;
72.             dataSum = 0;
73.             slaveSelect_ctrl(select);
74.         }
75.     }
76.
77. }
```

```

75.
76.        tx = rng() & 0xFFFF;
77.        SPI4->CR1 = SPI_M_TX;
78.        SPI4->DR = tx;
79.        dataSum += tx;
80.        dataCnt--;
81.        state = state_rx;
82.    }
83. }
84.
85. void SPI4_IRQHandler(void){
86.     if(SPI4->SR & SPI_SR_RXNE){
87.         uint16_t rx;
88.         rx = SPI4->DR;
89.         if ( rx != (uint16_t)dataSum ) __BKPT();
90.     }
91. }

```

1 - 6) wygląda strasznie, ale to tylko 3 definicje wartości konfiguracyjnych rejestru SPI_CR1 wprowadzone aby potem uprościć zapis:

- SPI_M_TX - ustawienia do nadawania (16 bit, SPI włączone, SS sterowane programowo, preskaler, master, tryb dwukierunkowy - BIDIMODE, nadawanie - BIDIOE)
- SPI_M_RX - ustawienia do odbierania (jw. tylko bez BIDIOE)
- SPI_M_RXN - ustawienia jak do odbierania, tylko bez bitu włączającego SPI (bitu SPE), zaraz się wyjaśni po co

10 - 24) funkcje: *delay()*, *rng()* oraz *slaveSelect_ctrl()* chyba nie wymagają specjalnego komentarza

26 - 30) tak jak pisałem, w trybie *half-duplex* master zaczyna odbierać, gdy tylko zostanie włączony interfejs SPI. I kontynuuje odbieranie do czasu wyłączenia peryferiala. Pojawia się więc pytanie: co jeśli chcemy odebrać tylko jedną „porcję” danych (8 lub 16bitów)? Musimy włączyć SPI w trybie odbiorczym (to rozpoczęcie komunikacji) i wyłączyć, zanim zacznie się druga transakcja. W RMie w rozdziale *Disabling the SPI* jest informacja, że jeśli chcemy przerwać odbieranie, to należy wyłączyć SPI jeden cykl zegara SPI po rozpoczęciu ostatniej transmisji. W praktyce trzeba zapewnić aby wyłączenie nastąpiło z opóźnieniem równym minimum jednemu okresowi zegara SPI. Oczywiście opóźnienie nie może być dowolnie długie. Jeżeli będzie dłuższe od okresu całej transmisji SPI, to rozpoczęcie się kolejny przesył.

Celem funkcji *start_rx()* jest inicjowanie transmisji odbiorczej. SPI jest konfigurowane w trybie odbiorczym, następnie wprowadzane jest krótkie opóźnienie i SPI jest wyłączone. Włączenie SPI powoduje że układ rozpoczyna transmisję odbiorczą. Opóźnienie dobrano na oko. Po wyłączeniu SPI układ kontynuuje rozpoczętą transmisję. Przy czym wyłączenie ma polegać tylko na skasowaniu bitu SPE. Pozostała konfiguracja powinna pozostać bez zmian.

32 - 54) w funkcji głównej nie ma nic ciekawego: włączenie zegarów, generatora liczb losowych, konfiguracja wyprowadzeń (przypominam, że w trybie dwukierunkowym master korzysta tylko z linii MOSI), włączenie SysTicka i przerwania odbiorczego od SPI. Koniec.

Działanie programu oparte jest o przerwanie licznika systemowego. Co 1ms, w przerwaniu SysTicka, program uruchamia na przemian transmisję odbiorczą i nadawczą. Zmienna *state* przechowuje informację o kierunku wykonywanej transmisji. Przy pierwszym uruchomieniu przerwania ma ona wartość *state_tx* oznaczający, że master będzie wysyłał dane. Spełniony jest wtedy warunek z 67 linii programu.

69 - 74) wyrażenie warunkowe odpowiedzialne za sterowanie linią wyboru układu podrzędnego i losowanie długości pakietu danych. Nic się nie zmieniło w porównaniu z przykładem w wersji „podstawowej”.

76) losowanie wartości do wysłania slave'owi

77) konfiguracja SPI w trybie nadawczym (w szczególności obejmuje ustawienie bitu BIDIOE)

78) wysłanie wylosowanej wartości

79) aktualizacja sumy wysłanych wartości

80) dekrementacja licznika wysłanych danych

81) zmiana wartości zmiennej *state*. Przy kolejnym wystąpieniu przerwania SysTicka zmienna będzie miała wartość *state_rx*. Prawdziwy będzie warunek z linii 64.

64 - 67) master rozpoczyna transmisję odbiorczą (funkcja *start_rx()*). Nie czekamy na jej zakończenie. Gdy transmisja się zakończy, wywołane zostanie przerwanie odbiorcze SPI, z linii 85. Wartość zmiennej *state* jest zmieniana, dzięki temu przy kolejnym przerwaniu SysTicka układ znowu będzie nadajnikiem.

85 - 91) przerwanie odbiorcze SPI. Odczytujemy odebraną wartość i porównujemy ją z sumą wysłanych danych. Jeżeli nie są równe to program jest przerywany.

No i tyle w temacie. Powyższy duet programików działał kilka minut na biurku i sumy po obu stronach się zgadzały. Z drugiej strony, szczerze powiedziawszy nie wiem na ile „elegancko” mi to wyszło. Pilnowanie zmian kierunku komunikacji w trybie *half-duplex* jest dosyć upierdliwe. Na szczęście rzadko pojawia się potrzeba implementacji egzotycznych trybów działania SPI :)

Co warto zapamiętać z tego rozdziału:

- implementacja trybu *half-duplex* jest niewdzięczna
- w trybie *half-duplex* trzeba programowo zadbać o zmiany kierunku transmisji
- w trybie *half-duplex* master działający jako odbiornik rozpoczyna transmisję (generuje sygnał zegarowy) natychmiast po włączeniu SPI

19.5. CRC w SPI (F103 i F429)

Moduł SPI posiada możliwość sprzętowej kontroli przesyłanych danych. Działa to tak, że podczas wysyłania, na bieżąco, liczona jest suma CRC (CRC8 lub CRC16 w zależności od wielkości ramki danych). Po wysłaniu ostatniego bajtu, należy ustawić bit SPI_CR1_CRCNEXT. Spowoduje to przesłanie obliczonej wartości CRC. I tak dalej... szalenie pasjonujący mechanizm. Doczytaj sobie sam. A! I zwróć uwagę na erratę, bo jest tam kilka wzmianek o CRC w SPI. Jak ktoś jest nawiedzony i mało mu jeszcze SPI to proponuję wzbogacić przykład *half-duplex* o kontrolę CRC :]

Mnie natomiast zainteresowało co innego. Czy nie da się czasem wykorzystać tego CRC z SPI do obliczania sumy jakichkolwiek danych, niekoniecznie przesyłanych przez SPI. A po co mi to? Sprzętowe CRC w mikrokontrolerach F103 i F429 (mówię o układzie peryferyjnym CRC) ma możliwość obliczania tylko wartości CRC32 z 4-ro bajtowych danych. Dodatkowo nie można zmienić wielomianu wykorzystywanego przy obliczaniu CRC (0x4c11db7). Blok CRC w SPI natomiast, potrafi obliczać CRC z danych 8 i 16 bitowych. Do tego można sobie ustawić dowolny wielomian CRC (patrz rejestr SPI_CRCPR). Jak zapewne się domyślasz, obecność tego rozdziału w Poradniku, świadczy o tym, że eksperymenty zakończyły się powodzeniem. Pacjent żyje i tym podobne.

SPI może obliczać CRC danych wysyłanych „na niby”. Jest tylko jeden warunek: wyprowadzenie SPI_SCK musi być skonfigurowane jako funkcja alternatywna. Tzn. ten pin musi sobie machać sygnałem zegarowym SPI. A czy cokolwiek jest do niego podłączone, to już inna inszość. Nie wiem czemu CRC nie chce działać bez funkcji alternatywnej na tym wyprowadzeniu. Pozostałe wyprowadzenia interfejsu SPI nie muszą być skonfigurowane. I dobrze.

Zadanie domowe 19.5: za pomocą funkcji CRC w SPI policzyć CRC8 (wielomian może być domyślny - 0x7) z jakiejś porcji danych. Porównać wynik z kalkulatorami online, np.:

- http://depa.usst.edu.cn/chenjq/www2/software/crc/CRC_Javascript/CRCCalculation.htm
(ustawienia: order - 8; polynom - 7; input - wedle potrzeb)
- <http://www.zorc.breitbandkatze.de/crc.html>
(ustawienia: order - 8; polynom - 7; initial value - 0; final xor value - 0; oba rewersy odptaszone)

Przykładowe rozwiązań (F103; uwaga na SPI_SCK - PA5, niech wisi w powietrzu)

```
1. uint16_t spi_rw(uint16_t byte){  
2.     while( !(SPI1->SR & SPI_SR_TXE) );  
3.     SPI1->DR = byte;  
4.     while( !(SPI1->SR & SPI_SR_RXNE) );  
5.     byte = SPI1->DR;  
6.     return byte;  
7. }  
8.  
9. int main(void) {  
10.    RCC->APB2ENR = RCC_APB2ENR_IOPAEN | RCC_APB2ENR_SPI1EN;  
11.    gpio_pin_cfg(GPIOA, PA5, gpio_mode_alternate_PP_10MHz); /* SCK */  
12.  
13.    /* Master SPI1 */  
14.    SPI1->CR1 = SPI_CR1_SSM | SPI_CR1_SSI | SPI_CR1_SPE | SPI_CR1_MSTR | SPI_CR1_CRCEN;  
15.  
16.    static const char* data = "W Paryzu najlepsze kasztany sa na placu Pigalle.";  
17.    uint32_t i=0;  
18.  
19.    for(i=0;i<strlen(data);i++) spi_rw(data[i]);  
20.    uint32_t crc = SPI1->TXCRCR;  
21.    __BKPT();  
22.  
23. } /* main */
```

Działanie programu powinno być dla Ciebie oczywiste. Funkcja *spi_rw()* nic się nie zmieniła, jest skopiowana z poprzednich przykładów. W programie włączany jest blok SPI i port A. Tak jak wspominałem, bez konfiguracji linii SCK obliczanie CRC nie działa, więc konfiguruję SCK w 12 linijce. Potem jest szybka konfiguracja SPI, jedyna nowość to bit włączający kalkulator CRC (SPI_CR1_CRCEN). Następnie w pętli wpisuję wszystkie dane, tak samo jakbym je wysyłał przez SPI. Na PA5 pojawia się przebieg zegarowy. Pozostałe wyprowadzenia związane z interfejsem nie są ustawione w konfiguracji funkcji alternatywnej, więc nic się z nimi nie dzieje. Na koniec odczytuje wartość CRC z rejestru SPI_TXCRCR. Wynik (0xBC) oczywiście jest zgodny z kalkulatorami :]

Co warto zapamiętać z tego rozdziału:

- funkcję CRC układu SPI można wykorzystać do obliczania sumy dowolnych danych
- aby działało zliczanie CRC konieczne jest skonfigurowanie wyprowadzenia sygnału zegarowego w trybie alternatywnym

19.6. Outsider (F334)

Jak już nieraz zauważyliśmy, w F334 wszystko musi działać nieco inaczej. SPI oczywiście też. Smutna wiadomość jest taka, że do dyspozycji mamy tylko jeden interfejs SPI. A teraz wesołe wiadomości:

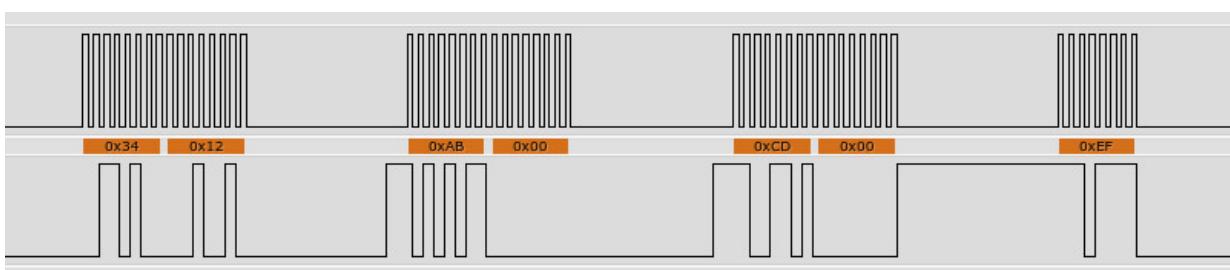
- rozmiar przesyłanej ramki danych może wynosić od 4 do 16 bitów, jest konfigurowany za pomocą SPIx_CR2_DS
- interfejs posiada dwie kolejki fifo (nadawczą i odbiorczą) o pojemności 2x32 bity każda:
 - RXNE jest zgłaszanego po odebraniu danych jeśli osiągnięto założony próg zapełnienia kolejki fifo
 - TXE jest zgłaszanego gdy stopień zapełnienia kolejki jest mniejszy bądź równy połowie jej pojemności
- 16-bitowy dostęp do rejestru danych, jeśli wybrano ramkę danych o rozmiarze do jednego bajta, powoduje uruchomienie mechanizmu pakowania danych:
 - zapisanie 2B wygeneruje dwa przesyły - najpierw młodszej części zapisanych danych, później starszej
 - odczytanie wartości 2B spowoduje odczytanie dwóch ramek danych
- przy sprzętowym sterowaniu (w trybie master) sygnałem wyboru układu podrzędnego doszła opcja umożliwiająca uzyskanie „pulsowania” linii NSS po skończonym transferze

SPI nie jest jakoś wyjątkowo ciekawe, więc tak na szybko spłodziłem przykładowy kod obrazujący działanie funkcji „pulsującego NSS” oraz pakowania danych.

Przykładowe rozwiązanie (F334):

```
1. void delay(void){
2.     volatile uint32_t i=5;
3.     while(i--);
4. }
5.
6. int main(void){
7.
8.     RCC->AHBENR = RCC_AHBENR_GPIOAEN;
9.     RCC->APB2ENR = RCC_APB2ENR_SPI1EN;
10.
11.    gpio_pin_cfg(GPIOA, PA4, gpio_mode_AF5_PP_LS);
12.    gpio_pin_cfg(GPIOA, PA5, gpio_mode_AF5_PP_LS);
13.    gpio_pin_cfg(GPIOA, PA7, gpio_mode_AF5_PP_LS);
14.    SPI1->CR2 = SPI_CR2_SSOE | 7<<8 | SPI_CR2_NSSP;
15.    SPI1->CR1 = SPI_CR1_SPE | SPI_CR1_MSTR;
16.
17.    while(1){
18.
19. #if (0)
20.     SPI1->DR = 0x1234;
21.     delay();
22.     SPI1->DR = 0xAB;
23.     delay();
24.     SPI1->DR = (uint8_t)0xCD;
25.     delay();
26.     *(volatile uint8_t *)&SPI1->DR = 0xEF;
27.     delay();
28.     delay();
29.     delay();
30. #endif
31.
32. #if (0)
33.     *(volatile uint8_t *)&SPI1->DR = 0x12;
34.     *(volatile uint8_t *)&SPI1->DR = 0x34;
35.     *(volatile uint8_t *)&SPI1->DR = 0x56;
36.     *(volatile uint8_t *)&SPI1->DR = 0x78;
37.     delay();
38.     SPI1->DR = 0xABCD;
39.     delay();
40. #endif
41.
42.    }
43. }
```

W konfiguracji nie ma nic nowego poza bitem odpowiedzialnym za włączenie pulsowania linii NSS. Ważne - rozmiar ramki danych zostawiłem domyślny - 8 bitów. Na początek sprawdźmy jak działa pakowanie danych. W tym celu „włącz” pierwszy kawałek kodu, objęty dyrektywami komplikacji warunkowej (19 - 30). Na rysunku 19.4 pokazany jest wynik jego działania (przebieg SCK i MOSI).



Rys. 19.4. SPI w F334, pakowanie danych (góra: SCK, dół: MOSI)

Program powstał na szybko i na kolanie, stąd zamiast sprawdzania flag bloku SPI dorzuciłem prymitywne opóźnienie zapobiegające nadpisaniu danych do wysłania. Tak czy siak, w pętli głównej są wysyłane po kolej, cztery wartości:

20) wysyłana jest wartość 16-bitowa³⁰² 0x1234. Czyli SPI ustawione na przesyły 8-bitowe, a my wpychamy wartość 16-bitową. Czego się spodziewamy? F103/F429 obcięłoby wpisaną wartość i wysłało tylko 8 bitów. F334 ma funkcję pakowania - najpierw powinno wysłać dolną połowę (0x34), potem górną (0x12). Paczamy na rysunek 19.4 celem potwierdzenia :)

22) teraz do rejestru danych wpisujemy wartość 0xAB. Czego oczekujemy na „wyjściu” tym razem? No to po kolej, dla sportu:

- rejestr SPI_DR jest zadeklarowany w nagłówku mikrokontrolera jako 32-bitowy
- tak jak pisałem w rozdziale 18.5 - o „szerokości” dostępu do pamięci, decyduje miejsce docelowe - mamy rejestr zadeklarowany jako 32 bitowy, więc i taki zapis zostanie użyty
- „fizycznie” SPI_DR jest 16-bitowym rejestrem, tzn. tylko 16 bitów ma jakieś znaczenie dla bloku SPI, reszta to dziura w pamięci
- kompilator przeprowadza domyślną promocję wartości (np. 0xAB) do typu int, który w naszym przypadku ma 4B

Tak więc spodziewamy się, że kompilator „obsłuży” zarówno wartość 0xAB jak i rejestr docelowy z wykorzystaniem rozkazów operujących danymi 32 bitowymi. Wartość 0xAB zostanie uzupełniona zerami do wymaganego rozmiaru. Blok SPI natomiast, uwzględni tylko 16 bitów z wpisanej wartości. Aktywuje się mechanizm pakowania danych - najpierw zostanie wysłana dolna połowa wartości (0xAB) a potem górną (0x00). Sprawdzamy?

Poniżej jest wygenerowany kod dla tego przypisania. Tak jak się spodziewaliśmy użyty został rozkaz zapisu, operujący na całym słowie (*str*).

Kompilat:

1.	SPI1->DR = 0xAB;
2.	8000238: 4b0a ldr r3, [pc, #40] ; (8000264 <main+0x84>)
3.	800023a: 22ab movs r2, #171 ; 0xab
4.	800023c: 60da str r2, [r3, #12]

Rysunek 19.4 potwierdza ponadto drugą część naszych wniosków. No i fajnie.

302 przypominam, że rejestr danych SPI jest 16-bitowy (patrz RM)

24) kolejne wersja przypisania, która niestety nie wnosi nic nowego i ciekawego. Przerabialiśmy to w rozdziale dotyczącym dostępu do rejestru danych bloku CRC (18.5). Efekt działania jest taki sam jak linii 22

26) wreszcie, za pomocą ślicznego rzutowania, udało się wysłać 1B przez SPI (patrz rysunek 19.4). Dla chętnych poniżej wycinek listingu. Pojawił się upragniony rozkaz **strb**.

Kompilat:

```
1. 800024c: *(volatile uint8_t *)&SPI1->DR = 0xEF;
2. 800024e:    ldr    r3, [pc, #24] ; (8000268 <main+0x88>)
3. 800024e:    22ef    movs   r2, #239   ; 0xef
4. 8000250:    701a    strb   r2, [r3, #0]
```

No to teraz zobaczymy jak działa druga z nowych funkcji bloku SPI. Chodzi o „pulsowanie” linii NSS przy sterowaniu sprzętowym przez mastera. Przekompilujemy kod, tym razem z „włączoną” drugą częścią (linie 32 - 40). Nie wdając się w szczegóły, efekt pokazano na rysunku 19.5.



Rys. 19.5. SPI w F334, pulsowanie NSS (góra: NSS, środek: SCK, dół: MOSI)

W programie cztery razy wysyłany jest pojedynczy bajt danych, potem wysyłane są dwa bajty (z wykorzystaniem pakowania danych). Jak widać na rysunku, linia wyboru układu podzielnego dezaktywuje się na chwilę po każdym wysłanym bajcie. Fajnie, prawda? :)

Co warto zapamiętać z tego rozdziału:

- SPI w F334 ma kolejki fifo
- przy wysyłaniu/odbieraniu danych 1B, trzeba rzutować rejestr danych SPI - inaczej uruchomi się funkcja pakowania danych i wysłane zostaną 2B

20. KOMPARATOR I WZMACNIACZ („*NEC TEMERE, NEC TIMIDE*”³⁰³)

20.1. Komparator analogowy (F334)

Mikrokontroler F334 posiada trzy wbudowane komparatory analogowe: COMP2, COMP4, COMP6³⁰⁴. Każdy komparator posiada wyjście cyfrowe, którego stan zależy od wyniku porównania dwóch napięć na wejściach analogowych. O polaryzacji sygnału wyjściowego decyduje bit COMPx_CSR_COMPxPOL. Do wejść analogowych mogą zostać doprowadzone napięcia z przedziału od 0 do V_{DDA}.

Każdy komparator może pracować *samodzielnie (standalone)* lub współpracować z innymi periferiami mikrokontrolera. Praca samodzielna polega na tym, że wszystkie „sygnałowe” wyprowadzenia komparatora są dostępne poprzez piny mikrokontrolera. To tak, jakby komparator nie miał nic wspólnego z resztą mikrokontrolera poza zasilaniem i obudową. W poniżej tablicy znajduje się podsumowanie możliwych konfiguracji wejść i wyjść komparatorów.

Tabela 20.1 Wejścia i wyjścia komparatorów analogowych (na podstawie tabeli *STM32F334xx comparator input/outputs summary* z RMa oraz datasheetu)

Komparator	COMP2	COMP4	COMP6
wejście odwracające/ujemne	PA2, PA4 ³⁰⁵ DAC1_CH1, DAC1_CH2, DAC2_CH1, Vrefint, ¼ Vrefint, ½ Vrefint, ¾ Vrefint	PB2, PA4 ³⁰⁵	PB15, PA4 ³⁰⁵
wejście nieodwracające/dodatnie	PA7	PB0	PB11
wyjście	PA2 (AF8), PA12 (AF8), PB9 ³⁰⁶ (AF8) TIM1_OCREF_CLR, TIM1_IC1, TIM2_IC4, TIM2_OCREF_CLR, TIM3_IC1, TIM3_OCrefClear, HRTIM_EEV1, HRTIM_EEV6	PB1 (AF8) TIM3_IC3 TIM3_OCrefClear TIM15_OCREF_CLR TIM15_IC2 HRTIM_EEV2 HRTIM_EEV7	PA10 (AF8), PC6 (AF7) TIM2_IC2 TIM2_OCREF_CLR TIM16_OCREF_CLR TIM16_IC1 HRTIM_EEV3 HRTIM_EEV8
		T1BKIN, T1BKIN2	

303 „Bez zuchwałości, ale i bez lęku.”

304 też uważam, że nazwanie ich COMP1, COMP2 i COMP3 byłoby zbyt proste i mało profesjonalne

305 wejście odwracające PA4 jest „wątpliwe”: jest uwzględnione w schemacie blokowym komparatora w RMie oraz w tabeli *STM32F334x4/6/8 pin definitions* w datasheetcie; natomiast nie jest wymienione w tabeli *Alternate functions* w datasheetcie oraz w tabeli *STM32F334xx comparator input/outputs summary* w RMie - trzeba to sprawdzić :)

[sprawdziłem - działa! - oczywiście tylko, gdy DAC nie jest włączony, bo na tym samym pinie jest jego wyjście]

306 wyjście COMP2 na PB9 jest wymienione w tabelach *STM32F334x4/6/8 pin definitions* oraz *Alternate functions* w datasheetcie; nie pojawia się natomiast w tabeli w RMie... i weź bądź tu mądry (sprawdziłem - działa!)

Pomijając pracę *samodzielna*, komparatory mogą generalnie współpracować z licznikami. Dodatkowo do wejścia odwracającego można podać sygnał pochodzący z wewnętrznego źródła napięcia odniesienia (Vrefint) lub wygenerowany przez przetwornik DAC.

Do czego można wykorzystać komparator? Taki książkowy (RMowy) przykład to sterowanie np. silnikiem i kontrola prądu. Timer generuje sygnał PWM sterujący jakimś układem wykonawczym. Do komparatora doprowadzony jest sygnał napięciowy proporcjonalny do prądu silnika. Jeżeli natężenie przekroczy zadany próg, to komparator wyłącza generowanie PWM (np. poprzez funkcję *break*). Połączenie wyjścia komparatora z sygnałami IC (*Input Capture*) licznika, można wykorzystać do jakichś pomiarów czasowych - np. pomiaru czasu, w którym sygnał miał wartość niższą niż zadany próg. Przykładowo można w ten sposób zrealizować miernik pojemności (poprzez pomiar czasu ładowania kondensatora).

Co jeszcze ciekawego potrafią komparatorze? Oczywiście potrafią generować przerwania. Wyjścia komparatorów podłączone są do kontrolera EXTI. Przerwania mogą być generowane przy obu zboczach sygnału wyjściowego. Wykorzystanie kontrolera EXTI powoduje, że przerwania są w stanie wybudzać mikrokontroler z trybów uśpienia *Sleep* (nic specjalnego) oraz *Stop* (to już ciekawiej). Ponadto tytuły układy posiadają jeszcze dwa fizyczne:

- funkcję blokowania konfiguracji (*lock*)
- funkcję „wygaszania” wyjścia (*blinking*)

Funkcja blokowania chyba nie budzi wątpliwości. Komparatory mogą być wykorzystane do krytycznych zadań, np. zabezpieczeń nadprądowych, termicznych, itp. Z tego względu, aby zwiększyć niezawodność układu, zaimplementowano możliwość blokowania ich konfiguracji. Służy do tego bit COMPx_CSR_COMPxLOCK. Ustawienie tego bitu powoduje zablokowanie całego rejestru konfiguracyjnego COMPx_CSR (staje się tylko do odczytu), aż do resetu systemowego mikrokontrolera. Dzięki temu błędnie działający program³⁰⁷ nie będzie mógł zmienić konfiguracji zabezpieczonych komparatorów.

Funkcja *blinking* jest wykorzystywana przy sterowaniu (przez PWM) jakimś obwodem energoelektronicznym. Chodzi o to, aby układ kontroli prądu (oparty o komparator) nie reagował na chwilowe szpilki występujące tuż po komutacji. Idea jest taka, aby sygnał z komparatora był ignorowany tuż po zboczu sygnału PWM. W tym celu wyjście komparatora, przed dojściem do licznika (np. do wejścia *break*) przechodzi przez bramkę *and*. Do drugiego wejścia bramki, doprowadzony jest sygnał z wybranego bloku porównawczego licznika (patrz

³⁰⁷ program zawsze może pójść w malinowy chruśniak!

`COMPx_CSR_COMPx_BLANKING`). Manipulując konfiguracją tego bloku porównawczego, można „wygaszać” sygnał komparatora w określonych momentach synchronicznie z generowanym przebiegiem PWM. Ładnie widać to na rysunku *Comparator output blanking* w rozdziale *Comparator output blanking function*.

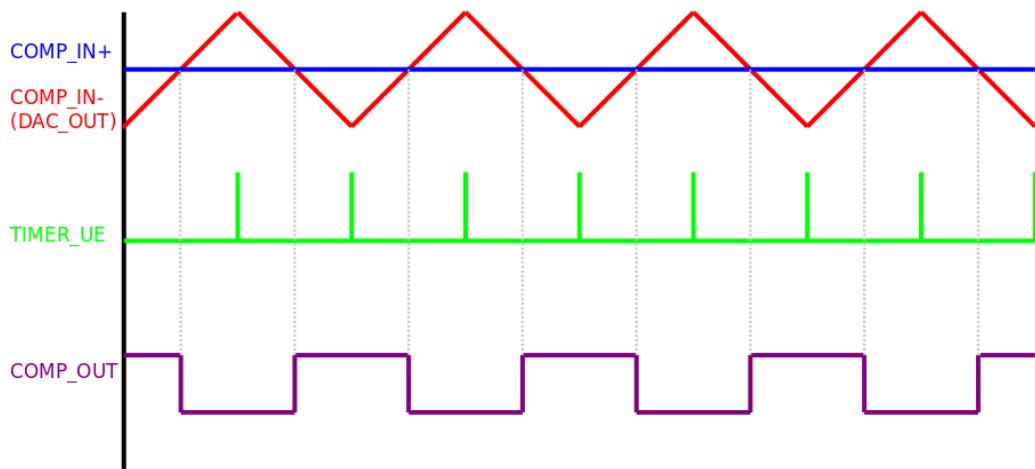
Zadanie domowe 20.1: uruchomić komparator analogowy. Do jednego wejścia doprowadzić stałe napięcie $\sim \frac{1}{2} V_{DDA}$. Do drugiego wejścia doprowadzić przebieg trójkątny o amplitudzie $\sim V_{DDA}$. Włączyć przerwanie komparatora. Niech w ISR będzie machanie dwoma wybranymi wyprowadzeniami mikrokontrolera (po jednym na każde zbocze sygnału wyjściowego). Z pomocą oscyloskopu/analizatora sprawdzić, podejrzeć efekty działania układu :)

Przykładowe rozwiązanie (gadanie na początek): zaczniemy od planu operacyjnego. Łatwiej będzie zrozumieć kod, gdy będzie wiadomo co i po co się w nim dzieje :) Sam komparator jest raczej nudny i prosty. Ale przykład jest nieco ciekawszy! Już w niejednym przykładzie stosowałem „sztuczki”, żeby tylko nie musieć organizować dodatkowego *hardware'u*³⁰⁸. Tym razem jest podobnie :] Skąd mi się urodził ten „przebieg trójkątny”? A no stąd: rysunek 14.2. DAC potrafi generować trójkąt. Dodatkowo wyjście DACa jest połączone z wejściem komparatora. Cóż za wspaniały zbieg okoliczności, nie uważaś? Nic tylko to wykorzystać. W programie będziemy mieli zatem komparator i DAC. I...? I co jeszcze $>?$ Nic nie sugeruję, ale powinieneś to już wiedzieć... Nie! Nie DMA... nie tym razem. No! Oczywiście, że timer. W końcu, jak doskonale pamiętamy, DAC może generować przebieg trójkątny (tudzież szum), tylko gdy jest okresowo wyzwalany z zewnątrz - np. timerem.

Dorzucimy sobie jeszcze jeden myk. DAC skonfigurujemy tak, aby na jedno zbocze sygnału trójkątnego przypadało 4095 wyzwoleń przetwornika. Inaczej mówiąc na początku, na wyjściu DACa, będzie w przybliżeniu 0. Każde kolejne wyzwolenie przetwornika, będzie powodowało wzrost napięcia. Po 4095 wyzwoleniach napięcie osiągnie wartość maksymalną (około V_{DDA}). Kolejne 4095 wyzwoleń to będzie „opadanie” napięcia do zera. I zapętlaj. W programie dorzucimy przerwanie licznika, które co 4095 wyzwoleń DACa, będzie machało jakimś pinem mikrokontrolera. Dzięki temu będziemy mieli punkt odniesienia na analizatorogramie.

Podsumowując spodziewamy się czegoś takiego jak na rysunku 20.1.

308 „Zdrowy organizm bronie się przed pracą.”



Rys. 20.1. Przykład z komparatorem (czerwony - przebieg trójkątny z DAC podany na wejście ujemne komparatora; niebieski - stałe napięcie podane na wejście dodatnie; zielony - zmiana kierunku zmian przebiegu trójkątnego; chyba fioletowy - wyjście komparatora)

No to jedziemy z tym kodem:

Przykładowe rozwiązanie (F334):

```
1. static inline void wave_pin(GPIO_TypeDef * const port, GpioPin_t pin){
2.     port->BSRR = pin;
3.     __DSB(); __DSB(); __DSB(); __DSB(); __DSB();
4.     port->BRR = pin;
5. }
6.
7. int main(){
8.
9.     RCC->AHBENR |= RCC_AHBENR_GPIOCEN | RCC_AHBENR_GPIOAEN | RCC_AHBENR_GPIOBEN;
10.    RCC->APB1ENR = RCC_APB1ENR_DAC1EN | RCC_APB1ENR_TIM6EN;
11.    RCC->APB2ENR = RCC_APB2ENR_SYSCFGEN;
12.
13.    gpio_pin_cfg(GPIOC, PC10, gpio_mode_out_PP_LS); //rising edge
14.    gpio_pin_cfg(GPIOC, PC12, gpio_mode_out_PP_LS); //falling edge
15.    gpio_pin_cfg(GPIOC, PC11, gpio_mode_out_PP_LS); //dir change
16.    gpio_pin_cfg(GPIOA, PA4, gpio_mode_analog); //dac1_ch1 / comp2_in-
17.    gpio_pin_cfg(GPIOA, PA7, gpio_mode_analog); //comp2_in+
18.    gpio_pin_cfg(GPIOA, PA12, gpio_mode_AF8_PP_LS); //comp2_out
19.    gpio_pin_cfg(GPIOB, PB9, gpio_mode_AF8_PP_LS); //comp2_out
20.
21.    DAC1->CR = DAC_CR_EN1 | DAC_CR_MAMP1 | DAC_CR_WAVE1_1 | DAC_CR_TEN1;
22.
23.    COMP2->CSR = COMP2_CSR_COMP2LOCK | COMP2_CSR_COMP2INSEL_2 | COMP2_CSR_COMP2EN;
24.    COMP2->CSR = 0;
25.
26.    TIM6->ARR = 100;
27.    TIM6->DIER = TIM_DIER_UIE;
28.    TIM6->CR2 = TIM_CR2_MMS_1;
29.    TIM6->CR1 = TIM_CR1_CEN;
30.    NVIC_EnableIRQ(TIM6_DAC1 IRQn);
31.
32.    EXTI->RTSR = EXTI_RTSR_TR22;
33.    EXTI->FTSR = EXTI_FTSR_TR22;
34.    EXTI->IMR = EXTI_IMR_MR22;
35.    NVIC_EnableIRQ(COMP2 IRQn);
36.
37.    while(true);
38. }
39.
40. void COMP2_IRQHandler(void){
41.     if(BB(EXTI->PR, EXTI_PR_PR22)){
42.         EXTI->PR = EXTI_PR_PR22;
43.
44.         if(BB(COMP2->CSR, COMP2_CSR_COMP2OUT)) {
45.             wave_pin(GPIOC, PC10);
46.         } else {
47.             wave_pin(GPIOC, PC12);
48.         }
49.     }
50. }
51.
52. void TIM6_DAC1_IRQHandler(void){
53.     static uint32_t cnt;
54.
55.     if(BB(TIM6->SR, TIM_SR UIF)){
56.         TIM6->SR = ~TIM_SR UIF;
57.         if(++cnt == 4095){
58.             cnt=0;
59.             wave_pin(GPIOC, PC11);
60.         }
61.     }
62. }
```

1 - 5) taka mała funkcja pomocnicza do machania wyprowadzeniami. Ustawia stan wysoki na pinie, a po małym opóźnieniu przywraca stan niski. Funkcja przyjmuje dwa argumenty: wskaźnik na strukturę opisującą port (z pliku nagłówkowego mikrokontrolera) i maskę pinu. Wywołanie funkcji można sobie obejrzeć np. w linijce 45 :)

Dwie sprawy. Po pierwsze primo przypominam, że bit banding nie obejmuje rejestrów GPIO w F334. Dlatego też wykorzystałem atomiczne rejesty BSRR, BRR³⁰⁹... Po drugie, do uzyskania opóźnienia wykorzystałem instrukcje barierowe DSB. W żadnym razie nie są tu potrzebne akurat te instrukcje. Może tam być cokolwiek co zajmie procesor na kilka cykli (wywoła opóźnienie) i nie da się wy-optymalizować kompilatorowi. Cokolwiek! Ja z przyzwyczajenia wrzuciłem akurat DSB.

11) zwracam uwagę na włączenie zegara bloku SYSCFG. Komparatory nie mają osobnego bitu włączającego zegar. Zamiast tego trzeba włączyć SYSCFG. Przynajmniej tak sugeruje dokumentacja... sprawdziłem, faktycznie bez SYSCFG nie działa :]

13 - 19) konfiguracja wyprowadzeń:

- PC10 - tą nóżką będziemy machać w przerwaniu komparatora przy rosnącym zboczu sygnału wyjściowego
- PC12 - tą nóżką będziemy machać w przerwaniu komparatora przy opadającym zboczu sygnału wyjściowego
- PC11 - tą nóżką będziemy machać w przerwaniu timera (co 4095 wyzwoleń DACa)
- PA4 - to jest wyjście przetwornika DAC i jednocześnie wejście odwracające komparatora
- PA7 - wejście nieodwracające komparatora (tu doprowadziłem stałe napięcie, z dzielnika napięcia, równe około $\frac{1}{2} V_{DDA}$)
- PA12, PB9 - to są wyjścia komparatora (komparator ma oczywiście tylko jedno wyjście, ale można je wyprowadzić na kilka pinów)

Zahaltujmy się na chwilę przy tych wyjściach komparatora. Zgodnie z tabelą 20.1 trzy nóżki mikrokontrolera mogą, w ramach funkcji alternatywnej, współpracować z wyjściem komparatora (PA2, PA12, PB9). PA2 sobie darujemy, bo to jest, alternatywnie, linia USARTu2 i jest domyślnie połączona z USARTem ST-Linka. PA12 nie budzi żadnych wątpliwości. PB9 natomiast jest „wątpliwe” - odsyłam do przypisu 306. Dlatego też, przy okazji tego zadania, przetestujemy sobie wyjście na PB9.

21) konfiguracja DACa to dla nas pestka z masłem. Kolejne bity odpowiadają za: włączenie przetwornika, ustawienie amplitudy generowanego przebiegu (maks.), wybór generowanego przebiegu (trójkąt), włączenie wyzwalania zewnętrznym trygierzem. Za wybór konkretnego trygierza odpowiadają bity pola TSELx. W programie wykorzystuję licznik TIM6 co jest domyślną opcją, stąd też nic nie ustawiam.

309 oczywiście najpierw napisałem kod z wykorzystaniem BB i się odbiłem od ściany...

23) a oto i nasz bohater - pan komparator. Tu się na chwilę zatrzymamy. Na początku zwróćmy uwagę na nazwy rejestrów i bitów. Mikrokontroler posiada kilka komparatorów. Z jakichś mistycznych powodów, ST postanowiło w nazwach bitów umieszczać numer komparatora. Przykładowo COMP2_CSR_COMP2EN. Żeby było śmieszniej, w nagłówku zdefiniowane są również nazwy bez tych numereków, np. COMP_CSR_COMPxEN. Np. weźmy sobie bit odpowiedzialny za blokowanie konfiguracji przetwornika (COMPxLOCK). Odnoszą się do niego cztery różne definicje (wszystkie są jednakowe - to maski 31-go bitu rejestru):

- COMP_CSR_COMPxLOCK;
- COMP2_CSR_COMP2LOCK;
- COMP4_CSR_COMP4LOCK;
- COMP6_CSR_COMP6LOCK;

Ja tam nie przepadam za takim nadmiarowym bałaganem i zostawiłbym tylko te ogólne.

Żeby było jeszcze śmieszniej, w nagłówku poza układami COMP2, COMP4, COMP6 zdefiniowana jest dodatkowa instancja bez numerka - po prostu COMP. Przykład:

```
COMP->CSR = COMP_CSR_COMPxEN;
```

Powyższy zapis spowoduje włączenie, którego komparatora? > A drugiego, bo w nagłówku jest „przekierowanie” definicji COMP do COMP2. Po co? Przypuszczam, że chodzi o kompatybilność z innymi mikrokontrolerami rodziny. Ale to tylko domysły. Tak czy siak postanowiłem wspomnieć, żeby nie było niespodzianki. Jedziemy dalej!

W rejestrze konfiguracyjnym komparatora ustawiam trzy bity. Ten z „lock” w nazwie odpowiada za zamrożenie konfiguracji komparatora. Po ustawieniu tego bitu, rejestr COMPx_CSR staje się tylko do odczytu. Odblokuje go dopiero reset mikrokontrolera. Kolejny bit odpowiada za wybór źródła dla odwracającego wejścia komparatora. Źródeł jest kilka (odsyłam do RMa oraz tabeli 20.1). My ustawiamy PA4/DAC1_CH1. Z ciekawostek: w RMie te bity nazywają się COMP2INMSEL zaś w pliku nagłówkowym jest COMP2INSEL. Zjadło się eM komuś. Ostatni bit odpowiada za włączenie komparatora. Po włączeniu zapewne należałoby odczekać jakiś „wake up time”... ale my nie będziemy przejmować się pierdołami. Jedziemy po bandzie :]

24) to dla sprawdzenia, czy blokowanie konfiguracji rzeczywiście działa. We wcześniejszej linijce włączliśmy funkcję „blokowania” konfiguracji komparatora, gdyby funkcja nie zadziałała, to ta linijka wyzerowałaby całą konfigurację i komparator by nie działał.

26 - 30) konfiguracja licznika TIM6. Czemu ten licznik? A czemu nie :) Oczywiście wybrany był w oparciu o tabelkę *External triggers (DAC1)*. Nic ciekawego tu nie ma. Licznik coś tam zlicza, generuje przerwania przy przepełnieniu, wysyła sygnał TRGO do DACa i tyle.

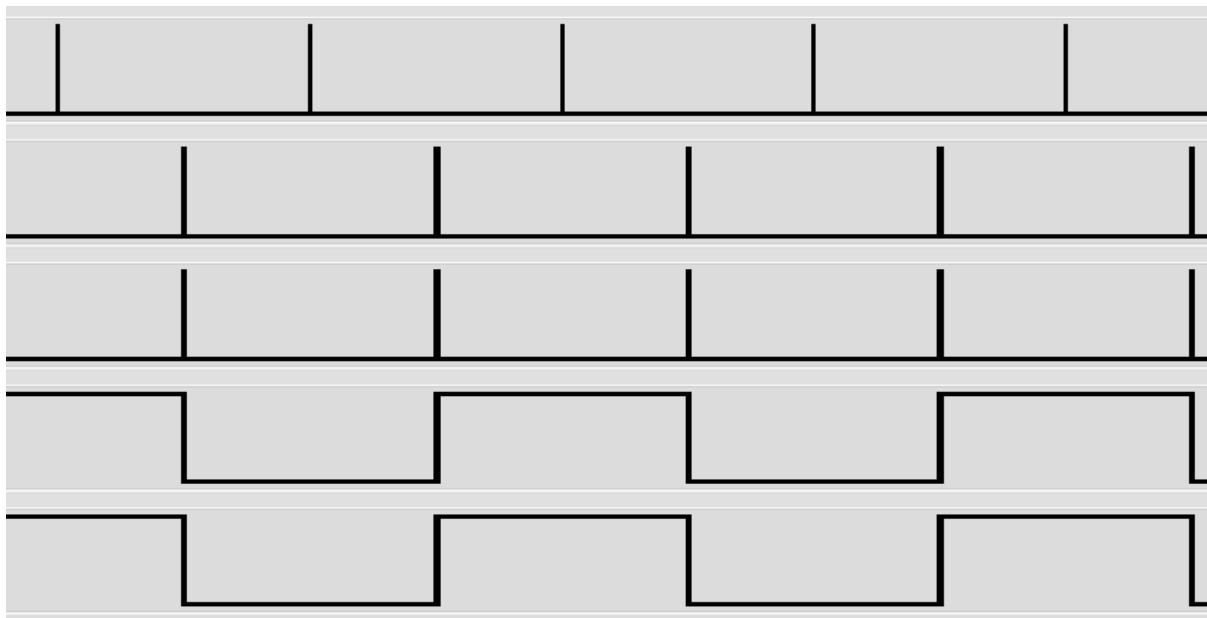
32 - 35) wyjście komparatora połączone jest z kontrolerem przerwań EXTI - linia EXTI22. W rejestrach kontrolera wybieramy na jakie zbocza ma reagować (analogicznie jak w przypadku przerwań zewnętrznych opisanych w rozdziale 7) i odmaskowujemy przerwanie. Na koniec, standardowo, włączamy w kontrolerze NVIC.

40 - 50) przerwanie komparatora. Pierwszy punkt programu to (jak zawsze) sprawdzenie flagi przerwania i jej skasowanie. Jak widać na listingu, flaga znajduje się w bloku EXTI. Rejestr konfiguracyjny komparatora (COMPx_CSR) nie zawiera żadnych flag związanych z przerwaniami (ani „włącznika” przerwań, ani flagi oczekiwania przerwania).

W programie włączone zostały przerwania od obu zbocz sygnału (pacz linijka 32 i 33). Niestety nie ma żadnego sprzętowego mechanizmu, który pozwoliłby automatycznie zidentyfikować zbocze, które wywołało przerwanie. Musimy sobie poradzić na piechotę. W tym celu sprawdzany jest stan bitu COMPxOUT. Odzwierciedla on aktualny stan sygnału wyjściowego z komparatora. Przerwanie zostało wywołane przez jakieś zbocze, więc jeśli zbadamy aktualny stan sygnału, to będziemy widzieli co to było za zbocze. Przynajmniej mamy taką nadzieję... W zależności od stanu sygnału/zbocza machany jest inny pin mikrokontrolera.

52 - 62) na koniec zostało przerwanie licznika. Tu nie ma nic nowego. Przerwanie jest zgłasiane co przepełnienie licznika, czyli przy każdym wyzwoleniu przetwornika DAC. ISR co 4095 przepełnień woła funkcję machającą pinem. Cel tego machania był już omówiony, więc nic nas nie zaskakuje ;)

No to czas na efekty naszych zmagań.



Rys. 20.2. Zabawa z komparatorem (*od góry: PC11, PC10, PC12, PA12, PB9*)

I jak? Oczywiście nie zgadza się z tym czego oczekiwaliśmy. Licząc od góry, drugi i trzeci przebieg to piny machane w zależności od zbocza sygnału wyjściowego komparatora. Szpilki powinny być na zmianę - raz zbocze rosnące, potem malejące... Co tu się zadziało? Ano życie. Bawimy się komparatorem analogowym. Jedną nogę wystawiliśmy poza idealny, synchroniczny, cyfrowy świat zer i jedynek. Co gorzej nasz układ z komparatorem nie posiada żadnej histerezy (programowalną histerezę posiadają m.in. komparatory w niektórych STM32F37x i STM32F30x). Efekt jest taki, że zamiast jednego, ładnego zbocza sygnału wyjściowego, mamy przez chwilę (kilkadziesiąt mikrosekund) sieczkę. Dopadają nas problemy podobne do tych związanych z drganiami elementów stykowych. Jak masz ochotę to spróbuj „zdeboundingować” komparator :)

Zadanie domowe 20.2: z poprzedniego przykładu wyrzucamy oba przerwania. Zamiast tego dokładamy drugi timer, który będzie sprzętowo generował PWM. Komparator spinamy z tym timerem tak, aby sygnał z komparatora (wysoki) wyłączał PWM. Gdy wyjście komparatora wróci do stanu niskiego, PWM ma się znowu generować³¹⁰.

Żeby nie było tak abstrakcyjnie, taki układ mógłby realizować zabezpieczenie nadprądowe np. silnika. PWM steruje jakąś energoelektroniką, która zasila silnik. Do komparatora doprowadzony jest sygnał proporcjonalny do prądu uzwojenia. Gdy prąd za bardzo wzrośnie, to PWM jest wyłączany.

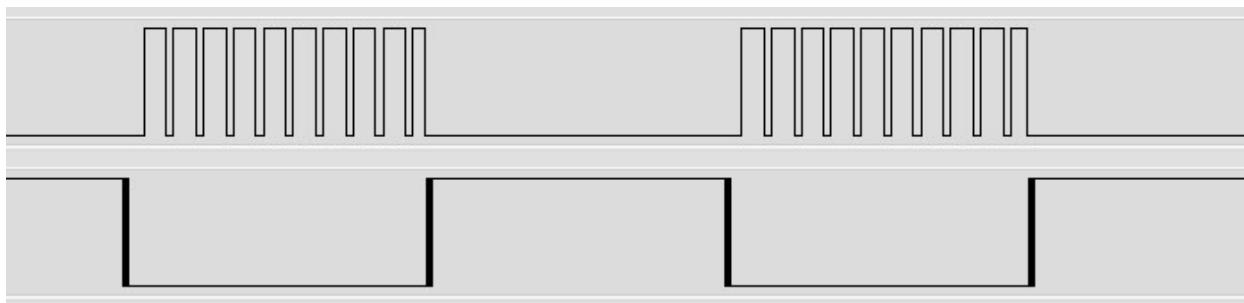
³¹⁰ przypominam o możliwości zmiany polaryzacji wyjścia komparatora (patrz COMPx_CSR_COMPxPOL)

Przykładowe rozwiązanie (F334):

```
1. int main(){
2.
3.     RCC->AHBENR |= RCC_AHBENR_GPIOAEN;
4.     RCC->APB1ENR = RCC_APB1ENR_DAC1EN | RCC_APB1ENR_TIM6EN;
5.     RCC->APB2ENR = RCC_APB2ENR_SYSCFGEN | RCC_APB2ENR_TIM1EN;
6.
7.     gpio_pin_cfg(GPIOA, PA4, gpio_mode_analog); //dac1_ch1 / comp2_in-
8.     gpio_pin_cfg(GPIOA, PA7, gpio_mode_analog); //comp2_in+
9.     gpio_pin_cfg(GPIOA, PA8, gpio_mode_AF6_PP_LS); //tim1_ch1
10.    gpio_pin_cfg(GPIOA, PA12, gpio_mode_AF8_PP_LS); //comp2_out
11.
12.    DAC1->CR = DAC_CR_EN1 | DAC_CR_MAMP1 | DAC_CR_WAVE1_1 | DAC_CR_TEN1;
13.    COMP2->CSR = COMP2_CSR_COMP2INSEL_2 | COMP2_CSR_COMP2OUTSEL_1 | COMP2_CSR_COMP2OUTSEL_2 |
14.        COMP2_CSR_COMP2EN;
15.
16.    TIM6->ARR = 100;
17.    TIM6->CR2 = TIM_CR2_MMS_1;
18.    TIM6->CR1 = TIM_CR1_CEN;
19.
20.    TIM1->ARR = 1000;
21.    TIM1->CCR1 = 250;
22.    TIM1->CCMR1 = 0b111<<4 /* PWM mode 2 */ | TIM_CCMR1_OC1CE;
23.    TIM1->CCER = TIM_CCER_CC1E;
24.    TIM1->BDTR = TIM_BDTR_MOE;
25.    TIM1->CR1 = TIM_CR1_CEN;
26.
27.    while(true);
28. }
```

Coś się zmieniło w kodzie, poza przybyciem TIM1? A tak! Doszło ustawienie wyjścia komparatora (COMPx_OUTSEL). Z kolei w konfiguracji licznika „nietypowy” jest tylko bit OC1CE. Posiłkując się schematami blokowymi licznika i zdobytym doświadczeniem, obczaj to sam ;)

Efekt działania przedstawia się jak poniżej. Wysoki stan na wyjściu komparatora powoduje natychmiastowe (bez czekania do końca trwającego okresu) wyzerowanie wyjścia PWM.



Rys. 20.3. PWM przerywany komparatorem (*od góry: TIM1_CH1; COMP_OUT*)

Co warto zapamiętać z tego rozdziału:

- komparator pozwala na sprzętową realizację zabezpieczeń nadprądowych, termicznych, itp.
- komparator może wybudzać mikrokontroler z trybu uśpienia
- jest prosty (tylko jeden rejestr konfiguracyjny)
- komparator może bezpośrednio współpracować z przetwornikiem DAC i licznikami

- noty aplikacyjne dotyczące komparatorów:
 - *AN3248 Using STM32L1 analog comparators in application cases*
 - *AN4112 Using STM32F05xx analog comparators in application cases*
 - *AN4232 Getting started with analog comparators for STM32F3 series*

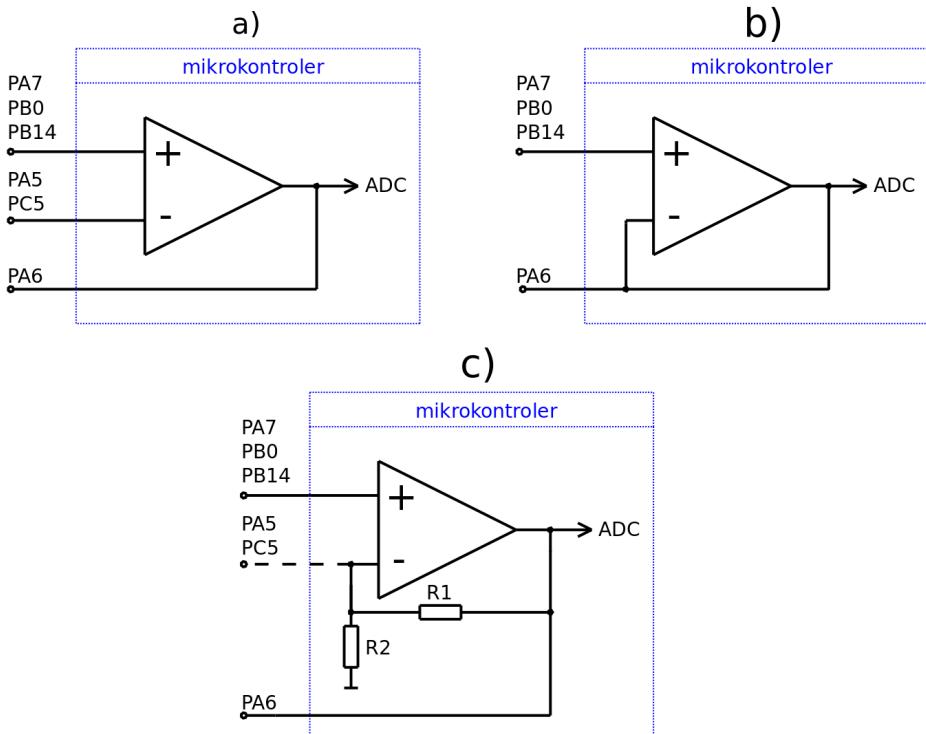
20.2. Wzmacniacz operacyjny (F334)

Tym razem to już w ogóle opuszczamy bezpieczny cyfrowy świat. Mikrokontroler z białej śnieżynki wyposażony jest w jeden układ wzmacniacza operacyjnego (OPAMP²). W pliku nagłówkowym, podobnie jak w przypadku komparatora, mamy dwa zestawy definicji - z sufiksem „2” i bez. Jako że mamy tylko jeden wzmacniacz, proponuję używanie wersji krótszej (bez tej dwójki). Kto jest za? Kto przeciw? Kto się wstrzymał? Wniosek jednogłośnie zatwierdzono.

Jeżeli, drogi Czytelniku, masz nikłe pojęcia na temat wzmacniaczy operacyjnych (tak jak ja), to to jest dobry moment, żeby się doszkolić we własnym zakresie³¹¹. Wzmacniacz STMowy może pracować w trzech podstawowych konfiguracjach:

- jako samodzielny układ (*standalone*) - w tej wersji, wszystkie wyprowadzenia sygnałowe wzmacniacza (dwa wejścia i wyjście) są dostępne poprzez nóżki mikrokontrolera (rys. 20.4 a)
- wtórnik napięciowy (*follower*) - wejście odwracające jest sprzętowo połączone z wyjściem wzmacniacza (rys. 20.4 b)
- wzmacniacz o wzmacnieniu ustawianym programowo (*Programmable Gain Amplifier - PGA*) - wzmacniacz pracuje w konfiguracji wzmacniacza nieodwracającego, przy czym rezystory ustalające wzmacnienie sygnału konfigurowane są programowo (rys. 20.4 c)

³¹¹ liczę na to, że wystarczy nam: https://pl.wikibooks.org/wiki/Wzmacniacze_operacyjne ;)



Rys. 20.4. Tryby pracy wzmacniacza operacyjnego (a - standalone, b - follower, c - programmable gain amplifier)

Do czego możemy wykorzystać wzmacniacz? Zdecydowanie najbardziej oczywistym zastosowaniem jest „kondycjonowanie” sygnałów analogowych przed pomiarem przez ADC. W rozdziale 13.3 starałem się omówić (najlepiej jak potrafiłem!) problem wynikający z konieczności naładowania pojemności w obwodzie wejściowym przetwornika. Dla przypomnienia: jeżeli impedancja źródła mierzonego sygnału jest zbyt duża, to kondensator nie zdaży się naładować i wyniki konwersji będzie sobie można generalnie wsadzić psu w budę. I tutaj właśnie możemy wykorzystać wzmacniacz pracujący w konfiguracji wtórnika napięciowego (rys. 20.4 b). Wejście wtórnika ma bardzo dużą impedancję (nie będzie obciążać źródła sygnału), zaś jego wzmacnienie wynosi 1. Cytując wikipedię „*układы te są stosowane w celu odseparowania źródła sygnału od odbiornika*” (źródło: patrz przypis 311).

Idąc dalej, może nam się przydarzyć taka niefrasobliwa sytuacja, że amplituda mierzonego przebiegu/napięcia będzie mała w stosunku do zakresu pracy przetwornika ADC. Czytelnicy mający doświadczenie i wiedzę na temat przetwarzania sygnałów, z pewnością zgodzą się³¹², że zależy nam na tym aby dopasować mierzony sygnał do zakresu pracy przetwornika. Jeśli ADC pracuje z napięciem referencyjnym np. 3,3V to bezpośrednie mierzenie sygnału o amplitudzie powiedzmy 10mV jest niefortunnym pomysłem. Warto byłoby go wzmacnić. Można wykorzystać do tego wzmacniacz pracujący w konfiguracji PGA (Programmable Gain Amplifier) - rys. 20.4 b lub w konfiguracji niezależnej/samodzielnnej rys. 20.4 a. Przy czym w tym drugim przypadku o

312 a jak nie to niech śłą wiadomości z wyjaśnieniem czemu się nie zgadzają :)

niezbędne dodatkowe elementy układu musimy już zadbać sami. W wersji PGA mamy do wyboru cztery, zdefiniowane na sztywno, wartości wzmacnienia - patrz macierz 20.2.

Tabela 20.2 Wzmocnienie PGA (oznaczenia zgodnie z rys. 20.4)

Wzmocnienie [-]	R1 [kΩ]	R2 [kΩ]
2 ±1%	5,4	5,4
4 ±1%	16,2	5,4
8 ±1%	37,8	5,4
16 ±1%	40,5	2,7

Pokazane powyżej trzy konfiguracje nie wyczerpują tematu. Można je nieco pomieszać i np. do PGA dodać dodatkowe elementy pętli sprzężenia zwrotnego. Przerywana linia na rys. 20.4 c symbolizuje możliwość wyprowadzenia sygnału wejścia odwracającego na pin mikrokontrolera. To na co warto zwrócić uwagę to to, że niezależnie od wybranej konfiguracji wyłączenie wzmacniacza zawsze, „bezapelacyjnie, do samego końca”, anektuje wyprowadzenie PA6. Od teraz jest to analogowe wyjście wzmacniacza bez względu na konfigurację rejestrów GPIO. Nie można wykorzystać tego pinu jako cyfrowego I/O. Ponadto, wyjście wzmacniacza, idzie sobie bezpośrednio do ADC. Dokładniej do trzeciego kanału drugiego przetwornika (ADC2_IN3).

Zachowanie pozostałych nóżek związanych ze wzmacniaczem (wejścia) wynika z konfiguracji wzmacniacza. Jeśli któraś nóżka została wybrana do pracy jako wejście OPAMPA (w rejestrze OPAMP_CSR) to automatycznie jest ona przejmowana przez ten układ i jej konfiguracja w rejestrach GPIO nie ma nic do gadania. Podobna sytuacja miała miejsce np. z wyjściem układu DAC. Również, podobnie jak wtedy, zalecane jest wcześniejsze przestawienie pinu w tryb analogowy, aby zminimalizować jakieś tam pasożytnicze prądy i inne niegodziwości.

Popatrzmy co tam dalej mamy ciekawego... A! Wzmacniacz wymaga kalibracji. Kalibracje należy wykonać dwukrotnie - osobno dla dwóch par różnicowych (NMOS i PMOS). Nie ma chyba sensu omawiać tej kalibracji teraz „na sucho” - wyjdzie w przykładzie. Jeżeli nie mamy ochoty na kalibrowanie, to możemy wykorzystać domyślne wartości kalibracyjne. Przypuszczam, że będzie się to wiązać z większym błędem offsetu albo czymś takim o zbliżonej nazwie.

Pozostałe bajery:

- funkcja blokowania konfiguracji
- multiplekser wejściowy sterowany licznikiem

Pierwsza kropka raczej nie powinna budzić wątpliwości. Działa to tak samo jak w przypadku komparatora. Druga kropka też jest całkiem prosta. Możemy skonfigurować dwie pary nóżek, które mają pracować jako wejścia wzmacniacza. Na przykład (patrz rysunek 20.4):

- pierwsza para:
 - PA7 - wejście nieodwracające
 - PA5 - wejście odwracające
- druga para:
 - PB14 - wejście nieodwracające
 - PA5 - wejście odwracające (nie jest powiedziane, że obie nóżki muszą się zmieniać w drugiej parze)

Do tego wybieramy źródło sterujące multiplekserem (przełącznikiem wejść). Wróć... „wybieramy” to za duże słowo, bo do wyboru mamy tylko kanał 6 licznika 1... W każdym razie sygnał z TIM1_CC6 będzie powodował automatyczne przełączanie par wejść. No... i w sumie tyle :)

To na koniec jeszcze kilka podstawowych danych elektrycznych i przejdziemy do naszej ulubionej części rozdziału :]

Tabela 20.3 Parametry wzmacniacza operacyjnego (na podstawie datasheetu F334)

Parametr	Oznaczenie	Wartość
zakres napięć wejściowych	V_{in}	$0 - V_{DDA}$
maksymalny offset po kalibracji	$V_{I_{offset}}$	$3mV$
minimalna rezystancja obciążenia	R_{LOAD}	$4k\Omega$
maksymalna pojemność obciążenia	C_{LOAD}	$50pF$
maksymalny prąd wyjścia	I_{LAOD}	$500\mu A$
pasmo	GBW	$8,2MHz$
pasmo w konfiguracji PGA	PGA BW	$4MHz$ (wzmocnienie x2) $2MHz$ (wzmocnienie x4) $1MHz$ (wzmocnienie x8) $0,5MHz$ (wzmocnienie x16)
maksymalny czas strojenia	$t_{OFFTRIM}$	$2ms$
maksymalny czas wybudzania	t_{WAKEUP}	$5\mu s$
minimalny czas próbkowania wyjścia przez ADC	$t_{S_OPAM_VOUT}$	$400ns$

Zadanie domowe 20.3: za pomocą przetwornika ADC mierzymy sygnał pochodzący ze świata zewnętrznego (np. jakiś sinus bo ładny) o niewielkiej amplitudzie³¹³. Równocześnie mierzymy ten sam sygnał przepuszczony przez wzmacniacz operacyjny, pracujący w konfiguracji PGA. Wzmocnienie należy dobrać adekwatnie do amplitudy „posiadanej” sygnału. Oczywiście przeprowadzamy pełną kalibrację wzmacniacza i ADC. Wyniki niech będą przesyłane do dwóch buforów (jeden na pomiary bezpośrednie, drugi na pomiary ze wzmacnieniem). Zawartość buforów, na koniec, odczytamy debuggerem i wyplotujemy na PC. Do dzieła!

Przykładowe rozwiążanie - inwokacja: tym razem, dla odmiany, przytachałem sobie z piwnicy generator funkcyjny. Tak więc w moim rozwiążaniu testowy sygnał (ślus) będzie generowany przez zewnętrzne urządzenia. Ale nie traktuj tego jako wymówki, aby nie rozwiązywać zadania samemu! Nic (poza ewentualnym lenistwem) nie stoi na przeszkodzie, aby wykorzystać wbudowany przetwornik DAC mikrokontrolera. Także do roboty! :)

Treść zadania nie narzuca zastosowania konkretnych rozwiązań. Można np. wykorzystać jeden ADC mierzący oba sygnały albo dwa ADC działające równolegle. Konwersje mogą być przeprowadzane w ramach grupy regularnej albo wstrzykiwanej. Wyniki mogą być przesyłane przez DMA albo odbierane w przerwaniu. Wolna wola, amerykanka i demokracja Panie! Fajnie by było, jakbyś zatrzymał się w tym momencie i przemyślał „za i przeciw” różnych rozwiązań. Jak Ty byś to zrobił? No to pomyśl, a ja w tym czasie popracuję nad moją wersją ;)

[... kilka dni później ...]

Zdecydowałem się na użycie jednego przetwornika mierzącego oba kanały. Jest to oczywiście podyktowane lenistwem - skonfigurowanie jednego przetwornika to mniej kodu do napisania. Ponadto zrezygnowałem z wykorzystania DMA. Wyniki mają być zapisane w dwóch osobnych tablicach, a skoro zdecydowałem się na jeden przetwornik, to DMA nie będzie w stanie „rozdzielić” wyników do osobnych tablic. Zmierzone wartości dla obu kanałów wylądowałyby naprzemiennie w jednym buforze. Czyli potem trzeba by się „babrać” w ich rozdzielenie... nie nie nie. Lenistwo zwyciężyło i zdecydowałem się na odbieranie wyników w przerwaniu ADC. Grupa regularna czy truskawkowa? > Aż się prosi o strzykawkową. Dlatego, że w tej konfiguracji wyniki konwersji poszczególnych kanałów są zapisywane w osobnych rejestrach danych (JDRx), więc nie trzeba będzie, w przerwaniu, pilnować numeru mierzzonego aktualnie kanału.

No to już wiesz, jak ja się zabräłem do tego programu. Przy czym, gdybym miał to napisać jeszcze raz, to jednak zdecydowałbym się na DMA. Ale o tym później. Uwaga! Mam nową zabawę :] W kodzie, który poniżej zaprezentuję znajduje się kilka błędów. Postaraj się je wyłapać.

³¹³ zwracam uwagę na to, że nasz przebieg nie może być symetryczny względem poziomu masy - cały powinien być przesunięty „nad” poziom 0 :)

Kod jest dosyć długi jak na Poradnikowe przykłady, więc dla ułatwienia analizy (żeby nie latać po dokumencie) jest pocięty na kilka kawałków.

Przykładowe rozwiązanie - funkcja główna (F334):

```
1. static volatile uint16_t adc_direct[2560];
2. static volatile uint16_t adc_amplifier[2560];
3.
4. int main(void){
5.
6.     RCC->AHBENR |= RCC_AHBENR_GPIOAEN | RCC_AHBENR_ADC12EN;
7.     RCC->APB1ENR = RCC_APB1ENR_TIM6EN;
8.     RCC->APB2ENR = RCC_APB2ENR_SYSCFGEN;
9.
10.    gpio_pin_cfg(GPIOA, PA5, gpio_mode_out_PP_LS); //dioda
11.    gpio_pin_cfg(GPIOA, PA6, gpio_mode_analog); //opamp out
12.    gpio_pin_cfg(GPIOA, PA7, gpio_mode_analog); //adc2_in4, comp_in+
13.
14.    /* TIM6 - delay */
15.    TIM6->PSC = 8000-1;
16.    TIM6->EGR = TIM_EGR_UG;
17.
18.    uint8_t trimoffset = 0;
19.    uint32_t tmp_reg;
20.
21.    tmp_reg = OPAMP_CSR_OPAMPxEN | OPAMP_CSR_USERTRIM | OPAMP_CSR_CALON | OPAMP_CSR_CALSEL;
22.
23.    do {
24.        OPAMP->CSR = tmp_reg | (trimoffset & 0x1fu) << 24;
25.        trimoffset++;
26.        delay_ms(2);
27.    } while(BB(OPAMP->CSR, OPAMP_CSR_OUTCAL));
28.
29.    trimoffset = 0;
30.    tmp_reg = OPAMP->CSR & ~OPAMP_CSR_CALSEL_1;
31.
32.    do {
33.        OPAMP->CSR = tmp_reg | (trimoffset & 0x1fu) << 19;
34.        trimoffset++;
35.        delay_ms(2);
36.    } while(BB(OPAMP->CSR, OPAMP_CSR_OUTCAL));
37.
38.    OPAMP->CSR &= ~(OPAMP_CSR_CALSEL | OPAMP_CSR_CALON);
39.    OPAMP->CSR |= OPAMP_CSR_PGGAIN_0 | OPAMP_CSR_VMSEL_1 | OPAMP_CSR_VPSEL;
40.
41.    /* Zegar ADC = HCLK/2 = 4MHz */
42.    ADC1_2_COMMON->CCR = ADC12_CCR_CKMODE_1;
43.    adc2_calib();
44.    adc2_start();
45.    adc2_cfg();
46.    ADC1->CR |= ADC_CR_JADSTART;
47.
48.    /* TIM6 - ADC injected trigger */
49.    RCC->APB1RSTR = RCC_APB1RSTR_TIM6RST;
50.    RCC->APB1RSTR = 0;
51.
52.    TIM6->ARR = 10;
53.    TIM6->EGR = TIM_EGR_UG;
54.    TIM6->CR2 = TIM_CR2_MMS_1;
55.    TIM6->CR1 = TIM_CR1_CEN;
56.
57.    NVIC_EnableIRQ(ADC1_2_IRQn);
58.
59.    while ( ADC2->CR & ADC_CR_ADEN );
60.
61.    GPIOA->BSRR = GPIO_BSRR_BS_5;
62.    __BKPT();
63.    while(1);
64. }
```

1, 2) dwie tablice na wyniki. Wielkość tablic dobrąłem tak, aby pi razy drzwi zapełnić całą dostępną pamięć operacyjną układu.

6 - 8) włączamy zegar dla wykorzystywanych układów (port A, przetwornik ADC, licznik TIM6 oraz blok SYSCFG). Podobnie jak w przypadku komparatora, wzmacniacz operacyjny nie posiada dedykowanego bitu włączającego sygnał zegarowy. Należy za to włączyć blok SYSCFG.

10 - 12) konfiguracja wyprowadzeń mikrokontrolera. Dioda podłączona do PA5 jest wykorzystywana jako wskaźnik zakończenia pomiarów. Sygnał testowy (nasz sinusoidus) jest doprowadzony do wejścia nieodwracającego wzmacniacza - nóżka PA7. Tak jak wspominałem wzmacniacz, po włączeniu, automatycznie przejmuje kontrolę nad nóżką PA6 (wyjście wzmacniacza). Czy nam się to podoba czy nie :) Wyprowadzenie konfigurujemy jako *analogowe* aby pozbyć się pasożytniczych prądów czy czegoś tam.

Spora część programu, związana z uruchomieniem i konfiguracją przetwornika ADC, została skopiowana z rozwiązania zadania 13.12. Po szczegółowe opisy, odsyłam do tamtego przykładu.

15, 16) „wstępna” konfiguracja licznika TIM6. Licznik będzie wykorzystywany podczas uruchamiania modułów ADC i OPAMP. Opierać się na nim będzie prosta funkcja opóźniająca (*delay_ms*). Zostanie ona omówiona później.

Rozpoczynamy zabawę w kalibrację wzmacniacza. Kalibrację należy przeprowadzić dwukrotnie: raz dla pary tranzystorów typu N i drugi raz dla pary tranzystorów typu P. Zaczynamy od pary N.

Kalibracja, w skrócie, wygląda tak, że:

- włączamy wzmacniacz (*OPAMP_CSR_OPAMPxEN*)
- ustawiamy bity *OPAMP_CSR_CALON* oraz *OPAMP_CSR_USERTRIM*
- ustawiamy pole *OPAMP_CSR_CALSEL* na wartość 0b11 dla pary typu N (lub 0b10 dla pary tranzystorów P)
- w pętli inkrementujemy (od zera) wartość pola *OPAMP_CSR_TRIMOFFSETN* (lub *TRIMOFFSETP* dla drugiej pary tranzystorów)
- po wpisaniu każdej nowej wartości oczekujemy chwilkę na jej „zadziałanie” - patrz wartość *tofftrim* z tabeli 20.3
- pętlę przerywamy³¹⁴, gdy wyzeruje się bit *OPAMP_CSR_OUTCAL*

³¹⁴ warto byłoby dodać jeszcze zabezpieczenie przerywające pętle jeśli coś z tą kalibracją nie wypali - żeby program nie zawisł w tym na amen

No to jedziemy:

21) przygotowuję sobie pomocniczą zmienną z wartością rejestru *OPAMP_CSR*. Pomocnicza wartość zawiera ustawione, wspomniane wyżej, bity.

23 - 27) ładuję moją pomocniczą wartość do rejestru *OPAMP_CSR*, oczekuję wymagane 2 ms i testuję bit *OPAMP_CSR_OUTCAL*. Całość zapętlam dopóki bit jest ustawiony. W każdym kolejnym obiegu pętli inkrementuję wartość pola *TRIMOFFSETN* (pięciobitowe pole zaczynające się od bitu 24).

Gdy zakończy się kalibracja pary N następuje powtórka z rozrywki dla drugiej pary tranzystorów. Zeruję pomocniczą zmienną z *offsetem* (linia 29), ustawiam inną wartość pola *CALSEL* (linia 30) i wchodzę w pętlę praktycznie identyczną jak poprzednio (linie 32 - 36), inne jest jedynie położenia pola *TRIMOFFSETP*. Przy czym należy uważać aby nie skasować sobie, ustawionej w poprzedniej pętli, wartości kalibracyjnej *TRIMOFFSETN* (przemyśl linijkę 30).

Kiedy kalibrację mamy już za sobą, pozostają nam jeszcze dwie sprawy:

- należy skasować bity związane z kalibracją w rejestrze *OPAMP_CSR* (linijka 38)
- należy ustawić wzmacnienie i „powiązać” wejścia wzmacniacza z nóżkami mikrokontrolera (linijką 39)

Za wybór nóżek odpowiadają pola bitowe z „SEL” w nazwie:

- VM_SEL - konfiguracja wejścia odwracającego (*M* w nazwie najprawdopodobniej oznacza *minus*), do wyboru mamy:
 - dwie nóżki (PA5, PC5)
 - wewnętrzne rezystory (konfiguracja PGA)
 - wewnętrzne połączenie z wyjściem (wtórnik)
- VP_SEL - konfiguracja wejścia nieodwracającego (*P* - *positive* albo *plus*), do wyboru są trzy nóżki mikrokontrolera (PA7, PB0, PB14)
- VMS_SEL - konfiguracja alternatywnego³¹⁵ wejścia odwracającego (*S* - *secondary*)
- VPS_SEL - konfiguracja alternatywnego³¹⁵ wejścia nieodwracającego (*S* - *secondary*)

Za konfigurację wzmacnienia PGA odpowiada pole bitowe *PGA_GAIN*. Do wyboru mamy kilkanaście pozycji. Wersje z dopiskiem „*Internal feedback connected to*” powodują dodatkowo

³¹⁵ jeżeli nie pamiętasz to cofnij się do początku tego rozdziału i przeczytaj akapit dotyczący wejściowego multipleksera sterowanego licznikiem

połączenie wejścia odwracającego z wybranym pinem mikrokontrolera (patrz linia przerywana na rysunku 20.4 c.

Na tym kończy się konfiguracja wzmacniacza a zaczyna ADC... które oczywiście mamy w małym palcu :]

42) włączenie zegara przetwornika

43) kalibracja przetwornika. Funkcja kalibrująca jest taka sama jak w rozwiązaniu zadania 13.12 (patrz [listing](#)) tylko zmienił się numer przetwornika.

44) w tej funkcji też praktycznie nic się nie zmieniło

45) funkcja konfigurującą ADC (zaraz do niej dojdziemy)

46) odpalenie przetwornika - od teraz jest gotowy na trygierze z licznika

49 - 50) przywracam domyślne (jak po resecie) ustawienia licznika TIM6

52 - 55) konfiguruję licznik do wyzwalania przetwornika ADC. Częstotliwość wyzwalania licznika została dobra na oko tak, aby tablicę wypełnić kilkoma okresami przebiegu testowego.

57) włączenie przerwania od zakończenia całej sekwencji³¹⁶ konwersji wstrzykiwanych (dwa kanały). W ISR będziemy odbierać wyniki z rejestrów przetwornika i zapisywać je w tablicach. Dodatkowo po zapełnieniu tablic wyłączymy przetwornik.

59) pętla *while* oczekująca na wyłączenie przetwornika (czyli na zakończenie pomiarów)

61 - 64) dioda, breakpoint, koniec, kurtyna w dół.

Zostały nam jeszcze drobiazgi na koniec:

Przykładowe rozwiązanie - ISR i funkcje pomocnicze (F334):

```
1. void delay_ms(uint32_t cnt){
2.     TIM6->ARR = cnt;
3.     TIM6->CR1 = TIM_CR1_OPM | TIM_CR1_CEN;
4.     while (~TIM6->SR & TIM_SR_UIF);
5.     TIM6->SR = 0;
6. }
7.
8. void adc2_cfg(void){
9.     ADC2->IER = ADC_IER_JEOS; //inna nazwa
10.    ADC2->JSQR = ADC_JSQR_JEXTEN_0 |
11.        14<<2 /* TIM6_TRGO */ |
12.        3<<8 /* CH3 */ |
13.        4<<14 /* CH4 */ |
14.        1 /* dwie konwersje */;
15.    ADC2->SMPR1 = 1<<9 | 1<<12; /* sampling = 2,5 fadc = 0,625us */
16. }
17.
18. void ADC1_2_IRQHandler(void){
19.     static uint32_t i;
20.
21.     if ( ADC2->ISR & ADC_ISR_JEOS ) {
22.         ADC2->ISR = ADC_ISR_JEOS;
23.         adc_amplifier[i] = ADC2->JDR1;
24.         adc_direct[i] = ADC2->JDR2;
25.         if(++i == 2560) ADC1->CR |= ADC_CR_JADSTP;
26.     }
27. }
```

316 przerwania mogą być również zgłasiane po każdej konwersji z osobna - patrz bit *ADCx_IER_JEOC*

1 - 6) funkcja opóźniająca. Przypominam, że w *mejnie* (linie 15, 16) licznik został skonfigurowany tak, aby zliczał z częstotliwością 1kHz (jeden tyk co milisekundę). Jeśli więc do rejestru przeładowania wpiszemy jakąś wartość x , odpalimy licznik i poczekamy na przepelenie. To to przepelenie wystąpi z grubsza po x milisekundach. TIM6 to licznik z grupy *basic*, w jego rejestrze statusowym nie ma nic poza flagą UIF, stąd pozwoliłem sobie na taki beztrosko partyzancki zapis jak w linii 5.

8 - 16) konfiguracja przetwornika ADC2: przerwanie od końca sekwencji wstrzykiwanej, wybór trygierza, zaprogramowanie konwersji dwóch kanałów, ustawienie czasu próbkowania.

18 - 27) na koniec została procedurka obsługi przerwania przetwornika AaDeCze. Tu też nie ma nic specjalnego: sprawdzenie flagi, skasowanie, przepisanie wyników do tablic, inkrementacja indeksu tablicy, sprawdzenie czy tablica jest pełna i ewentualne wyłączenie przetwornika. Banał. Jedyna ciekawostka to inna nazwa bitu w pliku nagłówkowym (*ADC_IER_JEOS*), niż w RMie (*ADC_IER_JEOSIE*).

Na początku opisu obiecałem, że pojawią się pewne **błędy w kodzie** i zachęcałem do ich samodzielnego wyłapania. Ile udało Ci się zauważyc? Założmy, że w ogóle zapomniałeś o sprawie i nie znalazłeś ani jednego :) Dodatkowo założmy, że ja też nie wiedziałem o tych błędach i wcale nie wstawiłem ich specjalnie tylko np. z nieuwagi i roztargnienia (co wcale nie jest prawdą!). No więc komplujemy nasz program³¹⁷, wgrywamy go, uruchamiamy i czekamy w podnieceniu aż zapali się dioda oznaczająca koniec pomiarów... i czekamy... czekamy... czekamy... „A planety szaleją...”. Pewna reklama w TV głosiła ongiś hasło³¹⁸, że „życie jest jedną wielką poczekalnią”. No ale w naszym programie jednak coś grubo nie gra. Co teraz? Na założenie tematu na forum przyjdzie jeszcze czas :] Spróbujmy poradzić sobie sami! Do tego nieco intelligentniej niż gapiąc się tępco w kod przez kilka godzin.

Pierwszy najbardziej oczywisty krok: uruchamiamy debugger i sprawdzamy gdzie utknął procesor. Już po chwili wiemy, że zatrzymał się na pętli oczekującej na koniec pomiarów. Pętla czeka na wyłączenie przetwornika, które następuje w przerwaniu. Przerwania bywają zdradliwe, więc dla pewności ustawmy sobie pułapkę na początku ISR, żeby sprawdzić czy procesor w ogóle skacze do tej procedury. Nie skacze! To już coś. W *default handlerze* też nie ląduje, więc to nie jest problem wynikający np. ze złej nazwy przerwania. Chodzi o coś innego. No to zacznijmy od „źródła”. Motorem wszystkich działań w programie jest licznik TIM6, zobaczymy czy on się „kręci”. W tym celu, w debuggerze, podglądam wartości rejestrów licznika. A dokładniej podglądam rejestr

317 swoją drogą zadziwiające jest, jak dużo osób wychodzi z założenia, że „jeśli program kompluje się bez błędów to powinien dobrze działać”

318 notabene według mnie szalone trafne

CNT, potem na chwilę uruchamiam program, znowu go pauzuję i patrzę czy wartość licznika się zmienia. Zmienia. Flaga *UIF* też jest ustawiona co dodatkowo potwierdza, że licznik się kręci i przepelnia.

Konfiguracja licznika jest banalna i wygląda ok, więc idziemy szukać dziury gdzie indziej. ADC. Skoro licznik działa a nie ma żadnych przerwań, to najprawdopodobniej konwersje nie są w ogóle wyzwalane. Przetwornik ma sporo rejestrów i łatwo coś przegapić, ale ja jestem w tej sympatycznej sytuacji, że 75% kodu skopiowałem z rozdziału **13.8**. A tam przykład w najlepsze działał z licznikiem TIM6. Ha! Mam pomysł - przetwornik ma różne zabezpieczenia przed nadpisaniem danych (*overrun*), może właśnie coś takiego się uruchomiło i zablokowało jego działanie? Szybkie sprawdzenie rejestrów ADC i żadna flaga typu *OVR* nie rzuca się w oczy. Jeszcze raz patrzę na kod włączający ADC i... no jasne, banalny błąd *Copy'ego & Paste'a* - linia 46 - to nie ten przetwornik! Poprawiamy, komplujemy, wgrywamy, odpalamy. I znowu napotykamy czteroliterowe drzewo z rodziny ślazowatych, o którym pisał J. Kochanowski. Wyeliminowanie jednego problemu, odsłoniło drugi. Tym razem, owszem, wpadamy do ISR wyjątku... ale imię jego *default handler*.

Pierwszy podejrzany to znowu nazwa przerwania. Ponownie więc ustawiam pułapkę na początku ISR żeby zobaczyć czy procesor w ogóle tu dociera. Dociera. No to krokuję (*step* w debuggerze) całe ISR - nic złego się nie dzieje. Kilka razy uruchamiam (*run*) program z aktywną pułapką, za każdym razem wchodzi do ISR. No to kolejnym podejrzewanym jest przepisywanie danych do tablic. Tablicom i wskaźnikom zawsze źle z oczu patrzy. Może tu coś jest spaprane i program wyjeżdża za tablice przez co zamazuje zawartość ramu? Warto byłoby zobaczyć jak wygląda kilka końcowych zapisów do buforów. Nigdy nie pamiętam jak się ustawia taką pułapkę, która się aktywuje tylko gdy spełniony jest określony warunek (np. dotyczący wartości zmiennej)³¹⁹, więc po prostu podmieniam wartość zmiennej *i* na 2555 i patrzę co się dzieje z końcowymi zapisami do tablic. Ha, mam Cię! Wartość zmiennej wzrosła grubo powyżej granicy 2560 a przerwanie dalej jest zgłaszane. Jak nietrudno zauważyc, w 25 linii ISR jest ten sam błąd co poprzednio - nie podmieniłem numeru przetwornika po skopiowaniu kodu. Poprawiamy.

Poprawiliśmy, ale jeszcze coś jest nie teges. Niby wszystko działa, ale program nie przechodzi pętli z linii 59. Tu już sprawa jest oczywista - w pętli sprawdzany jest nie ten bit co należy. Zamiast *ADEN* powinno być *JADSTART*.

No! Teraz program działa prawie tak jak chciałem. „Prawie” wynika z obecności jeszcze jednego błędu. Tym razem jest to grubszy błąd. Patrz na konfigurację licznika wyzwalającego przetwornik ADC. Częstotliwość wyzwalania ustawiono na około 727kHz. Niby nie dużo dla takiego mikrokontrolera, w końcu ADC może zbierać do kilku milionów próbek na sekundę. Na 319 dodatkowo ma ona tę wadę, że program debugowany z taką pułapką działa koszmarnie wolno

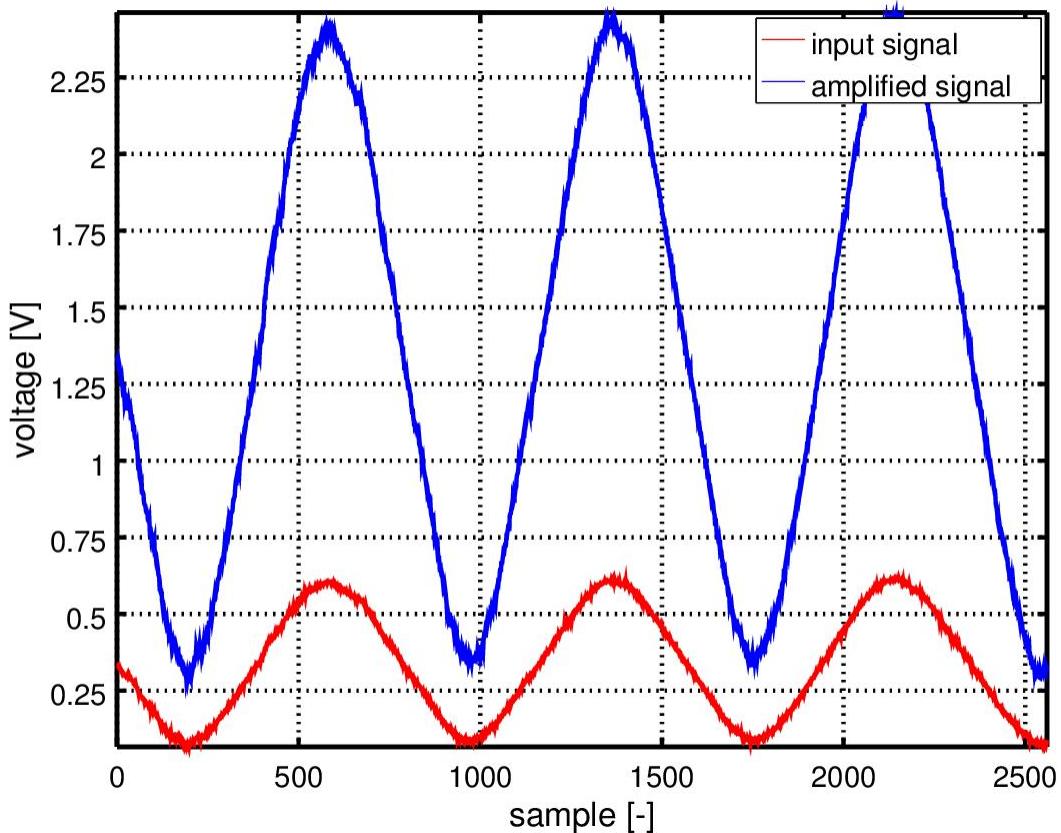
czym polega problem? Ano polega na tym, że w programie, zamiast wykorzystać DMA, zdecydowałem się na odbieranie wyników w przerwaniu. Mikrokontroler domyślnie działa z częstotliwością 8MHz, czyli w przybliżeniu 10x szybciej niż częstotliwość wyzwalania. Czyli przerwanie powinno następować co około 10 cykli zegara systemowego. Widzisz katastrofę? Obsługa przerwania na pewno trwa więcej niż 10 cykli. Ba! Same „wewnętrzne” działania (pamiętasz rozdział 5?) procesora, związane z wejściem w obsługę wyjątku, to było coś koło 12 cykli zegara (maksymalnie). ISR ma na oko z 30 rozkazów asemblera, z czego część wykonuje się dłużej niż 1 cykl zegarowy. Czyli kolejne przerwanie będzie, na bank, zgłasiane zanim zakończy się obsługa poprzedniego. Zdecydowanie to nie ma prawa działać :)

Rozwiązanie? Można rozburzyć system zegarowy mikrokontrolera do wyższej częstotliwości. Można zatrudnić do pomocy DMA. Ponieważ jednak to jest tylko „akademicki” przykład, zdecydowałem się na prostsze rozwiązanie: zmieniłem konfigurację TIM6 tak aby trygierzył nieco rzadziej ($ARR=99$) i „spowolniłem” sygnał testowy tak, aby zarejestrował się tylko kilka okresów tegoż.

Podsumujmy więc najważniejsze błędy z tego programu:

- pomycone instancje przetworników (ADC1 zamiast ADC2) w:
 - funkcji głównej - linia 46 listingu
 - procedurze obsługi przerwania - linia 25 listingu
- pomycona nazwa bitu w pętli *while* z 59 linii funkcji głównej (*ADC_CR_ADEN* zamiast *ADC_CR_JADSTART*)
- zbyt szybkie wyzwalanie konwersji (konfiguracja licznika TIM6)

Skoro program został poprawiony i jako tako działa, to pozostało nam odczytanie zebranych wyników. Poniżej znajduje się wykres sporządzony z próbek przebiegu testowego zarejestrowanego bezpośrednio i po wzmacnieniu (x4) przez wbudowany wzmacniacz. Surowe wartości próbek przeliczono na napięcie.



Rys. 20.5. Wykres zarejestrowanych próbek sygnału testowego zmierzonego bezpośrednio (czerwony) i po wzmacnieniu x4 (niebieski). Wartości przeliczono na napięcie przy założeniu $V_{ref} = 3,3V$.

Tabela 20.4 Parametry zmierzonych sygnałów (stosunek obliczono z wartości przed zaokrągleniem)

Sygnal zmierzony	wartość minimalna [V]	przesunięcie [V]	wartość maksymalna [V]
bezpośrednio	0,07	0,35	0,62
ze wzmacnieniem	0,24	1,36	2,46
stosunek wartości	4,10	4,00	3,99

Jak dla mnie - działa.

W ramach ciekawostki jeszcze powiem, że domyślne (fabryczne) wartości kalibracyjne mojego mikrokontrolera wynoszą:

- TRIMOFFSETN - 0x10
- TRIMOFFSETP - 0x10

Zaś wartości otrzymane w procesie „osobistej” kalibracji:

- TRIMOFFSETN - 0x10
- TRIMOFFSETP - 0x0F

Co warto zapamiętać z tego rozdziału:

- wzmacniacz może pracować w trzech podstawowych konfiguracjach:
 - niezależnie/samodzielnie
 - z programowo ustawianym wzmacnieniem (PGA)
 - wtórnik napięciowy
- wyjście wzmacniacza jest podłączone do trzeciego kanału przetwornika ADC2 oraz **zawsze** jest wyprowadzone na nóżkę PA6 (gdy wzmacniacz jest włączony)
- wzmacniacz posiada opcję kalibracji (jeśli z niej nie skorzystamy to zostaną wykorzystane domyślne wartości kalibracyjne)

DODATEK 1: FUNKCJA KONFIGURUJĄCA PORTY (F103)

Funkcja konfigurująca porty F103

(bardzo silnie wzorowana na przykładach ze strony <http://www.freddiechopin.info/>):

```
1. void gpio_pin_cfg(GPIO_TypeDef * const port, GpioPin_t pin, GpioMode_t mode){  
2.     pin = __builtin_ctz(pin)*4;  
3.  
4.     uint32_t volatile * cr_reg;  
5.     uint32_t cr_val;  
6.  
7.     cr_reg = &port->CRL;  
8.  
9.     if (pin > 28){  
10.         pin -= 32;  
11.         cr_reg = &port->CRH;  
12.     }  
13.  
14.  
15.     cr_val = *cr_reg;  
16.     cr_val &= ~((uint32_t)(0x0f << pin));  
17.     cr_val |= (uint32_t)(mode << pin);  
18.     *cr_reg = cr_val;  
19.  
20. }
```

Definicje nazw trybów konfiguracji:

```
1. typedef enum {  
2.     /* Push-Pull */  
3.     gpio_mode_output_PP_2MHz = 2,  
4.     gpio_mode_output_PP_10MHz = 1,  
5.     gpio_mode_output_PP_50MHz = 3,  
6.  
7.     /* Open-Drain */  
8.     gpio_mode_output_OD_2MHz = 6,  
9.     gpio_mode_output_OD_10MHz = 5,  
10.    gpio_mode_output_OD_50MHz = 7,  
11.  
12.    /* Push-Pull */  
13.    gpio_mode_alternate_PP_2MHz = 10,  
14.    gpio_mode_alternate_PP_10MHz = 9,  
15.    gpio_mode_alternate_PP_50MHz = 11,  
16.  
17.    /* Open-Drain */  
18.    gpio_mode_alternate_OD_2MHz = 14,  
19.    gpio_mode_alternate_OD_10MHz = 13,  
20.    gpio_mode_alternate_OD_50MHz = 15,  
21.  
22.    /* Analog input (ADC) */  
23.    gpio_mode_input_analog = 0,  
24.    /* Floating digital input. */  
25.    gpio_mode_input_floating = 4,  
26.    /* Digital input with pull-up/down (depending on the ODR reg.). */  
27.    gpio_mode_input_pull = 8  
28.  
29. } GpioMode_t;
```

Definicje nazw pinów (maski bitowe):

```

1. typedef enum {
2.     PA0 = 0x00000001,
3.     PA1 = 0x00000002,
4.     PA2 = 0x00000004,
5.     PA3 = 0x00000008,
6.     PA4 = 0x00000010,
7.     PA5 = 0x00000020,
8.     PA6 = 0x00000040,
9.     PA7 = 0x00000080,
10.    PA8 = 0x00000100,
11.    PA9 = 0x00000200,
12.    PA10 = 0x00000400,
13.    PA11 = 0x00000800,
14.    PA12 = 0x00001000,
15.    PA13 = 0x00002000,
16.    PA14 = 0x00004000,
17.    PA15 = 0x00008000,
18.
19.    PB0 = 0x00000001,
20.    ...
21.    PB15 = 0x00008000,
22.    ...
23. }
```

Opis działania funkcji: w linijce trzeciej maska pinu (argument funkcji, np. PA0, PC7, ...) zamieniana jest na nr bitu w rejestrze (np. PA7 to siódmy bit) i mnożona razy cztery. Po opis funkcji wbudowanej *ctz*³²⁰ odsyłam do Internetów. Poniższa tabelka pokazuje o co chodzi (dla skrócenia zapisu przyjąłem że port ma 8 pinów, z wyjątkiem ostatniego przypadku):

Tabela 1.1 Przeliczanie maski na nr bitu

pin	maska pinu	<code>__builtin_ctz(„maska”)*4</code>
P0	0b00000001	0
P1	0b00000010	4
P2	0b00000100	8
P3	0b00001000	12
P7	0b10000000	28
P8	0b1 00000000	32

Obliczona wartość jest wykorzystywana przy określaniu pozycji bitów związanych z danym pinem w rejestrze CRL/H. Na każdy pin przypadają 4 bity konfiguracyjne. Czyli dla pinu zerowego będą to bity **0**, 1, 2, 3; dla pinu pierwszego bity **4**, 5, 6, 7; dla pinu drugiego bity **8**, 9, 10 ,11... itd. Widać związek z tabelką? Już tłumaczę po co te cyrki, czemu nie podaję w argumencie funkcji od razu numeru pinu tylko się uparłem na maskę? Bo dzięki temu mogę wykorzystać moje definicje (PB0, PC6, ...) do operacji na rejestrach ODR i IDR. Przykładowo:

```
GPI0A->ODR |= PA3;
```

320 Count Trailing Zeros - policz ile ostatnich (najmniej znaczących) bitów jakiejś wartości to zera

Jedziemy dalej z funkcją. Wskaźnik *cr_reg* ustawiany jest na dolny rejestr konfiguracyjny CRL. Jeśli konfigurujemy pin wyższy niż 7 to musimy się przestawić na rejestr CRH. W takiej sytuacji (dla pinów od ósmego w góre) spełniony jest warunek z linii 10 (patrz tabela 1.1). Wskaźnik *cr_reg* przestawiany jest na rejestr CRH, zaś zmienna *pin* pomniejszana o 32. Odejmowanie jest potrzebne aby prawidłowo określić położenie bitów konfiguracyjnych pinów 8 - 15 w rejestrze CRH.

Teraz już z górką. W linii 15. wartość rejestru konfiguracyjnego przepisywana jest do zmiennej pomocniczej *cr_val*. Następnie, ze względu na pułapkę o której wspominałem (o tu Tertio!) zerowane są wszystkie bity związane z konfigurowanym pinem. Widać tu zastosowanie wyliczonej wartości zmiennej *pin*, która określa przesunięcie żądanych bitów w rejestrze. W linii 17. zostaje ustawiona nowa wartość bitów konfiguracyjnych odpowiadająca wybranemu trybowi. Ukoronowaniem działa jest zapisanie wyniku do rejestru. Najważniejszym mykiem tego programu jest to, że w nazwach trybów są od razu zakodowane wartości bitów konfiguracyjnych.

DODATEK 2: FUNKCJA KONFIGURUJĄCA PORTY (F429)

Funkcja konfigurującą porty F429

(bardzo silnie wzorowana na przykładach ze strony <http://www.freddiechopin.info/>):

```
1. void gpio_pin_cfg(GPIO_TypeDef * const __restrict__ port, GpioPin_t pin, GpioMode_t mode){  
2.  
3.     if (mode & 0x100u) port->OTYPER |= pin;  
4.     else port->OTYPER &= (~uint32_t)~pin;  
5.  
6.     pin = __builtin_ctz(pin)*2;  
7.  
8.     uint32_t reset_mask = ~(0x03u << pin);  
9.     uint32_t reg_val;  
10.  
11.    reg_val = port->MODER;  
12.    reg_val &= reset_mask;  
13.    reg_val |= (((mode & 0x600u) >> 9u) << pin );  
14.    port->MODER = reg_val;  
15.  
16.    reg_val = port->PUPDR;  
17.    reg_val &= reset_mask;  
18.    reg_val |= (((mode & 0x30u) >> 4u) << pin );  
19.    port->PUPDR = reg_val;  
20.  
21.    reg_val = port->OSPEEDR;  
22.    reg_val &= reset_mask;  
23.    reg_val |= (((mode & 0xC0u) >> 6u) << pin);  
24.    port->OSPEEDR = reg_val;  
25.  
26.  
27.    volatile uint32_t * reg_addr;  
28.    reg_addr = &port->AFR[0];  
29.  
30.    pin*=2;  
31.  
32.    if ( pin > 28){  
33.        pin -= 32;  
34.        reg_addr = &port->AFR[1];  
35.    }  
36.  
37.    reg_val = *reg_addr;  
38.    reg_val &= ~(0x0fu << pin);  
39.    reg_val |= (uint32_t)(mode & 0x0ful) << pin;  
40.    *reg_addr = reg_val;  
41. }
```

Uwaga! Tak jak wspomniano w rozdziale 3.4, od (chyba) 10 wersji dokumentacji (RM) nastąpiła zmiana w nazewnictwie „prędkości” wyjść. Z racji uwagi jednak na lenistwo własne moje, nie chce mi się zmieniać definicji wykorzystywanych przez omawianą tu funkcję. W związku z powyższym, pozostawiam „stare” określenia prędkości:

- LS - *Low Speed*
- MS - *Medium Speed*
- FS - *Fast/Full Speed* (w „nowym” RMie ta nazwa została zmieniona na: *High Speed*)
- HS - *High Speed* (w „nowym” RMie ta nazwa została zmieniona na: *Very High Speed*)

Definicje nazw trybów konfiguracji:

```

typedef enum {

/* Push-Pull; Low, Medium, Full, High Speed. */
    gpio_mode_output_PP_LS = 512,
    gpio_mode_output_PP_MS = 576,
    gpio_mode_output_PP_FS = 640,
    gpio_mode_output_PP_HS = 704

    /* Open-Drain */
    gpio_mode_output_OD_LS = 768,
    gpio_mode_output_OD_MS = 832,
    gpio_mode_output_OD_FS = 896,
    gpio_mode_output_OD_HS = 960,

    /* Open-Drain with weak Pull-Up */
    gpio_mode_output_OD_PU_LS = 784,
    gpio_mode_output_OD_PU_MS = 848,
    gpio_mode_output_OD_PU_FS = 912,
    gpio_mode_output_OD_PU_HS = 976,

/* Push-Pull in output state. No pullup in input
state. Alternate peripheral controls actual state. */
    gpio_mode_AF0_PP_LS = 1024,
    gpio_mode_AF0_PP_MS = 1088,
    gpio_mode_AF0_PP_FS = 1152,
    gpio_mode_AF0_PP_HS = 1216,

/* Push-Pull when output. Pull-Up when input. */
    gpio_mode_AF0_PP_PU_LS = 1040,
    gpio_mode_AF0_PP_PU_MS = 1104,
    gpio_mode_AF0_PP_PU_FS = 1168,
    gpio_mode_AF0_PP_PU_HS = 1232,

/* Push-Pull when output. Pull-Down when input. */
    gpio_mode_AF0_PP_PD_LS = 1056,
    gpio_mode_AF0_PP_PD_MS = 1120,
    gpio_mode_AF0_PP_PD_FS = 1184,
    gpio_mode_AF0_PP_PD_HS = 1248,

/* Open-Drain */
    gpio_mode_AF0_OD_LS = 1280,
    gpio_mode_AF0_OD_MS = 1344,
    gpio_mode_AF0_OD_FS = 1408,
    gpio_mode_AF0_OD_HS = 1472,

/* Open-Drain when output. Pull-Up when input. */
    gpio_mode_AF0_OD_PU_LS = 1296,
    gpio_mode_AF0_OD_PU_MS = 1360,
    gpio_mode_AF0_OD_PU_FS = 1424,
    gpio_mode_AF0_OD_PU_HS = 1488,

/* Open-Drain when output. Pull-Down when input. */
    gpio_mode_AF0_OD_PD_LS = 1312,
    gpio_mode_AF0_OD_PD_MS = 1376,
    gpio_mode_AF0_OD_PD_FS = 1440,
    gpio_mode_AF0_OD_PD_HS = 1504,

    gpio_mode_AF1_PP_LS = 1025,
    gpio_mode_AF1_PP_MS = 1089,
    gpio_mode_AF1_PP_FS = 1153,
    gpio_mode_AF1_PP_HS = 1217,
    gpio_mode_AF1_PP_PU_LS = 1041,
    gpio_mode_AF1_PP_PU_MS = 1105,
    gpio_mode_AF1_PP_PU_FS = 1169,
    gpio_mode_AF1_PP_PU_HS = 1233,
    gpio_mode_AF1_PP_PD_LS = 1057,
    gpio_mode_AF1_PP_PD_MS = 1121,
    gpio_mode_AF1_PP_PD_FS = 1185,
    gpio_mode_AF1_PP_PD_HS = 1249,

    gpio_mode_AF1_OD_LS = 1281,
    gpio_mode_AF1_OD_MS = 1345,
    gpio_mode_AF1_OD_FS = 1409,
    gpio_mode_AF1_OD_HS = 1473,
```

```

gpio_mode_AF7_OD_PU_LS = 1303,
gpio_mode_AF7_OD_PU_MS = 1367,
gpio_mode_AF7_OD_PU_FS = 1431,
gpio_mode_AF7_OD_PU_HS = 1495,
gpio_mode_AF7_OD_PD_LS = 1319,
gpio_mode_AF7_OD_PD_MS = 1383,
gpio_mode_AF7_OD_PD_FS = 1447,
gpio_mode_AF7_OD_PD_HS = 1511,

gpio_mode_AF8_PP_LS = 1032,
gpio_mode_AF8_PP_MS = 1096,
gpio_mode_AF8_PP_FS = 1160,
gpio_mode_AF8_PP_HS = 1224,
gpio_mode_AF8_PP_PU_LS = 1048,
gpio_mode_AF8_PP_PU_MS = 1112,
gpio_mode_AF8_PP_PU_FS = 1176,
gpio_mode_AF8_PP_PU_HS = 1240,
gpio_mode_AF8_PP_PD_LS = 1064,
gpio_mode_AF8_PP_PD_MS = 1128,
gpio_mode_AF8_PP_PD_FS = 1192,
gpio_mode_AF8_PP_PD_HS = 1256,

gpio_mode_AF8_OD_LS = 1288,
gpio_mode_AF8_OD_MS = 1352,
gpio_mode_AF8_OD_FS = 1416,
gpio_mode_AF8_OD_HS = 1480,
gpio_mode_AF8_OD_PU_LS = 1304,
gpio_mode_AF8_OD_PU_MS = 1368,
gpio_mode_AF8_OD_PU_FS = 1432,
gpio_mode_AF8_OD_PU_HS = 1496,
gpio_mode_AF8_OD_PD_LS = 1320,
gpio_mode_AF8_OD_PD_MS = 1384,
gpio_mode_AF8_OD_PD_FS = 1448,
gpio_mode_AF8_OD_PD_HS = 1512,

gpio_mode_AF9_PP_LS = 1033,
gpio_mode_AF9_PP_MS = 1097,
gpio_mode_AF9_PP_FS = 1161,
gpio_mode_AF9_PP_HS = 1225,
gpio_mode_AF9_PP_PU_LS = 1049,
gpio_mode_AF9_PP_PU_MS = 1113,
gpio_mode_AF9_PP_PU_FS = 1177,
gpio_mode_AF9_PP_PU_HS = 1241,
gpio_mode_AF9_PP_PD_LS = 1065,
gpio_mode_AF9_PP_PD_MS = 1129,
gpio_mode_AF9_PP_PD_FS = 1193,
gpio_mode_AF9_PP_PD_HS = 1257,

gpio_mode_AF9_OD_LS = 1289,
gpio_mode_AF9_OD_MS = 1353,
gpio_mode_AF9_OD_FS = 1417,
gpio_mode_AF9_OD_HS = 1481,
gpio_mode_AF9_OD_PU_LS = 1305,
gpio_mode_AF9_OD_PU_MS = 1369,
gpio_mode_AF9_OD_PU_FS = 1433,
gpio_mode_AF9_OD_PU_HS = 1497,
gpio_mode_AF9_OD_PD_LS = 1321,
gpio_mode_AF9_OD_PD_MS = 1385,
gpio_mode_AF9_OD_PD_FS = 1449,
gpio_mode_AF9_OD_PD_HS = 1513,

gpio_mode_AF10_PP_LS = 1034,
gpio_mode_AF10_PP_MS = 1098,
gpio_mode_AF10_PP_FS = 1162,
gpio_mode_AF10_PP_HS = 1226,
gpio_mode_AF10_PP_PU_LS = 1050,
gpio_mode_AF10_PP_PU_MS = 1114,
gpio_mode_AF10_PP_PU_FS = 1178,
gpio_mode_AF10_PP_PU_HS = 1242,
gpio_mode_AF10_PP_PD_LS = 1066,
gpio_mode_AF10_PP_PD_MS = 1130,
gpio_mode_AF10_PP_PD_FS = 1194,
gpio_mode_AF10_PP_PD_HS = 1258,
```

<code>gpio_mode_AF1_OD_LS</code> = 1297,	<code>gpio_mode_AF10_OD_LS</code> = 1290,
<code>gpio_mode_AF1_OD_MS</code> = 1361,	<code>gpio_mode_AF10_OD_MS</code> = 1354,
<code>gpio_mode_AF1_OD_PU_FS</code> = 1425,	<code>gpio_mode_AF10_OD_FS</code> = 1418,
<code>gpio_mode_AF1_OD_PU_HS</code> = 1489,	<code>gpio_mode_AF10_OD_HS</code> = 1482,
<code>gpio_mode_AF1_OD_PD_LS</code> = 1313,	<code>gpio_mode_AF10_OD_PU_LS</code> = 1306,
<code>gpio_mode_AF1_OD_PD_MS</code> = 1377,	<code>gpio_mode_AF10_OD_PU_MS</code> = 1370,
<code>gpio_mode_AF1_OD_PD_FS</code> = 1441,	<code>gpio_mode_AF10_OD_PU_FS</code> = 1434,
<code>gpio_mode_AF1_OD_PD_HS</code> = 1505,	<code>gpio_mode_AF10_OD_PU_HS</code> = 1498,
 	<code>gpio_mode_AF10_OD_PD_LS</code> = 1322,
<code>gpio_mode_AF2_PP_LS</code> = 1026,	<code>gpio_mode_AF10_OD_PD_MS</code> = 1386,
<code>gpio_mode_AF2_PP_MS</code> = 1090,	<code>gpio_mode_AF10_OD_PD_FS</code> = 1450,
<code>gpio_mode_AF2_PP_FS</code> = 1154,	<code>gpio_mode_AF10_OD_PD_HS</code> = 1514,
<code>gpio_mode_AF2_PP_HS</code> = 1218,	
<code>gpio_mode_AF2_PP_PU_LS</code> = 1042,	<code>gpio_mode_AF11_PP_LS</code> = 1035,
<code>gpio_mode_AF2_PP_PU_MS</code> = 1106,	<code>gpio_mode_AF11_PP_MS</code> = 1099,
<code>gpio_mode_AF2_PP_PU_FS</code> = 1170,	<code>gpio_mode_AF11_PP_FS</code> = 1163,
<code>gpio_mode_AF2_PP_PU_HS</code> = 1234,	<code>gpio_mode_AF11_PP_HS</code> = 1227,
<code>gpio_mode_AF2_PP_PD_LS</code> = 1058,	<code>gpio_mode_AF11_PP_PU_LS</code> = 1051,
<code>gpio_mode_AF2_PP_PD_MS</code> = 1122,	<code>gpio_mode_AF11_PP_PU_MS</code> = 1115,
<code>gpio_mode_AF2_PP_PD_FS</code> = 1186,	<code>gpio_mode_AF11_PP_PU_FS</code> = 1179,
<code>gpio_mode_AF2_PP_PD_HS</code> = 1250,	<code>gpio_mode_AF11_PP_PU_HS</code> = 1243,
 	<code>gpio_mode_AF11_PP_PD_LS</code> = 1067,
<code>gpio_mode_AF2_OD_LS</code> = 1282,	<code>gpio_mode_AF11_PP_PD_MS</code> = 1131,
<code>gpio_mode_AF2_OD_MS</code> = 1346,	<code>gpio_mode_AF11_PP_PD_FS</code> = 1195,
<code>gpio_mode_AF2_OD_FS</code> = 1410,	<code>gpio_mode_AF11_PP_PD_HS</code> = 1259,
<code>gpio_mode_AF2_OD_HS</code> = 1474,	
<code>gpio_mode_AF2_OD_PU_LS</code> = 1298,	<code>gpio_mode_AF11_OD_LS</code> = 1291,
<code>gpio_mode_AF2_OD_PU_MS</code> = 1362,	<code>gpio_mode_AF11_OD_MS</code> = 1355,
<code>gpio_mode_AF2_OD_PU_FS</code> = 1426,	<code>gpio_mode_AF11_OD_FS</code> = 1419,
<code>gpio_mode_AF2_OD_PU_HS</code> = 1490,	<code>gpio_mode_AF11_OD_HS</code> = 1483,
<code>gpio_mode_AF2_OD_PD_LS</code> = 1314,	<code>gpio_mode_AF11_OD_PU_LS</code> = 1307,
<code>gpio_mode_AF2_OD_PD_MS</code> = 1378,	<code>gpio_mode_AF11_OD_PU_MS</code> = 1371,
<code>gpio_mode_AF2_OD_PD_FS</code> = 1442,	<code>gpio_mode_AF11_OD_PU_FS</code> = 1435,
<code>gpio_mode_AF2_OD_PD_HS</code> = 1506,	<code>gpio_mode_AF11_OD_PU_HS</code> = 1499,
 	<code>gpio_mode_AF11_OD_PD_LS</code> = 1323,
<code>gpio_mode_AF3_PP_LS</code> = 1027,	<code>gpio_mode_AF11_OD_PD_MS</code> = 1387,
<code>gpio_mode_AF3_PP_MS</code> = 1091,	<code>gpio_mode_AF11_OD_PD_FS</code> = 1451,
<code>gpio_mode_AF3_PP_FS</code> = 1155,	<code>gpio_mode_AF11_OD_PD_HS</code> = 1515,
<code>gpio_mode_AF3_PP_HS</code> = 1219,	
<code>gpio_mode_AF3_PP_PU_LS</code> = 1043,	<code>gpio_mode_AF12_PP_LS</code> = 1036,
<code>gpio_mode_AF3_PP_PU_MS</code> = 1107,	<code>gpio_mode_AF12_PP_MS</code> = 1100,
<code>gpio_mode_AF3_PP_PU_FS</code> = 1171,	<code>gpio_mode_AF12_PP_FS</code> = 1164,
<code>gpio_mode_AF3_PP_PU_HS</code> = 1235,	<code>gpio_mode_AF12_PP_HS</code> = 1228,
<code>gpio_mode_AF3_PP_PD_LS</code> = 1059,	<code>gpio_mode_AF12_PP_PU_LS</code> = 1052,
<code>gpio_mode_AF3_PP_PD_MS</code> = 1123,	<code>gpio_mode_AF12_PP_PU_MS</code> = 1116,
<code>gpio_mode_AF3_PP_PD_FS</code> = 1187,	<code>gpio_mode_AF12_PP_PU_FS</code> = 1180,
<code>gpio_mode_AF3_PP_PD_HS</code> = 1251,	<code>gpio_mode_AF12_PP_PU_HS</code> = 1244,
 	<code>gpio_mode_AF12_PP_PD_LS</code> = 1068,
<code>gpio_mode_AF3_OD_LS</code> = 1283,	<code>gpio_mode_AF12_PP_PD_MS</code> = 1132,
<code>gpio_mode_AF3_OD_MS</code> = 1347,	<code>gpio_mode_AF12_PP_PD_FS</code> = 1196,
<code>gpio_mode_AF3_OD_FS</code> = 1411,	<code>gpio_mode_AF12_PP_PD_HS</code> = 1260,
<code>gpio_mode_AF3_OD_HS</code> = 1475,	
<code>gpio_mode_AF3_OD_PU_LS</code> = 1299,	<code>gpio_mode_AF12_OD_LS</code> = 1292,
<code>gpio_mode_AF3_OD_PU_MS</code> = 1363,	<code>gpio_mode_AF12_OD_MS</code> = 1356,
<code>gpio_mode_AF3_OD_PU_FS</code> = 1427,	<code>gpio_mode_AF12_OD_FS</code> = 1420,
<code>gpio_mode_AF3_OD_PU_HS</code> = 1491,	<code>gpio_mode_AF12_OD_HS</code> = 1484,
<code>gpio_mode_AF3_OD_PD_LS</code> = 1315,	<code>gpio_mode_AF12_OD_PU_LS</code> = 1308,
<code>gpio_mode_AF3_OD_PD_MS</code> = 1379,	<code>gpio_mode_AF12_OD_PU_MS</code> = 1372,
<code>gpio_mode_AF3_OD_PD_FS</code> = 1443,	<code>gpio_mode_AF12_OD_PU_FS</code> = 1436,
<code>gpio_mode_AF3_OD_PD_HS</code> = 1507,	<code>gpio_mode_AF12_OD_PU_HS</code> = 1500,
 	<code>gpio_mode_AF12_OD_PD_LS</code> = 1324,
<code>gpio_mode_AF4_PP_LS</code> = 1028,	<code>gpio_mode_AF12_OD_PD_MS</code> = 1388,
<code>gpio_mode_AF4_PP_MS</code> = 1092,	<code>gpio_mode_AF12_OD_PD_FS</code> = 1452,
<code>gpio_mode_AF4_PP_FS</code> = 1156,	<code>gpio_mode_AF12_OD_PD_HS</code> = 1516,
<code>gpio_mode_AF4_PP_HS</code> = 1220,	
<code>gpio_mode_AF4_PP_PU_LS</code> = 1044,	<code>gpio_mode_AF13_PP_LS</code> = 1037,
<code>gpio_mode_AF4_PP_PU_MS</code> = 1108,	<code>gpio_mode_AF13_PP_MS</code> = 1101,
<code>gpio_mode_AF4_PP_PU_FS</code> = 1172,	<code>gpio_mode_AF13_PP_FS</code> = 1165,
<code>gpio_mode_AF4_PP_PU_HS</code> = 1236,	<code>gpio_mode_AF13_PP_HS</code> = 1229,
<code>gpio_mode_AF4_PP_PD_LS</code> = 1060,	<code>gpio_mode_AF13_PP_PU_LS</code> = 1053,
<code>gpio_mode_AF4_PP_PD_MS</code> = 1124,	<code>gpio_mode_AF13_PP_PU_MS</code> = 1117,
<code>gpio_mode_AF4_PP_PD_FS</code> = 1188,	<code>gpio_mode_AF13_PP_PU_FS</code> = 1181,
<code>gpio_mode_AF4_PP_PD_HS</code> = 1252,	<code>gpio_mode_AF13_PP_PU_HS</code> = 1245,
 	<code>gpio_mode_AF13_PP_PD_LS</code> = 1069,
<code>gpio_mode_AF4_OD_LS</code> = 1284,	<code>gpio_mode_AF13_PP_PD_MS</code> = 1133,
<code>gpio_mode_AF4_OD_MS</code> = 1348,	<code>gpio_mode_AF13_PP_PD_FS</code> = 1197,

```

gpio_mode_AF4_OD_FS = 1412,
gpio_mode_AF4_OD_HS = 1476,
gpio_mode_AF4_OD_PU_LS = 1300,
gpio_mode_AF4_OD_PU_MS = 1364,
gpio_mode_AF4_OD_PU_FS = 1428,
gpio_mode_AF4_OD_PU_HS = 1492,
gpio_mode_AF4_OD_PD_LS = 1316,
gpio_mode_AF4_OD_PD_MS = 1380,
gpio_mode_AF4_OD_PD_FS = 1444,
gpio_mode_AF4_OD_PD_HS = 1508,

gpio_mode_AF5_PP_LS = 1029,
gpio_mode_AF5_PP_MS = 1093,
gpio_mode_AF5_PP_FS = 1157,
gpio_mode_AF5_PP_HS = 1221,
gpio_mode_AF5_PP_PU_LS = 1045,
gpio_mode_AF5_PP_PU_MS = 1109,
gpio_mode_AF5_PP_PU_FS = 1173,
gpio_mode_AF5_PP_PU_HS = 1237,
gpio_mode_AF5_PP_PD_LS = 1061,
gpio_mode_AF5_PP_PD_MS = 1125,
gpio_mode_AF5_PP_PD_FS = 1189,
gpio_mode_AF5_PP_PD_HS = 1253,

gpio_mode_AF5_OD_LS = 1285,
gpio_mode_AF5_OD_MS = 1349,
gpio_mode_AF5_OD_FS = 1413,
gpio_mode_AF5_OD_HS = 1477,
gpio_mode_AF5_OD_PU_LS = 1301,
gpio_mode_AF5_OD_PU_MS = 1365,
gpio_mode_AF5_OD_PU_FS = 1429,
gpio_mode_AF5_OD_PU_HS = 1493,
gpio_mode_AF5_OD_PD_LS = 1317,
gpio_mode_AF5_OD_PD_MS = 1381,
gpio_mode_AF5_OD_PD_FS = 1445,
gpio_mode_AF5_OD_PD_HS = 1509,

gpio_mode_AF6_PP_LS = 1030,
gpio_mode_AF6_PP_MS = 1094,
gpio_mode_AF6_PP_FS = 1158,
gpio_mode_AF6_PP_HS = 1222,
gpio_mode_AF6_PP_PU_LS = 1046,
gpio_mode_AF6_PP_PU_MS = 1110,
gpio_mode_AF6_PP_PU_FS = 1174,
gpio_mode_AF6_PP_PU_HS = 1238,
gpio_mode_AF6_PP_PD_LS = 1062,
gpio_mode_AF6_PP_PD_MS = 1126,
gpio_mode_AF6_PP_PD_FS = 1190,
gpio_mode_AF6_PP_PD_HS = 1254,

gpio_mode_AF6_OD_LS = 1286,
gpio_mode_AF6_OD_MS = 1350,
gpio_mode_AF6_OD_FS = 1414,
gpio_mode_AF6_OD_HS = 1478,
gpio_mode_AF6_OD_PU_LS = 1302,
gpio_mode_AF6_OD_PU_MS = 1366,
gpio_mode_AF6_OD_PU_FS = 1430,
gpio_mode_AF6_OD_PU_HS = 1494,
gpio_mode_AF6_OD_PD_LS = 1318,
gpio_mode_AF6_OD_PD_MS = 1382,
gpio_mode_AF6_OD_PD_FS = 1446,
gpio_mode_AF6_OD_PD_HS = 1510,

gpio_mode_AF7_PP_LS = 1031,
gpio_mode_AF7_PP_MS = 1095,
gpio_mode_AF7_PP_FS = 1159,
gpio_mode_AF7_PP_HS = 1223,
gpio_mode_AF7_PP_PU_LS = 1047,
gpio_mode_AF7_PP_PU_MS = 1111,
gpio_mode_AF7_PP_PU_FS = 1175,
gpio_mode_AF7_PP_PU_HS = 1239,
gpio_mode_AF7_PP_PD_LS = 1063,
gpio_mode_AF7_PP_PD_MS = 1127,
gpio_mode_AF7_PP_PD_FS = 1191,
gpio_mode_AF7_PP_PD_HS = 1255,

```

```

gpio_mode_AF13_PP_PD_HS = 1261,
gpio_mode_AF13_OD_LS = 1293,
gpio_mode_AF13_OD_MS = 1357,
gpio_mode_AF13_OD_FS = 1421,
gpio_mode_AF13_OD_HS = 1485,
gpio_mode_AF13_OD_PU_LS = 1309,
gpio_mode_AF13_OD_PU_MS = 1373,
gpio_mode_AF13_OD_PU_FS = 1437,
gpio_mode_AF13_OD_PU_HS = 1501,
gpio_mode_AF13_OD_PD_LS = 1325,
gpio_mode_AF13_OD_PD_MS = 1389,
gpio_mode_AF13_OD_PD_FS = 1453,
gpio_mode_AF13_OD_PD_HS = 1517,

gpio_mode_AF14_PP_LS = 1038,
gpio_mode_AF14_PP_MS = 1102,
gpio_mode_AF14_PP_FS = 1166,
gpio_mode_AF14_PP_HS = 1230,
gpio_mode_AF14_PP_PU_LS = 1054,
gpio_mode_AF14_PP_PU_MS = 1118,
gpio_mode_AF14_PP_PU_FS = 1182,
gpio_mode_AF14_PP_PU_HS = 1246,
gpio_mode_AF14_PP_PD_LS = 1070,
gpio_mode_AF14_PP_PD_MS = 1134,
gpio_mode_AF14_PP_PD_FS = 1198,
gpio_mode_AF14_PP_PD_HS = 1262,

gpio_mode_AF14_OD_LS = 1294,
gpio_mode_AF14_OD_MS = 1358,
gpio_mode_AF14_OD_FS = 1422,
gpio_mode_AF14_OD_HS = 1486,
gpio_mode_AF14_OD_PU_LS = 1310,
gpio_mode_AF14_OD_PU_MS = 1374,
gpio_mode_AF14_OD_PU_FS = 1438,
gpio_mode_AF14_OD_PU_HS = 1502,
gpio_mode_AF14_OD_PD_LS = 1326,
gpio_mode_AF14_OD_PD_MS = 1390,
gpio_mode_AF14_OD_PD_FS = 1454,
gpio_mode_AF14_OD_PD_HS = 1518,

gpio_mode_AF15_PP_LS = 1039,
gpio_mode_AF15_PP_MS = 1103,
gpio_mode_AF15_PP_FS = 1167,
gpio_mode_AF15_PP_HS = 1231,
gpio_mode_AF15_PP_PU_LS = 1055,
gpio_mode_AF15_PP_PU_MS = 1119,
gpio_mode_AF15_PP_PU_FS = 1183,
gpio_mode_AF15_PP_PU_HS = 1247,
gpio_mode_AF15_PP_PD_LS = 1071,
gpio_mode_AF15_PP_PD_MS = 1135,
gpio_mode_AF15_PP_PD_FS = 1199,
gpio_mode_AF15_PP_PD_HS = 1263,

```

```

gpio_mode_AF15_OD_LS = 1295,
gpio_mode_AF15_OD_MS = 1359,
gpio_mode_AF15_OD_FS = 1423,
gpio_mode_AF15_OD_HS = 1487,
gpio_mode_AF15_OD_PU_LS = 1311,
gpio_mode_AF15_OD_PU_MS = 1375,
gpio_mode_AF15_OD_PU_FS = 1439,
gpio_mode_AF15_OD_PU_HS = 1503,
gpio_mode_AF15_OD_PD_LS = 1327,
gpio_mode_AF15_OD_PD_MS = 1391,
gpio_mode_AF15_OD_PD_FS = 1455,
gpio_mode_AF15_OD_PD_HS = 1519,

```

```

/* Digital floating input. */
gpio_mode_in_floating = 0,

```

```

/* Digital input with Pull-Up */
gpio_mode_in_PU = 16,

```

```

/* Digital input with Pull-Down */
gpio_mode_in_PD = 32,

```

```

gpio_mode_AF7_OD_LS = 1287,
gpio_mode_AF7_OD_MS = 1351,
gpio_mode_AF7_OD_FS = 1415,
gpio_mode_AF7_OD_HS = 1479,
} /* Analog input/output */
} GpioMode_t;

```

Opis działania funkcji: zasada działania funkcji jest taka sama jak poprzednio. Cały myk polega na tym, że nazwy trybów zawierają wartości bitów konfiguracyjnych poszczególnych rejestrów. Wartość liczbową przyporządkowana nazwie każdego z trybów konfiguracji, zbudowana jest następująco:

- bity 0-3 to wartość pól AFRL (AFRH jeśli konfigurowany jest pin powyżej siódmego), lub zero jeśli to nie funkcja alternatywna
- bity 4-5 to wartość PUPDR
- bity 6-7 to wartość OSPEEDR
- bit 8 to wartość OT (rejestr GPIO_OTYPER)
- bity 9-10 to wartość MODER

Na przykład: aby ustawić pin jako wyjście typu open-drain z podciąganiem do góry i „prędkością” *fast* należy, zgodnie z tabelą 3.3 (rozdział 3.4) skonfigurować co następuje:

- MODER = 0b01
- OTYPER = 0b1
- OSPEEDR = 0b10
- PUPDR = 0b01

Zgodnie z opisem „budowy” wartości określającej tryb konfiguracji (poprzednie wykropkowanie), powstanie coś takiego:

Tabela 2.1 Budowa wartości liczbowej kodującej ustawienia trybu

bit	10	9	8	7	6	5	4	3	2	1	0
konfigurowane pole	MODER		OT		OSPEEDR		PUPDR		AFRL lub AFRH		
wartość	0b01		0b1	0b10		0b01		0b0000			

Czyli suma summarum, wartość tej liczby wyniesie: $0b01110010000 = 912$. I tak oto powstała wartość przyporządkowana nazwie *gpio_mode_output_OD_PU_FS*. Pozostałe wartości pokazane na omawianym listingu zostały utworzone w ten sam sposób³²¹.

321 bez żartów - oczywiście że nie liczyłem tego na piechotę, od tego jest komputer...

Tym sposobem jedna liczba zawiera w sobie wartości wszystkich rejestrów konfiguracyjnych pinu mikrokontrolera. W omawianej funkcji poszczególne wartości są wyłuskiwane (poprzez maskowanie i przesunięcia bitowe) i zapisywane w odpowiednich rejestrach.

DODATEK 3: MAKRO DO BIT BANDINGU

Upoznaję się z „narzędziami” do bit bandowania wyglądającymi paskudnie. Na szczęście należy do tej grupy narzędzi, które wystarczy raz napisać (i przetestować) i można więcej kodu nie oglądać. Oczywiście to tylko propozycja – zachęcam do poszukiwań lepszych rozwiązań :) Freddie proponuje, z tego co pamiętam, w swoich przykładach (i na forum) podejście polegające na korzystaniu z własnych plików nagłówkowych z definicjami bitów już w bb. Ja się wyłamuję i korzystam z takiego potworka³²²:

Straszak do bit bandingu (najprawdopodobniej bardzo silnie wzorowany na przykładach ze strony <http://www.freddiechopin.info/>):

```
1. enum { SRAM_BB_REGION_START = 0x20000000 };
2. enum { SRAM_BB_REGION_END = 0x200fffff };
3. enum { SRAM_BB_ALIAS = 0x22000000 };
4.
5. enum { PERIPH_BB_REGION_START = 0x40000000 };
6. enum { PERIPH_BB_REGION_END = 0x400fffff };
7. enum { PERIPH_BB_ALIAS = 0x42000000 };
8.
9. #define SRAM_ADR_COND(adres) ( (uint32_t)&adres >= SRAM_BB_REGION_START && (uint32_t)&adres <=
10. SRAM_BB_REGION_END )
11.
12. #define PERIPH_ADR_COND(adres) ( (uint32_t)&adres >= PERIPH_BB_REGION_START &&
13. (uint32_t)&adres <= PERIPH_BB_REGION_END )
14.
15. #define BB_SRAM2(adres, bit) ( SRAM_BB_ALIAS + ((uint32_t)&adres -
16. SRAM_BB_REGION_START)*32u + (uint32_t)(bit*4u) )
17.
18. #define BB_PERIPH(adres, bit) ( PERIPH_BB_ALIAS + ((uint32_t)&adres -
19. PERIPH_BB_REGION_START)*32u + (uint32_t)(__builtin_ctz(bit))*4u )
20.
21. /* bit - bit mask, not bit position! */
22. #define BB(adres, bit) (*__IO uint32_t *)(__builtin_bit_is_set((SRAM_ADR_COND(adres) ? BB_SRAM2(adres, bit) : \
23. (PERIPH_ADR_COND(adres) ? BB_PERIPH(adres, bit) : 0 )) \
24.
25. #define BB_SRAM(adres, bit) (*__IO uint32_t *)BB_SRAM2(adres, bit)
```

W pierwszej kolejności zdefiniowane są granice regionów i początki aliasów – za pomocą enumów... bo tak. Dalej są dwa makra sprawdzające czy przekazany im adres mieści się w bb regionie SRAMowym lub peripheralowym. Kolejne dwa makra obliczają adres w aliasie z wykorzystaniem formułki, którą wyprowadziliśmy w rozdziale 4.3. Jedno makro dla regionu w RAMie, drugie dla peryferiów. W makrze dla peryferiów dodatkowo wykorzystano funkcję wbudowaną *ctz* do obliczenia numeru bitu w słowie, gdyż parametrem wywołania makra będzie maska bitowa (np. PA1 - patrz przykłady niżej).

Makro, o wdzięcznej nazwie bb, ma najwięcej do roboty. W zależności od tego czy modyfikowany bit należy do regionu w RAMie czy peryferialnego, makro wstawia odpowiedni adres wyliczony przez jedno z poprzednich makr; następnie rzutuje to na wskaźnik i wyłuskuje wartość. Jeżeli podany bit nie leży w żadnym z regionów bb, to makro rozwinię się do postaci:

322 notabene dałbym sobie rękę uciąć, że też wzorowanego na jakimś przykładzie Freddiego

`*(uint32_t *)0`

co powinno szybko i bezboleśnie wykrzaczyć program i pomóc w lokalizacji błędu. O ostatnim makrze opowiem w kolejnym akapicie. Masakra prawda? Zachęcam do poszukiwań lepszego, wygodniejszego, bardziej wyrafinowanego sposobu korzystania z bb i podzielenia się nim ze *moi* :)

Poprawne działanie makra udowodniliśmy w rozdziale 4.4. Jak ktoś nie pamięta to proszę sobie przypomnieć jak to ładnie działało z bitami układów peryferyjnych. Ze zmiennym w RAMie sprawa jest nieco trudniejsza, gdyż kompilator nie zna adresu zmiennej w pamięci – to już broszka linkera. W związku z tym makra nie mogą zostać całkiem uproszczone na etapie komplikacji. Listing poniżej:

Modyfikacja zmiennej w pamięci SRAM za pomocą makra BB:

1.	BB(zmienna, 3) = 1;	
2.	8000184: 4b0c	ldr r3, [pc, #48] ; (80001b8 <main+0x34>)
3.	8000186: f103 4260	add.w r2, r3, #3758096384 ; 0xe0000000
4.	800018a: f5b2 1f80	cmp.w r2, #1048576 ; 0x100000
5.	800018e: d308	bcc.n 80001a2 <main+0x1e>
6.	8000190: f103 4240	add.w r2, r3, #3221225472 ; 0xc0000000
7.	8000194: f5b2 1f80	cmp.w r2, #1048576 ; 0x100000
8.	8000198: d208	bcs.n 80001ac <main+0x28>
9.	800019a: f103 7304	add.w r3, r3, #34603008 ; 0x2100000
10.	800019e: 015b	lsls r3, r3, #5
11.	80001a0: e005	b.n 80001ae <main+0x2a>
12.	80001a2: 015b	lsls r3, r3, #5
13.	80001a4: f103 5308	add.w r3, r3, #570425344 ; 0x22000000
14.	80001a8: 330c	adds r3, #12
15.	80001aa: e000	b.n 80001ae <main+0x2a>
16.	80001ac: 2300	movs r3, #0
17.	80001ae: 2201	movs r2, #1
18.	80001b0: 601a	str r2, [r3, #0]
19.	80001b8: 20000800	.word 0x20000800

Całość zaczyna się od wczytania do rejestru ogólnego r3 wartości spod adresu 0x800 01b8 (wynoszącej 0x2000 0800). Wartość wskazuje na jakiś adres w SRAMie. Za pomocą programu *nm* sprawdziłem adres pod jakim wylądowała moja *zmienna* – ta dam - to właśnie 0x2000 0800.

Drugi rozkaz to następująca operacja: $r2 = r3 + 0xe000 0000$. Wynik dodawania wynosi 0x1 0000 0800 co nie zmieści się w 32-bitowym rejestrze r2. Po obcięciu wartości zostanie samo 0x0000 0800. Patrząc na tą wartość „pod kątem bit bandingu” wygląda to jak obliczona różnica adresu zmiennej i adresu początku regionu – ale to tylko moje spekulacje.

Operacja trzecia to porównanie wartości r2 i stałej 0x0010 0000. Na bank jest to jedno z porównań w makrze od bb, sprawdzające do którego regionu należy argument. Dziwne wartości wynikają z tego, że kompilator coś sobie uprościł/przeliczył/skrócił – rozkminimy za chwilę. Rozkaz *bcc* spowoduje skok pod podany adres, jeśli w poprzedzającym porównaniu (*cmp*) pierwszy argument był mniejszy od drugiego. W naszym przypadku skok nastąpi jeśli $r2 < 0x0010 0000$. W przeciwnym wypadku skoku nie będzie i program poleci dalej do kolejnej instrukcji dodawania: $r2 = r3 + 0xc000 0000 = 0xE000 0800$. Znowu następuje porównanie i warunkowy skok

jeśli $r2 \geq 0x0010\ 0000$. Dalsze analizowanie asemblera pozostawiam Czytelnikowi bo opis i tak będzie zbyt zakrecony żeby się połapać.

Do dalszej analizy proponuję „pseudo-kod” stworzony na podstawie omawianego listingu (zdecydowanie zachęcam do własnej analizy i porównania wyników):

Pseudo kod na podstawie komplatu przykłady z BB i zmienną w pamięci SRAM:

```
1.     r3 = &zmienna
2.     r2 = r3 + 0xE000 0000 = 0x0000 0800
3.     if ( r2 < 0x0010 0000 ) goto L1
4.     r2 = r3 + 0xc000 0000 = 0xe000 0800
5.     if ( r2 >= 0x00100 0000 ) goto L2
6.     r3 = r3 + 0x0210 0000
7.     r3 = r3*32
8.     goto L3
9. L1:   r3 = r3*32
10.    r3 = r3 + 0x2200 0000
11.    r3 = r3 + 12
12.    goto L3
13. L2:   r3 = 0
14. L3:   r2 = 1
15.     *[r3] = r2
```

Zadanie jest ułatwione, bo wiemy do czego dążymy (makro BB). Opis słowny byłby pokręcony, więc na początek kilka uproszczeń matematycznych i podstawień. Pozbywamy się r2:

Usunięty rejestr pomocniczy r2:

```
1.     r3 = &zmienna
2.     if ( r3 + 0xE000 0000 < 0x0010 0000 ) goto L1
3.     if ( r3 + 0xc000 0000 >= 0x00100 0000 ) goto L2
4.     r3 = r3 + 0x0210 0000
5.     r3 = r3*32
6.     goto L3
7. L1:   r3 = r3*32
8.     r3 = r3 + 0x2200 0000
9.     r3 = r3 + 12
10.    goto L3
11. L2:   r3 = 0
12. L3:   *[r3] = 1
```

Przekształcamy operacje w warunkach i dalej upraszczamy:

Dalsze przekształcenia i uproszczenia pseudo kodu:

```
1.     r3 = &zmienna
2.     if ( &zmienna < 0x0010 0000 - 0xE000 0000 ) goto L1
3.     if ( &zmienna >= 0x00100 0000 - 0xc000 0000 ) goto L2
4.     r3 = &zmienna + 0x0210 0000
5.     r3 = r3*32
6.     goto L3
7. L1:   r3 = &zmienna * 32
8.     r3 = r3 + 0x2200 0000
9.     r3 = r3 + 12
10.    goto L3
11. L2:   r3 = 0
12. L3:   *[r3] = 1
```

I jeszcze trochę uprośćmy:

```
1.      if ( &zmienna < 0x200F FFFF ) goto L1
2.      if ( &zmienna >= 0x400F FFFF ) goto L2
3.      r3 = (&zmienna + 0x0210 0000)*32
4.      goto L3
5. L1:   r3 = &zmienna * 32 + 0x2200 0000 + 12
6.      goto L3
7. L2:   r3 = 0
8. L3:   *[r3] = 1
```

No i teraz już jest całkiem prosto. Jeśli spełniony będzie pierwszy warunek: zmienna leży poniżej górnej granicy regionu SRAM to lecimy do L1, obliczamy adres słowa aliasu zgodnie z naszą formułką bb i skaczemy do L3 – czyli do zapisu stałej 1 do obliczonego słowa. Policzymy czy adres się zgadza (uwaga na zaokrąglenia do 32bitów):

- z listingu: $0x2000\ 0800 * 32 + 0x2200\ 0000 + 12 = 0x2201\ 000c$
- z formułki: $0x2200\ 0000 + (0x2000\ 0800 - 0x2000\ 0000) * 32 + 3 * 4 = 0x2201\ 000c$

I o to chodziło :) W kwestii uzupełniania – drugi warunek to wyjechanie poza zakres *Peripheral*. Następuje wtedy skok do L2 i zabezpieczenie w postaci rozwinięcia makra do adresu „0”. W zakresie adresów peryferialnych żaden z warunków nie jest spełniony.

Swoją drogą... te dwa warunki nie wyczerpują wszystkich możliwości. A co jeśli będzie za SRAMem a przed *Peripheral*? Powinny być cztery warunki tak na logikę... sam nie wiem. Coś kompilator sobie to uprościł. Niech mu będzie – nie wnikam. Grunt, że działa.

Jak widać, w przypadku zmiennych w SRAMie, kod nie został „policzony i uproszczony” w trakcie komplikacji. Procesor musi się nieco więcej naprawić. Możemy mu jednak nieco pomóc za cenę naszej własnej wygody. Mianowicie, jeśli przy korzystaniu z bit bandingu w SRAMie zrezygnujemy z naszego uniwersalnego makra „BB” i jawnie wywołamy makro SRAMowe (BB_SRAM) to odpadnie problem porównywania adresów. Kod znaczaco się uprości. Po to właśnie powstało to ostatnie makro BB_SRAM:

Użycie makra BB_SRAM:

```
1.      BB_SRAM(zmienna, 3) = 1;
2. 8000184:    4b02        ldr      r3, [pc, #8] ; (8000190 <main+0xc>)
3. 8000186:    015b        lsls     r3, r3, #5
4. 8000188:    2201        movs     r2, #1
5. 800018a:    60da        str      r2, [r3, #12]
```

Analizę pozostawiam czytelnikowi.

DODATEK 4: TO BIT BAND OR NOT TO BIT BAND

Na Elektrodzie można znaleźć kilka tasiemcowatych dyskusji na temat tego czy zapis do rejonu aliasu bb trwa tyleż samo co do *normalnego* rejestrów peryferyjnych. Chodzi o to, czy te sprzętowe mechanizmy łączące *alias* i *region* wpływają jakoś na prędkość wykonywania operacji. Jako że jednoznacznej odpowiedzi nie znalazłem, sprawdziłem sam. Powstał prosty program testowy:

Program testowy:

```
1. uint32_t zmienna_odczyt_z_bb(void){
2.     uint32_t czas;
3.     system_cycnt_reset();
4.     __asm__ volatile (
5.         "push {r3}          \t\n"
6.         "push {r2}          \t\n"
7.         "mov r3, #0x22000000 \t\n"
8.
9.         "ldr r2, [r3, #0]    \t\n"
10.        /* 100x str lub ldr */
11.
12.
13.        "pop {r2}          \t\n"
14.        "pop {r3}          \t\n"
15.    );
16.
17.    czas = system_cycnt_get();
18.    return czas;
19. }
20.
21. static inline void system_cycnt_init(void){
22.     CoreDebug->DEMCR |= CoreDebug_DEMCR_TRCENA_Msk;
23.     DWT->CYCCNT = 0;
24. }
25.
26. static inline void __attribute__((always_inline)) system_cycnt_start(void) {
27.     DWT->CTRL |= DWT_CTRL_CYCCNTENA_Msk;
28. }
29.
30. inline void __attribute__((always_inline))system_cycnt_reset(void){
31.     DWT->CYCCNT = 0;
32. }
33.
34. inline uint32_t __attribute__((always_inline))system_cycnt_get(void){
35.     return DWT->CYCCNT;
36. }
37.
38. static inline void __attribute__((always_inline))system_cycnt_stop(void){
39.     DWT->CTRL &= ~DWT_CTRL_CYCCNTENA_Msk;
40. }
```

Idea działania jest taka, że mierzymy ile czasu zajmie wykonanie 100 instrukcji zapisujących oraz odczytujących rejestr z rejonu *aliasu* i „normalnej” pamięci. Na listingu nie pokazano tych 100 instrukcji, gdyż zająłby kilka stron :) W pełnej wersji programu testowego, linijka 9 powtarza się 100x, tak jak sugeruje komentarz z linii 10.

Pomiar „czasu” wykonany jest w oparciu o licznik cykli zegarowych (CYCCNT) jakiegoś tam bloku rdzenia (DWT). Takie rozwiązanie znalazłem kiedyś na forum ST³²³. Funkcja testowa działa następująco:

- zeruje licznik cykli zegara rdzenia
- zapisuje na stosie wartości modyfikowanych rejestrów (r2 i r3)
- w rejestrze r3 zapisuje testowany adres
- wykonuje 100 operacji zapisu (*str*) lub odczytu (*ldr*) spod testowego adresu
- przywraca wartości modyfikowanym rejestrom
- odczytuje czas z rejestru CYCCNT

Proste prawda? No to pora na wyniki:

Tabela 4.1 Czasy odczytów i zapisów z i bez bb

region	operacja	bitband	adres	Cortex M3	Cortex M4
SRAM	odczyt	nie	0x20002000	113	115
		tak	0x22000000	113	115
	zapis	nie	0x20002000	123	127
		tak	0x22000000	312	315
peryf	odczyt	nie	0x40000000	315	316
		tak	0x42000000	315	316
	zapis	nie	0x40000000	234	234
		tak	0x42000000	613	615

Testy powtarzałem kilka razy i wyniki miały 100% powtarzalność. Co można zauważyć?

- różnice między CM3 a CM4 są kosmetyczne
- odczyt pamięci z użyciem bb i bez bb trwa tyleż samo niezależnie od obszaru pamięci (pamięć SRAM lub rejesty peryferialni)
- odczytywanie pamięci układów peryferyjnych trwa blisko 3x dłużej niż pamięci SRAM (mikrokontroler podczas testów pracował z domyślną konfiguracją systemu zegarowego)
- zapisywanie pamięci układów peryferyjnych trwa mniej więcej 2x dłużej niż pamięci SRAM
- operacje zapisu do aliasu bb trwają około 3x dłużej niż bezpośrednio do pamięci układów peryferyjnych lub pamięci SRAM

323 patrz temat: *Duration of FLOAT operations* i post użytkownika *clive1*

Mój ulubiony kawałek każdego sprawozdania - wnioski :) Co z tego wynika w praktyce? Absolutnie nic. Sztuka dla sztuki. Program testowy użyty podczas eksperymentu jest nad wyraz osobliwy, specjalnie na potrzeby eksperymentu. Nikt nie wykonuje 100 operacji zapisu/odczytu tego samego adresu pamięci pod rząd. Żaden zdrowy kompilator czegoś takiego nie wygeneruje.

Ponadto, operacji zapisu i odczytu aliasu bb nie da się zastąpić, jeden do jednego, zapisem lub odczytem zwykłej pamięci. BB załatwia za nas operacje bitowe, sprzętowo. Bez tego mechanizmu musielibyśmy zastosować sekwencję RMW, czyli minimum trzy osobne rozkazy. Eksperiment testuje jedynie czas zapisu/odczytu do/z aliasu i regionu bb, a nie czas wykonywania (użytecznych w programie) operacji z użyciem tego mechanizmu. W związku z tym porównywanie uzyskanych wyników nie ma w praktyce sensu i do niczego przydatnego nie prowadzi :)

Otrzymane wyniki można potraktować jako ciekawostkę do rozmyślań, ale zdecydowanie nie jako „wadę” bb czy dowód na to że korzystanie z bb spowalnia realny i użyteczny program (a nie takie laboratoryjno akademickie wydumki jak ten mój). Miłych rozmyślań!

DODATEK 5: ATRYBUT INTERRUPT (F103, GCC)

Generalnie sprawa wygląda tak, że Cortex w swej genialności³²⁴ pozwala, aby procedura obsługi przerwania była zwykłą funkcją bez specjalnych atrybutów i udziwnień. Gdzieś już o tym pisałem... Ale! Pojawia się ciekawy problem z wyrównaniem stosu. Standard AAPCS (cokolwiek to jest) wymaga aby przy wchodzeniu do funkcji (jakiekolwiek), stos był zawsze wyrównany do 8-miu bajtów. Bo tak! Cały czas dba o to kompilator. Problem pojawia się przy przerwaniach. Mogą one wystąpić asynchronicznie, w każdej chwili, i nigdy nie wiadomo jak będzie wyglądał stos w chwili wystąpienia przerwania. W związku z tym istnieje ryzyko, że procedura obsługi przerwania zastanie stos nie wyrównany zgodnie ze standardem. Uprzedzając pytania: nie mam pojęcia czym to grozi. Coś mi chodzi po głowie, że komuś funkcje typu *printf* nie działały jeśli stos nie był prawidłowo wyrównany (problemy ze zmienną liczbą argumentów?).

Tak czy siak rozwiążaniem (obejściem?) tego problemu jest dodanie do ISR atrybutu *interrupt* (kompilator GCC). Jego celem jest wskazanie kompilatorowi że ta funkcja to ISR i stos w chwili jej wywołania jest nie-wiadomo-jaki, więc kompilator ma go profilaktycznie wyrównać. I dotąd wszystko się zgadza. Problem jest jednak taki, że od którejś tam rewizji (wersji) mikrokontrolera (chyba r1) wprowadzono bit STKALIGN. Pozwala on włączyć sprzętowe wyrównywanie stosu (przez rdzeń) przy wchodzeniu do ISR (domyślnie wyłączone). Po jego włączeniu rdzeń sam sobie wyrównuje stos i dodatkowe wyrównywanie przez kompilator nie jest potrzebne. Jest nadmiarowe. W niczym nie przeszkadza, ale po co tracić czas... i to jeszcze przy wchodzeniu w przerwanie. Co więcej od rewizji chyba r2, ta opcja jest włączona domyślnie. Pojawia się więc rozterka: co z atrybutem *interrupt*? Na 100% dodanie go nie będzie błędem i wszystko będzie działać. Ale czy warto wydłużać niepotrzebnie kod ISR, skoro procesor może zrobić to samo sprzętowo?

Rdzenie oznaczone są przez numer „rewizji” i „pod rewizji” np. r1p2. To jaki mamy rdzeń można odczytać z obudowy scalaczka (jak dekodować info poczytaj np. w erracie STM) lub debuggerem z procesora: rejestr pod adresem 0xe000 ed00. Przypominam (ale proszę sobie doszukać, bo nie jestem pewien na mur beton):

- od rewizji r1 wyrównywanie sprzętowe jest dostępne, ale domyślnie wyłączone
- od rewizji r2 wyrównywanie sprzętowe jest domyślnie włączone

Decyzję o tym czy stosować atrybut, czy też nie, pozostawiam Czytelnikowi. W ramach dmuchania na zimne, szczególnie na początku edukacji STMowej, proponuję go zostawić. Procesor w mojej

324 hasła dla zainteresowanych: sprzętowy *stacking* i wartość EXC_RETURN

płytkę HY-mini (cały czas mówimy tylko o F103) to r1p1, czyli mam bit STKALIGN, ale domyślnie wyłączony. Lektura tematyczna dla zainteresowanych:

- <http://www.elektroda.pl/rtvforum/topic2408935-30.html>
(wątek: *STM32 - ZL29ARM - Uruchamianie płytka bez bibliotek*)
- <https://gcc.gnu.org/onlinedocs/gcc/ARM-Function-Attributes.html#ARM-Function-Attributes>
(opis atrybutów funkcji kompilatora GCC)
- <http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.dui0203j/BABBHJDG.html>
(sposób obsługi wyjątków w środowisku RealView)
- https://gcc.gnu.org/bugzilla/show_bug.cgi?id=55757
(zgłoszenie błędu GCC: *Suboptimal interrupt prologue/epilogue for ARMv7-M (Cortex-M3)*)

Jeszcze ciekawy cytat (dwa cytaty) z *ARMv7-M Architecture Reference Manual* na koniec:

- „*Support of a 4-byte aligned stack (CCR.STKALIGN == '0') in ARMv7-M is deprecated.*”
- „*A side-effect when STKALIGN is enabled is that the amount of stack used on exception entry becomes a function of the alignment of the stack at the time that the exception is entered. As a result, the average and worst case stack usage will increase. The worst case increase is 4 bytes per exception entry.*”

DODATEK 6: PRZERWANIE WIDMO

O NVICu pisałem gdzieś jako o wrotach przez które poszczególne przerwania mogą się dostać do rdzenia. Jeśli zerkniemy do dokumentacji rdzeni (przygotowanej przez ARM, nie ST!) Cortex-M3 i Cortex-M4 to odnajdziemy informację, że rdzenie te mogą obsługiwać do 240 przerwań zewnętrznych. Tak obrazowo i infantylnie rzecz ujmując: z rdzenia wychodzi 240 kabelków, które potem producent mikrokontrolera podpina do swoich układów peryferyjnych wedle uznania. Rdzeniu (rdzeniowi?) wisi i powiewa co jest na końcu takiego „kabelka”. Zadaniem rdzenia jest jedynie przenieść się w odpowiednie miejsce w kodzie, gdy pojawi się sygnał na danej linii przerwania (w uproszczeniu). A czy sygnał pochodzi z licznika czy ADC to już rybka.

Jeśli teraz popatrzymy do dokumentacji przygotowanej przez ST to zobaczymy, że w tablicy wektorów przerwań zewnętrznych³²⁵ jest mniej pozycji! Przykładowo:

- STM32F103, connectivity line: 68 przerwań
- STM32F103, XL-density line: 60 przerwań
- STM32F405/407/415/417: 82 przerwania
- STM32F42x/43x: 91 przerwań

Czyli ST, z takich czy innych względów, nie wykorzystało wszystkich 240 kabelków wystających z rdzenia. Część z nich pozostała niepodłączona, dynda... I właśnie one są interesujące :)

Rdzeń nie ma pojęcia co wisi na końcu linii przerwania. W szczególności nie ma pojęcia czy w ogóle coś do niej jest podłączone! I to właśnie wykorzystamy w tym dowcipnym dodatku. To, że do linii nic nie jest podłączone oznacza, że żaden układ peryferyjny nie uruchomi tego przerwania. Ale przecież przerwanie można odpalić programowo w kontrolerze NVIC... Czujesz do czego zmierzam? Pytanie zagadka: czy można programowo wywołać takie „nieistniejące przerwanie widmo”? No... gdyby się nie dało to bym się nie produkował :) Poniżej przykład dla F429 (wybrałem ten mikrokontroler bo płytka jest akurat mniej zakurzona, z F103 działa to identycznie):

³²⁵ 325 zewnętrznych z punktu widzenia rdzenia (IRQ), nie mylić z przerwaniami zewnętrznymi mikrokontrolera!

Przerwanie widmo (F429):

```
1. #define led1_bb BB(GPIOG->ODR, PG13)
2. #define led2_bb BB(GPIOG->ODR, PG14)
3. #define PHANTOM_IRQn 91
4.
5. volatile uint32_t delay;
6.
7. int main(void){
8.
9.     RCC->AHB1ENR |= RCC_AHB1ENR_GPIOGEN;
10.
11.    gpio_pin_cfg(GPIOG, PG13, gpio_mode_out_PP_LS);
12.    gpio_pin_cfg(GPIOG, PG14, gpio_mode_out_PP_LS);
13.    SysTick_Config(16000000/4);
14.
15.    NVIC_EnableIRQ(PHANTOM_IRQn);
16.
17.    while(1){
18.        delay = 4;
19.        while(delay);
20.        NVIC_SetPendingIRQ(PHANTOM_IRQn);
21.    }
22.
23.
24. } /* main */
25.
26. void SysTick_Handler(void){
27.     if(delay) --delay;
28.     led1_bb ^= 1;
29. }
30.
31. void Phantom_IRQHandler(void){
32.     led2_bb ^= 1;
33. }
```

W programie wykorzystywane są dwie diody świecące (na PG13 i PG14). Linie 11 i 12 to konfiguracja tych pinów. Zegar włączam w linii 9, suma bitowa bo AHB1ENR ma domyślnie niezerową wartość. W linijkach 1 i 2 są dwie definicje mające na celu skrócenie zapisu przy dostępie do pinów za pomocą bb (tak dla wygody i urozmaicenia).

13) włączenie SysTicka, przerwanie co 250ms. W przerwaniu SysTicka dekrementowana jest zmienna globalna *delay* i machana jest jedna dioda.

15) włączenie przerwania „widmo”. Oczywiście w pliku nagłówkowym mikrokontrolera nie ma definicji dla niewykorzystywanych przerwań. W związku z tym stworzyłem ją sobie sam - linijka 3. Skąd wartość 91? STM32F429 wykorzystuje przerwania od numeru 0 do 90. Przerwanie 91 to pierwszy „niepodłączony nigdzie kabelek”.

17 - 21) w pętli nieskończonej jest proste opóźnienie oparte o SysTick. Co cztery przerwania SysTicka (ca 1s) wywoływana jest funkcja, która zmienia stan przerwania widmo na *pending*.

31 - 33) procedura obsługi przerwania widmo - machanie diodą (nazwę ISR dodałem wcześniej na ostatniej pozycji tablicy wektorów).

W efekcie działania programu obie diody ładnie migają (z różnymi częstotliwościami). Do czego to wykorzystać w praktyce? Do niczego. To tylko taka ciekawostka :)

DODATEK 7: GWAŁT NA NUCLEO W DWÓCH AKTACH

Skądinąd śliczna płytka NUCLEO-F334R8 podobnie jak (prawdopodobnie) inne płytki z tej serii, posiada dwie, nad wyraz, irytujące cechy. Irytujące do tego stopnia, że ku ich zgubie powstał ten dodatek.

Pierwsza sprawa to trój-podzielny programator ST-Link w wersji *full-wypas* V2-1. Urządzenie to przedstawia się w systemie jako trzy różne *device'y* - to się jakoś mądrze nazywa *USB composite device* chyba. Mniejsza. System operacyjny, po zadokowaniu wspomnianego ST-Linka w dziurce USB, wykrywa:

- programator ST-Link - jakby nie wykrywał programatora to byłaby nieco lipa :]
- wirtualny port szeregowy - ten ficzer jest fajny i przydatny
- urządzenie pamięci masowej - to jest idiotyzm...

Nie wiem co podkusiło ST, aby dorobić do ST-Linka pamięć masową. Jasne - są tam jakieś materiały propagandowe i można programować mikrokontroler „kopując” wsad na tego „udawanego pendrive'a”... tylko czy ktokolwiek z tego korzysta? Serio? Zresztą nie ważne, każdy programuje jak i czym chce.

Cały kłopot z tym ficzerem polega na tym, że system operacyjny po wykryciu nowego „dysku” automatycznie go montuje i otwiera. Innymi słowy za każdym razem gdy podłączalem Nucleo do komputera, wyskakiwało mi radosne okienko eksploratora plików. I było to na dłuższą metę dosyć drażniące. W związku z powyższym powstał plan pacyfikacji ST-Linka. Niespecjalnie interesuję się budową i działaniem systemu operacyjnego Linux (jakaś pochodna Ubuntu), więc gotowe rozwiążanie sobie wygooglałem. Sprawdziłem, działa, opisuję - może komuś się przyda: w katalogu */etc/modprobe.d/* dodałem nowy plik *stlink-storage.conf* o treści:

```
options usb-storage quirks=0483:374b:i
```

Dwa numerki w środku to PID i VID ST-Linka. Potem tylko reset i święty spokój! Zrzut z logu systemowego:

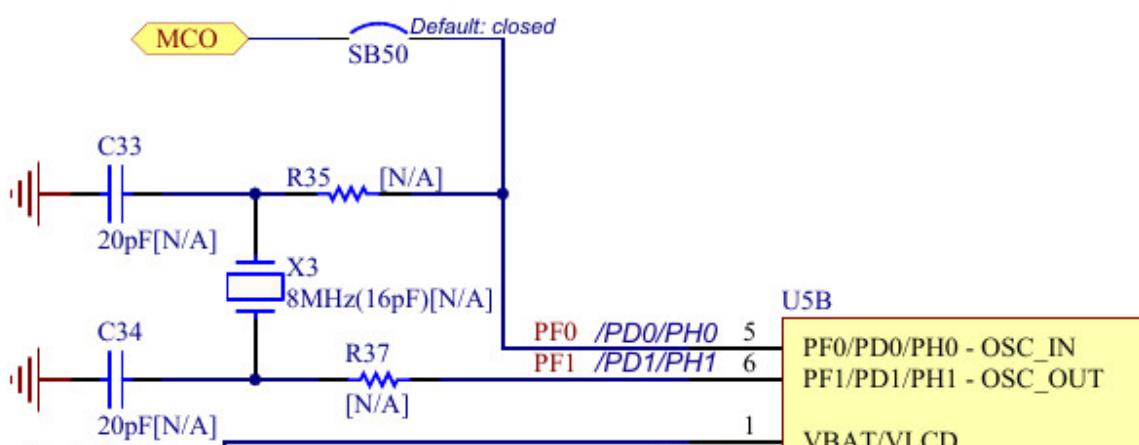
```
[146873.716073] usb 4-1: new full-speed USB device number 19 using uhci_hcd
[146873.893101] usb 4-1: New USB device found, idVendor=0483, idProduct=374b
[146873.893108] usb 4-1: New USB device strings: Mfr=1, Product=2, SerialNumber=3
[146873.893112] usb 4-1: Product: STM32 STLink
[146873.893116] usb 4-1: Manufacturer: STMicroelectronics
[146873.893120] usb 4-1: SerialNumber: 066DFF495056805087151149
```

```
[146873.954192] usb-storage 4-1:1.1: USB Mass Storage device detected
[146873.954377] usb-storage 4-1:1.1: device ignored
[146873.956384] cdc_acm 4-1:1.2: This device cannot do calls on its own. It is not a modem.
[146873.956413] cdc_acm 4-1:1.2: ttyACM0: USB ACM device
```

To był problem software'owy, teraz dla równowagi ubrudzimy się hardwarem³²⁶. Płytki Nucleo ma tylko jedną diodę świecącą do wykorzystania w programie. Wiem, że to jest budżetowy zestaw, ale jedna dioda to imo za mało. Z jednym przyciskiem da się żyć - goldpiny można sobie zwierać pierwszym lepszym przewodzikiem jaki wala się w szufladzie. Ale jedna dioda to już jest grubszy nietakt.

Plan jest taki: dołożyć drugą diodę do płytki Nucleo. Dodatkowo realizacja ma być „elegancka i finezyjna” jak przystało na gentlemana. Tzn. ma wyglądać tak, żeby nie było widać na pierwszy rzut oka (z daleka), że coś było kombinowane - żadne połączenia przewodami czy drucikami po powierzchni płytki nie wchodzą w rachubę! Absolutnie! Sprawę nieco utrudnia biała soldermaska przez którą g... widać i nie można sprawdzić jak ścieżki idą. Po krótkiej zadumie nad schematem i płytą znalazłem rozwiązanie :]

Na płytce znajduje się sporo nieużywanych pól lutowniczych, m.in. jakieś kondensatory które w tej wersji Nucleo nie występują, miejsca na zewnętrzne rezonatory, rezystory bez oporowe (0Ω) i mostki cynowe (SB - *solder bridge*) służące do konfiguracji zestawu. I właśnie je wykorzystamy. Domyślnie, do mikrokontrolera na płytce Nucleo, doprowadzony jest zewnętrzny sygnał zegarowy pochodzący z wyjścia MCO (*Master Clock Output*) mikrokontrolera wykorzystanego w ST-Linku. Zewnętrzny rezonator (kwarc) nie jest zainstalowany. Popatrzmy na kawałek schematu związany z tym rezonatorem.



Rys. 7.1. Fragment schematu zestawu Nucleo (źródło: www.st.com)

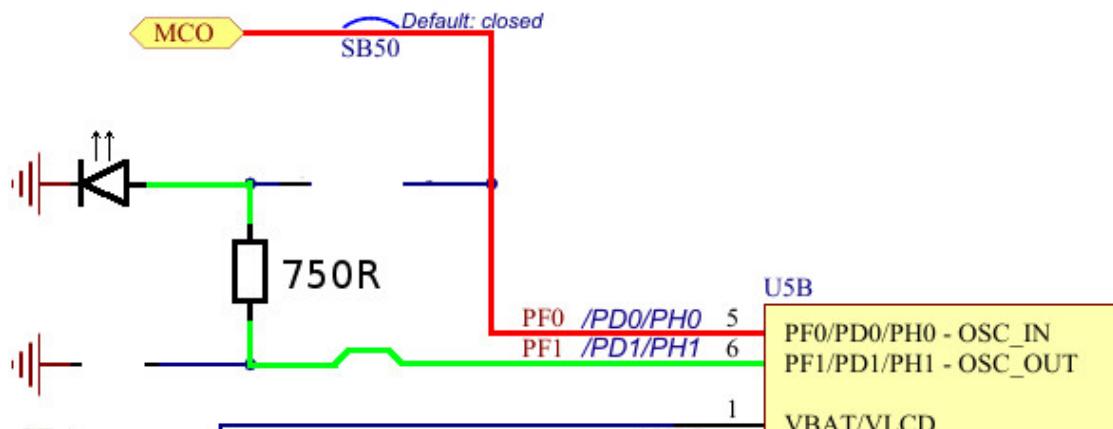
326 Ilu programistów potrzeba do zmiany żarówki? Żadnego, to problem sprzętowy!

W domyślnej konfiguracji sygnał zegarowy z ST-Linka doprowadzony jest do mikrokontrolera poprzez mostek SB50. Jeżeli użytkownik ma ochotę na podłączenie zewnętrznego кварca to musi wykonać kilka zmian:

- rozlutować SB50
- wlutować rezystory R35, R37
- wlutować kwarc X3
- wlutować kondensatory C33, C34

Założenie, że nie będziemy korzystać z zewnętrznego кварca, otwiera nam całkiem spore możliwości w kwestii dodania diody.

Na płytce przewidziano całkiem sporo miejsca na кварc w przewlekanej obudowie. Pierwszy mój pomysł był taki aby wlutować diodę w miejsce X3. Odrzuciłem go jednak, gdyż otwory pod кварc mają rozstaw około 10 mm. Niby dałoby się wcisnąć zwykłą diodę z rastrem 2,54 mm, ale ładnie by to nie wyglądało. A z kolei „duża” dioda (np. 10mm) wyglądałaby dziwnie. Opcja z diodą w miejscu кварcu odpadła. Drugi pomysł był taki, aby wlutować diodę i niezbędny rezistor w miejsce C34 i R37. Oba pola przeznaczone są na elementy SMD w rozmiarze 0603. Niestety w swoich przydasiach nie znalazłem takich maleństw, więc nie obyło się bez wizyty w „osiedlowym” elektroniku. Diody udało się kupić, rezystorów 0603 już nie... Suma summarum powstała trzecia wersja (patrz rysunek 7.2) z diodą SMD i rezystorem THT.



Rys. 7.2. Fragment zmodyfikowane schematu zestawu Nucleo

Rezystor został wlutowany w miejsce kwarcu od spodu płytki - żeby się nie rzucił w oczy. Dioda zajęła miejsce kondensatora C33, pady rezystora R37 zostały zwarte. Ta dam :] I teraz Nucleo ma dwie diody do użycia w programie. Cel osiągnięty. I świecili długą i szczęśliwie ;)

21. ERRATA I CHANGELOG („HOMINIS EST ERRARE, INSIPIENTIS IN ERRORE PERSEVERARE”³²⁷)

21.1. Błędy zauważone w wersji 1.0 Poradnika

Tabela 21.1. Błędy zauważone w wersji 1.0 Poradnika (z 06.11.2015r.)

lp.	Dotyczy	Opis
1.	rozdział 1.2, drugi akapit	Jest: Dotychczas bawiłem się tylko mikrokontrolerami AVR i moim największym A lepiej by było, gdyby było: Dotychczas bawiłem się tylko mikrokontrolerami ATmega oraz ATTiny i moim największym
2.	rozdział 1.3, pierwszy akapit	Jest: najlepiej AVR, bo będę je czasem traktował jako A lepiej by było, gdyby było (dodany przypis dolny): najlepiej AVR ^x , bo będę je czasem traktował jako ^x wszelkie nawiązania do mikrokontrolerów Atmel AVR odnoszą się do rodzin ATTiny i ATmega (chyba, że wyraźnie napisano inaczej)
3.	rozdział 2.7, drugi akapit	Jest: W AVR stosowana była taka konwencja: A lepiej by było, gdyby było: W AVR stosowana była taka konwencja (przypominam, że określenie „AVR” w Poradniku, odnosi się do popularnych układów ATmega i ATTiny):
4.	rozdział 3.4, rozwiązania zadania 3.5, ostatni akapit na stronie 46	Jest: Kłopot polega na tym że w pliku nagłówkowym mikrokontrolera STM32F429, zamiast A lepiej by było, gdyby było (dodany przypis dolny): Kłopot polega na tym że w pliku nagłówkowym mikrokontrolera STM32F429 ^x , zamiast ^x dotyczy tylko pliku nagłówkowego wyłuskanego z paczki z biblioteką SPL, nagłówka z paczki „Cube FW” ten problem nie dotyczy
5.	rozdział 5, tytuł rozdziału	Jest: 5. Przerwania i wyjątki („Macte animo, iuvenis!”) A lepiej by było, gdyby było: 5. Wyjątki („Macte animo, iuvenis!”)
6.	rozdział 5	Jest: trochę teorii o tym jak to jak to Cortex ogarnia przerwania A lepiej by było, gdyby było: trochę teorii o tym jak to jak to Cortex ogarnia wyjątki

327 „Ludzką rzeczą jest błędzić, głupców trwać w błędzie.”

		<p>Jest: tryb obsługi wyjątków (wyjątek to pojęcie zblżone do przerwania, tylko szersze).</p> <p>A lepiej by było, gdyby było: tryb obsługi wyjątków. Pod pojęciem wyjątku mieści się wszystko, co zaburza naturalny bieg programu. Np. znane z AVR przerwanie, które powoduje że procesor „przystaje” wykonywać aktualny program i przenosi się w zupełnie inne miejsce kodu (do procedury obsługi przerwania). Wyjątek to pojęcie ogólne. Przerwanie jest jednym z trzech rodzajów wyjątków (pozostałe dwa rodzaje to błędy i pułapki). W ramach pocieszenia powiem, że do momentu publikacji pierwszej wersji tego Poradnika, nie wiedziałem jaka jest w sumie różnica między przerwaniami a pozostałymi rodzajami wyjątków. Ba! Nie wiedziałem, że istnieje coś takiego jak pułapka... i ani trochę nie przeszkaǳało mi to w pisaniu prostych programów :)</p>
7.	rozdział 5.1, drugi akapit	<p>Jest:</p> <ul style="list-style-type: none">• handler mode - obsługa wyjątków (przerwań)• thread mode - wszystko co nie jest przerwaniem (np. funkcja main) <p>A lepiej by było, gdyby było:</p> <ul style="list-style-type: none">• handler mode - obsługa wyjątków (np. przerwań)• thread mode - wszystko co nie jest wyjątkiem (np. funkcja main)
8.	rozdział 5.1, wypunktowanie na stronie 63	<p>Jest:</p> <p>Co warto zapamiętać z tego rozdziału?</p> <p>A lepiej by było, gdyby było (dodany nowy kropek):</p> <p>Co warto zapamiętać z tego rozdziału?</p> <ul style="list-style-type: none">• wyjątek to ogólne pojęcie określające „coś”, co powoduje zmianę w przepływie programu (skok do procedury obsługi wyjątku), przerwanie jest jednym z rodzajów wyjątków
10.	rozdział 5.2, tytuł rozdziału	<p>Jest:</p> <p>5.2. Poznajmy wyjątki i przerwania w Cortexie</p> <p>A lepiej by było, gdyby było:</p> <p>5.2. Poznajmy wyjątki w Cortexie</p>
11.	rozdział 5.2, pierwszy akapit	<p>Jest:</p> <p>W Programming Manualu jest informacja, że przerwanie to wyjątek pochodzący od peryferiala lub wywołany (z premedytacją) programowo. Z tego co ja rozumiem to:</p> <ul style="list-style-type: none">• wyjątek to pojęcie ogólne, odnoszące się do wszystkiego co przerywa wykonywanie programu przez mikrokontroler i „przenosi rdzeń” w inne miejsce kodu• przerwanie to taki wyjątek pochodzący od peryferiala mikrokontrolera (a nie np. od któregoś z układów rdzenia) <p>I powyższego się będę trzymał... ale szczerze przyznaję, że nie czuję do końca jaka jest różnica między wyjątkiem a przerwaniem. I zupełnie dobrze mi się z tym żyje póki co.</p> <p>A lepiej by było, gdyby było:</p> <p>Pojęcie wyjątku obejmuje wszystko, co zaburza naturalny bieg programu. Przykładowo może to być przerwanie, które powoduje że procesor przerywa to co aktualnie robi i skacze w inne miejsce kodu (do procedury obsługi przerwania - ISR). Są trzy rodzaje wyjątków:</p> <ul style="list-style-type: none">• przerwanie• błąd• pułapka <p>Z naszego punktu widzenia, najciekawsze są przerwania. Przerwania mogą pochodzić od:</p> <ul style="list-style-type: none">• układów peryferyjnych mikrokontrolera jak w AVR (np. od interfejsów komunikacyjnych, przetworników ADC, nóżek mikrokontrolera, ...)• układów rdzenia (np. licznik SysTick)• mogą być wywoływane z premedytacją programowo (przerwanie PendSV)

		<p>Przerwania różnią się od pozostałych rodzajów wyjątków (błędów i pułapek) w dwóch zasadniczych kwestiach:</p> <ul style="list-style-type: none"> • nie muszą być obsłużone od razu - przerwanie może być np. wyłączone (jak w AVR -cli) lub mieć zbyt niski priorytet (patrz rozdział 5.4), w efekcie nie jest obsługiwane od razu po zgłoszeniu tylko przechodzi w stan oczekiwania • zgłoszenie przerwania nie wynika w sposób bezpośredni z tego co się dzieje aktualnie w programie (przerwania są asynchroniczne) - np. zgłoszenie przerwania zegarowego od licznika nie ma najczęściej nic wspólnego z rozkazami, które aktualnie wykonuje procesor <p>Pozostałe dwa rodzaje wyjątków są synchroniczne, czyli wynikają bezpośrednio z kodu programu (działania procesora). Np. próba wykonania dzielenia przez zero może wywołać wyjątek - błąd - który będzie powiązany z konkretnym rozkazem procesora (dzieleniem). Ponadto błędy i pułapki powinny zostać obsłużone natychmiast. Jeżeli jest to niemożliwe (np. błąd jest wyłączone lub ma zbyt niski priorytet) to następuje zjawisko „eskalacji” błędu i wywoływanym jest wyjątek HardFault. Błędami i pułapkami nie będziemy się zajmować szczegółowo, wyszlibyśmy zbyt daleko poza bezpieczny „playground” ogrodzony znakami „poczatkujący” :) Domyślnie wszystkie błędy (poza HardFault) są wyłączone, a przed wywołaniem sporej części z nich, skutecznie, chroni nas kompilator.</p>
12.	rozdział 5.2, tabela 5.1, opis przerwania NMI	<p>Jest: jeśli system kontroli sygnału zegarowego (Clock Security System) wykryje awarię zewnętrznego rezonatora.</p> <p>A lepiej by było, gdyby było: jeśli system kontroli sygnału zegarowego (Clock Security System) wykryje awarię zewnętrznego rezonatora (patrz zadanie 17.2).</p>
13.	rozdział 5.2, tabela 5.1, opis wyjątku Reset HardFault (patrz errata: rozdział 1.4, tabela 4, punkt 1)	<p>Jest: Wyjątek występujący przy błędzie w obsłudze innego wyjątku lub jeśli inny wyjątek związany z błędem (patrz niżej) ma ustawiony zbyt niski priorytet aby zostać obsłużonym.</p> <p>A lepiej by było, gdyby było: Błąd w obsłudze innego wyjątku lub jeśli inny błąd/pułapka nie może zostać obsłużony po zgłoszeniu (np. ma zbyt niski priorytet).</p>
14.	rozdział 5.2, akapit pod tabelą 5.1	<p>Jest: Jak widać w większości są to wyjątki związane z błędami (Faults). Nie ma co panikować - większości z nich nie musimy się bać, bo piszemy w języku wysokiego poziomu (C) i pilnuje nas kompilator.</p> <p>A lepiej by było, gdyby było: Jak widać spora część z nich to błędy (te z Fault w nazwie). Nie ma co panikować - większości z nich nie musimy się bać, bo piszemy w języku wysokiego poziomu (C) i pilnuje nas kompilator. Ponadto przypominam, że domyślnie wszystkie błędy poza HardFault są wyłączone (patrz rejestr SCB_SHCSR).</p>
15.	rozdział 5.2, akapit pod tabelą 5.1	<p>Jest: W 99% przypadków będziemy lądowali w Faultcie ze względu na odpalenie przerwania dla którego nie napisaliśmy procedury, ewentualnie przesadzimy ze zmiennymi i wysypie się stos.</p> <p>A lepiej by było, gdyby było: W 99% przypadków będziemy lądowali w Faultcie lub jakimś domyślnym handlerze wyjątku, ze względu na odpalenie przerwania dla którego nie napisaliśmy procedury obsługi, ewentualnie przesadzimy ze zmiennymi i wysypie się stos.</p>

16.	rozdział 5.2, drugi akapit pod tabelą 5.1	Jest: <i>To co nas będzie najbardziej interesowało to przerwania zewnętrzne (zewnętrzne z punktu widzenia rdzenia), czyli ostatni wiersz tabeli.</i> A lepiej by było, gdyby było: <i>To co nas będzie najbardziej interesowało to przerwania zewnętrzne (zewnętrzne z punktu widzenia rdzenia), czyli ostatni wiersz tabeli oraz przerwanie SysTick.</i>
17.	rozdział 5.2, ostatni akapit	Jest: <i>Zupełną nowością są priorytety wyjątków i przerwań (druga kolumna tabeli).</i> A lepiej by było, gdyby było: <i>Zupełną nowością są priorytety wyjątków (druga kolumna tabeli).</i>
18.	rozdział 5.3, przypis 82	Jest: <i>przypominam, że przerwania też są wyjątkami</i> A lepiej by było, gdyby było: <i>przypominam, że przerwania też są wyjątkami - jeśli dla kogoś jest już za dużo nowości, to proponuję aby tymczasowo myślał o każdym wyjątku jak o przerwaniu (takim jak znane z AVR)</i>
19.	rozdział 5.3, wypunktowanie	Jest: <i>active - wyjątek jest aktualnie obsługiwany przez procesor (wykonuje się ISR)</i> A lepiej by było, gdyby było: <i>active - wyjątek jest aktualnie obsługiwany przez procesor (wykonuje się procedura obsługi wyjątku, np. ISR przerwania)</i>
20.	rozdział 5.3, pierwszy akapit na stronie 68	Jest: <i>Gdy tylko stacking się zakończy, rozpoczyna się wykonywanie instrukcji z ISR.</i> A lepiej by było, gdyby było: <i>Gdy tylko stacking się zakończy, rozpoczyna się wykonywanie instrukcji z handlera wyjątku, np. ISR.</i>
21.	rozdział 5.3, pierwszy akapit na stronie 68	Jest: <i>Opóźnienie od pojawienia się tego co powoduje wyjątek do rozpoczęcia wykonywania ISR</i> A lepiej by było, gdyby było: <i>Opóźnienie od pojawienia się „tego co powoduje wyjątek” do rozpoczęcia wykonywania procedury jego obsługi</i>
22.	rozdział 5.3, drugi akapit na stronie 68	Jest: <i>Jeżeli w trakcie obsługi wyjątku zostanie on jeszcze raz wywołany [...] to będzie miał jednocześnie stan active i pending. Gdy zakończy się obsługa wyjątku (tego w stanie active) to dalej będzie utrzymyany stan oczekiwania (pending) i procesor ponownie skoczy do tej samej ISR. Dzięki temu nie nastąpi „zgubienie” drugiego przerwania z tego samego źródła.</i> A lepiej by było, gdyby było: <i>Jeżeli w trakcie obsługi przerwania zostanie ono jeszcze raz wywołane [...] to będzie miało jednocześnie stan active i pending. Gdy zakończy się obsługa przerwania (tego w stanie active) to dalej będzie utrzymywany stan oczekiwania (pending) i procesor ponownie skoczy do tej samej ISR. Dzięki temu nie nastąpi „zgubienie” drugiego przerwania z tego samego źródła.</i> <i>W przypadku pozostałych rodzajów wyjątków (błędów i pułapek) ponowne zgłoszenie tego samego wyjątku w trakcie jego obsługi spowoduje eskalację do wyjątku HardFault (patrz rozdział 5.2).</i>

23.	rozdział 5.3, trzeci akapit na stronie 68	Jest: Po zakończeniu ISR następuje powrót do „trybu użytkownika” (thread mode) - funkcji głównej jak ktoś woli. A lepiej by było, gdyby było: Po zakończeniu obsługi wyjątków następuje powrót do „trybu użytkownika” (thread mode) - funkcji głównej jak ktoś woli.
24.	rozdział 5.3, ostatnia kropka na stronie 68	Jest: - preemption - wywłaszczenie - to takie przerwanie przerwaniem - zostanie omówione w następnym rozdziale A lepiej by było, gdyby było: - preemption - wywłaszczenie - to przerwanie obsługi wyjątku przez inny wyjątek, zostanie omówione w następnym rozdziale
25.	rozdział 5.4, pierwszy akapit	Jest: Każdy wyjątek i przerwanie ma jakiś priorytet. A lepiej by było, gdyby było: Każdy wyjątek (np. przerwanie) ma jakiś priorytet.
26.	rozdział 5.4, dygresja między liniami	Jest: Reset ⁸⁴ ma najwyższy priorytet (-3). A lepiej by było, gdyby było (przesunięty przypis): Reset ma najwyższy priorytet (-3) ⁸⁴ .
27.	rozdział 5.4, pierwszy akapit na stronie 70 (patrz errata do wersji 1.1: tabela 2, pkt. 8.)	Jest: Tzn. jeśli w czasie trwania obsługi przerwania pojawi się nowe o wyższym priorytecie grupowym to nastąpi wywłaszczenie. Wywłaszczenie jest to przerwanie przerwania. Czyli procesor przerywa wykonywanie procedury przerwania o niższym priorytecie i skacze do tej od wyższego priorytetu. Przerwania w STM nie są blokowane po wejściu do ISR tak jak w AVRach. A lepiej by było, gdyby było: Tzn. jeśli w czasie trwania obsługi wyjątku (np. przerwania) pojawi się nowy o wyższym priorytecie grupowym to nastąpi wywłaszczenie. Wywłaszczenie jest to przerwanie wyjątku przez inny wyjątek (np. przerwanie przerwania przerwaniem). Czyli procesor przerywa wykonywanie procedury obsługi wyjątku o niższym priorytecie i skacze do tej od wyższego priorytetu. Przerwania (i ogólnie wyjątki) w STM nie są blokowane po wejściu do procedury obsługi tak jak w AVRach.
28.	rozdział 5.4, drugi akapit na stronie 70	Jest: Pod-priorytet decyduje tylko o kolejności wykonania przerwań posiadających ten sam priorytet grupowy. Jeżeli kilka przerwań o tym samym grupowym priorytecie oczekuje (pendinguje) to pod-priorytet decyduje o tym, w jakiej kolejności się wykonają. A lepiej by było, gdyby było: Pod-priorytet decyduje tylko o kolejności obsłużenia wyjątków posiadających ten sam priorytet grupowy. Jeżeli kilka wyjątków o tym samym grupowym priorytecie oczekuje (pendinguje) to pod-priorytet decyduje o tym, w jakiej kolejności się wykonają.
29.	rozdział 5.4, trzeci akapit na stronie 70	Jest: Najpierw zostają wykonane te przerwania, które mają A lepiej by było, gdyby było: Najpierw zostają wykonane te wyjątki, które mają
30.	rozdział 5.4, pierwszy akapit na stronie 71	Jest: czy podział na priorytet grupowy i pod-priorytet dotyczy tylko przerwań od peryferiów mikrokontrolera czy też przerwań od peryferiów rdzenia A lepiej by było, gdyby było: czy podział na priorytet grupowy i pod-priorytet dotyczy tylko przerwań od peryferiów mikrokontrolera czy też wyjątków pochodzących od rdzenia

31.	rozdział 5.4, pierwszy akapit na stronie 71	Jest: <i>podział priorytetów dotyczy wszystkich przerwań i wyjątków, bez różnicy czy pochodzą od peryferiów rdzenia czy mikrokontrolera.</i> A lepiej by było, gdyby było: <i>podział priorytetów dotyczy wszystkich wyjątków o konfigurowalnym priorytecie, bez różnicy czy pochodzą od peryferiów mikrokontrolera czy od rdzenia.</i>
32.	rozdział 5.4, drugi akapit na stronie 71	Jest: <i>Możliwe jest mianowicie zablokowanie przerwań do pewnego priorytetu.</i> A lepiej by było, gdyby było: <i>Możliwe jest mianowicie zablokowanie wyjątków do pewnego priorytetu.</i>
33.	rozdział 5.4, ostatni akapit	Jest: <i>- nie ma wcale -</i> A lepiej by było, gdyby było (dodany nowy akapit): <i>Przypominam, że jednym z cech błędów i pułapek jest to, że powinny zostać obsłużone od razu po zgłoszeniu. Należy to mieć na uwadze przy konfigurowaniu priorytetów wyjątków, jeśli zamierzamy korzystać z błędów lub pułapek. Jeżeli priorytet błędu/pułapki nie pozwoli na natychmiastowe obsłużenie (będzie równy lub niższy od aktualnego priorytetu procesora), to nastąpi eskalacja błędu i wygenerowanie błędu HardFault (o stałym priorytecie równym -1).</i>
34.	rozdział 5.4, wykropkowanie na końcu rozdziału (patrz errata do wersji 1.1: tabela 2, pkt. 9.)	Jest: <ul style="list-style-type: none">• w prostych programach nie ma potrzeby zmieniania priorytetów przerwań• po wejściu do ISR przerwania są blokowane jak w AVR A lepiej by było, gdyby było: <ul style="list-style-type: none">• w prostych programach nie ma potrzeby zmieniania priorytetów wyjątków• po wejściu do handlera wyjątku (np. ISR) wyjątki (w szczególności przerwania) nie są blokowane jak w AVR
35.	rozdział 5.5, pierwszy akapit	Jest: <i>Do konfigurowania przerwań ARM przygotowała</i> A lepiej by było, gdyby było: <i>Do konfigurowania wyjątków ARM przygotowała</i>
36.	rozdział 5.5, trzecia kropka na stronie 72	Jest: <i>funkcja przypisująca zakodowany priorytet konkretnemu przerwaniu</i> A lepiej by było, gdyby było: <i>funkcja przypisująca zakodowany priorytet konkretnemu wyjątkowi... wyjątkowi</i>
37.	rozdział 17.2, obserwacje z zadania 17.2, ostatnia kropka wypunktowania	Jest: <i>procesor wpada do default handlera</i> A lepiej by było, gdyby było: <i>procesor wpada do „domyślnej” procedury obsługi wyjątku (default handler)</i>

21.2. Błędy zauważone w wersji 1.1 Poradnika

Tabela 21.2. Błędy zauważone w wersji 1.1 Poradnika (wersja przejściowa, nie publikowana)

lp.	Dotyczy	Opis
1.	rozdział 2.6, drugi akapit (za wykropko-waniem)	<p>Jest: <i>Przykładem takiej biblioteki dostarczonej przez ST jest Standard Peripheral Library (SPL).</i></p> <p>A lepiej by było, gdyby było (dodano przypis dolny): <i>Przykładem takiej biblioteki dostarczonej przez ST jest Standard Peripheral Library (SPL)^x.</i></p> <p>^x biblioteka SPL nie jest już rozwijana przez ST, zastąpił ją STM32Cube</p>
2.	rozdział 2.6, drugi akapit na stronie 26	<p>Jest: <i>Dokładniej trzeba pobrać paczkę z biblioteką SPL lub czymś podobnym i wyłuskać z niej odpowiedni plik nagłówkowy.</i></p> <p>A lepiej by było, gdyby było: <i>Dokładniej trzeba pobrać paczkę z narzędziem STM32Cube i wyłuskać z niej odpowiedni plik nagłówkowy.</i></p>
3.	rozdział 2.6, wykropkowanie na stronie 26	<p>Jest:</p> <ul style="list-style-type: none"> • www.st.com • <i>Products → Microcontrollers → STM32 32-bit ARM Cortex MCUS → STM32F1 Series → STM32F103</i> • <i>Software → STM32 Standard Peripheral Libraries</i> • <i>STM32F10X Standard Peripheral Library</i> • <i>Get Software → Download</i> • <i>w rozpakowanym pliku: Libraries → CMSIS → CM3 → DeviceSupport → ST → STM32F10x</i> • <i>i mamy nasz garniec złota na krańcu tęczy – plik nagłówkowy mikrokontrolera z definicjami rejestrów i bitów: stm32f10x.h</i> <p>A lepiej by było, gdyby było:</p> <ul style="list-style-type: none"> • www.st.com • <i>Products → Microcontrollers → STM32 32-bit ARM Cortex MCUS → STM32F1 Series → STM32F103</i> • <i>Software → STM32Cube</i> • <i>STM32CubeF1</i> • <i>Get Software → Download</i> • <i>w rozpakowanym archiwum: Drivers → CMSIS → Device → ST → STM32F1xx → Include</i> • <i>w pliku stm32f1xx.h sprawdzamy nazwę pliku nagłówkowego dla naszego mikrokontrolera (np. mikrokontroler STM32F103VC to plik stm32f103xe.h)</i> • <i>i mamy nasz garniec złota na krańcu tęczy – plik nagłówkowy mikrokontrolera z definicjami rejestrów i bitów: stm32f103xe.h</i>

		Jest: Resztę można wywalić, albo zachować sobie na czarną godzinę. W chwilach całkowitej rezygnacji, jeśli coś nam nie będzie chciało działać, można podglądać źródła funkcji bibliotecznych. Gdyby za jakiś czas paczki z SPL zniknęły ze strony ST to pliki nagłówkowe można też znaleźć w paczkach biblioteki HAL czy CubeMX, tylko trzeba trochę pogrzebać (Cube/Drivers/CMSIS/Device/ST/...) i wybrać plik o nazwie najbardziej zbliżonej do posiadanej mikrokontrolera
4.	rozdział 2.6, akapit pod wykropkowaniem na stronie 26	A lepiej by było, gdyby było: Resztę można wywalić.
5.	rozdział 3.4, przykładowe rozwiązywanie zadania 3.5	Jest: <pre>if (GPIOA->IDR & GPIO_IDR_IDR_0){ GPIOG->ODR = GPIO_ODR_ODR_13; GPIOG->BSRRH = GPIO_BSRR_BS_14; } else { GPIOG->ODR &= ~GPIO_ODR_ODR_13; GPIOG->BSRRL = GPIO_BSRR_BS_14; }</pre> A lepiej by było, gdyby było: <pre>if (GPIOA->IDR & GPIO_IDR_IDR_0){ GPIOG->ODR = GPIO_ODR_ODR_13; GPIOG->BSRR = GPIO_BSRR_BR_14; } else { GPIOG->ODR &= ~GPIO_ODR_ODR_13; GPIOG->BSRR = GPIO_BSRR_BS_14; }</pre>
6.	rozdział 3.4, opis przykładowego rozwiązywania zadania 3.5	Jest: Proste? Dobra. Przyznać się, kto zauważał w kodzie coś dziwnego? ... nie, nie ten rodzynek na początku! Gdzieś w pętli głównej. Pierwsza dziwna rzecz w kodzie to instrukcja DSB po wyłączeniu zegara portów A i G. W STM32F429 występuje mała niedoróbka. A lepiej by było, gdyby było: Proste? Dobra. Przyznać się, kto zauważał w kodzie coś dziwnego? W STM32F429 występuje mała niedoróbka.
7.	rozdział 3.4, opis pułapki dotyczącej rejestru GPIO_BSRR	<i>Od wersji 1.2 Poradnika, przykładowe programy bazują na pliku nagłówkowym pochodzącym z STM32Cube a nie SPL. Opis „pułapki” zaczynający się słowami „Uwaga pułapka!” a kończący się kilka akapitów później słowami „Nie jest to jakiś wielki problem, tylko taki 'trap for young players' :)” nie jest już potrzebny! Nowy plik nagłówkowy nie zawiera tej pułapki. W związku z tym, zbędny opis zostaje usunięty wraz z przynależną mu tabelą 3.4.</i>
8.	rozdział 5.4, pierwszy akapit na stronie 70 (patrz errata do wersji 1.0: tabela 1, pkt. 27.)	Jest: Przerwania (i ogólnie wyjątki) w STM nie są blokowane po wejściu do procedury obsługi tak jak w AVRach. A lepiej by było, gdyby było: W AVRach wejście do procedury obsługi przerwania powodowało automatyczne „zablokowanie” możliwości wystąpienia innych przerwań (wszystkich). W przypadku Cortexa, wejście do procedury obsługi wyjątku blokuje jedynie wyjątki o priorytecie równym i niższym od aktualnie obsługiwanej. Dzięki temu możliwe jest wywłaszczenie przez inny, ważniejszy wyjątek. <i>Wejście do ISR w AVR też blokuje przerwania o równym/nizszym priorytecie, ale ponieważ wszystkie przerwania w AVR mają ten sam priorytet to efekt jest taki, że wszystkie zostają zablokowane.</i>
9.	rozdział 5.4, wypunktowanie na końcu rozdziału	Jest: - po wejściu do handlera wyjątku (np. ISR) wyjątki (w szczególności przerwania) nie są blokowane jak w AVR

	(patrz errata do wersji 1.0: tabela 1, pkt. 34.)	A lepiej by było, gdyby było: - po wejściu do handlera wyjątku (np. ISR) blokowane są tylko wyjątki o równym i niższym priorytecie - wyjątek o wyższym priorytecie może przerwać (wywąszczyć) aktualnie obsługiwany handler (nie jest do tego konieczne jakieś dodatkowe „odblokowywanie przerwań” jak w AVR)
10.	rozdział 9.2, opis rozwiązania zadania 9.1	Jest: <i>Z tego co widzę w przykładach do biblioteki SPL, zegar portu nie jest włączany przy korzystaniu z funkcji tamper</i> A lepiej by było, gdyby było (dodany przypis dolny): <i>Z tego co widzę w przykładach do biblioteki SPL^x, zegar portu nie jest włączany przy korzystaniu z funkcji tamper</i> ^x w chwilach zwątpienia można wesprzeć się przykładowymi programami dołączonymi do biblioteki, np. aby sprawdzić kolejność wykonywania jakichś operacji
11.	rozdział 18, tytuł rozdziału	Jest: <i>18. Changelog („Hominis est errare, insipientis in errore perseverare”)</i> A lepiej by było, gdyby było: <i>18. Errata („Hominis est errare, insipientis in errore perseverare”)</i>
12.	dodatek 4, przedostatni akapit	Jest: <i>Bez tego mechanizmu musielibyśmy zastosować sekwencję RMW, czyli minimum trzy osobne rozkazy. W związku z tym porównywanie czasów dostępu do pamięci aliasu i regionu nie ma w praktyce sensu :)</i> A lepiej by było, gdyby było: <i>Bez tego mechanizmu musielibyśmy zastosować sekwencję RMW, czyli minimum trzy osobne rozkazy. Eksperyment testuje jedynie czas zapisu/odczytu do/z aliasu i regionu BB, a nie czas wykonywania (użytecznych w programie) operacji z użyciem tego mechanizmu. W związku z tym porównywanie uzyskanych wyników nie ma w praktyce sensu i do niczego przydatnego nie prowadzi :)</i>

21.3. Błędy zauważone w wersji 1.2 Poradnika

Tabela 21.3. Błędy zauważone w wersji 1.2 Poradnika (z 16.11.2015r.)

lp.	Dotyczy	Opis
1.	rozdział 2.1, przypis dolny 21	<p>Jest: <i>ARM to jednocześnie nazwa architektury rdzenia i firmy która go produkuje</i></p> <p>A lepiej by było, gdyby było (dodano przypis dolny): <i>ARM to jednocześnie nazwa architektury rdzenia i firmy która ją opracowała; holding ARM sprzedaje licencje pozwalające na implementację rdzeni ARM przez producentów układów (np. przez ST w kontrolerze STM32)</i></p>
2.	rozdział 2.4, czwarty akapit i przypis dolny 33	<p>Jest: <i>Rdzeń jest produkowany przez holding ARM³³</i></p> <p>³³ www.arm.com; <i>ARM to jednocześnie nazwa producenta i architektury</i></p> <p>A lepiej by było, gdyby było (dodano przypis dolny): <i>Rdzeń został opracowany przez holding ARM³³</i></p> <p>³³ www.arm.com; <i>ARM to jednocześnie nazwa architektury i firmy która ją opracowała</i></p>
3.	rozdział 2.6, mini akapit między wykropkowaniami na stronie 27	<p>Jest: <i>Pozostało jeszcze zdobycie kawałka od ARMa. Najnowszą wersję oczywiście pobieramy ze strony producenta (wymaga założenia darmowego konta):</i></p> <p>A lepiej by było, gdyby było (dodano przypis dolny): <i>Pozostało jeszcze zdobycie kawałka od ARMa. Najnowszą wersję oczywiście pobieramy z oficjalnej strony firmy (wymaga założenia darmowego konta):</i></p>
4.	rozdział 2.7, akapit „tertio”	<p>Jest: <i>Notabene w AVR też chyba część rejestrów miała domyślną wartość różną od zera – nie chce mi się szukać...</i></p> <p>A lepiej by było, gdyby było (dodano przypis dolny): <i>Notabene w AVR też chyba część rejestrów miała domyślną wartość różną od zera – nie chce mi się szukać...</i></p> <p><i>Należy również zwrócić uwagę na bity oznaczone jako Reserved (zastrzeżony). Zapisując coś do rejestru zawierającego pola zastrzeżone należy zadbać o to, aby wpisywana wartość zawierała domyślne (początkowe) stany zastrzeżonych bitów. Np. jeśli pole Reserved ma domyślną wartość zero, to w nowej wartości wpisywanej do rejestru też musi być wyzerowane. Cytując RM: „must be kept at reset value”.</i></p>
5.	rozdział 3.4, pierwszy akapit pod rozwiązaniem zadania 3.5	<p>Jest: <i>DSB, w uproszczeniu, wstrzymuje wykonywanie programu do czasu zakończenia operacji na pamięci. Mniejsza z tym. Grunt, że takie obejście jest proste, praktyczne i działa. Trzeba zapamiętać i tyle.</i></p> <p>A lepiej by było, gdyby było (dodano przypis dolny): <i>DSB, w uproszczeniu, wstrzymuje wykonywanie programu do czasu zakończenia operacji na pamięci.</i></p> <p><i>Przy czym nie ma konieczności stosowania akurat instrukcji barierowej DSB. To nie o nią tu chodzi. Ważne jest krótkie opóźnienie (2 cykle zegara dla układów szyny AHB lub 1 + wartość preskalera AHB/APB cykli dla układów szyny APB - wyjaśni się w rozdziale 17) pomiędzy włączeniem zegara dla układu peryferyjnego, a pierwszym dostępem do jego rejestrów (np. zapisem do rejestrów konfiguracyjnych). Opóźnienie można uzyskać dowolną metodą, byleby była skuteczna. Zamiast instrukcji barierowej, można tak zaplanować program, aby konfiguracja peryferiali nie występowała natychmiast po włączeniu ich zegarów. Każde rozwiązanie jest dobre! W większości przykładowych</i></p>

programów (dla F4) będę stosował instrukcję barierową nawet jeśli opóźnienie będzie wynikało z innych działań w programie. DSB będzie kluło w oczy i przypominało o erracie :)

21.4. Błędy zauważone w wersji 1.3 Poradnika

Tabela 21.4. Błędy zauważone w wersji 1.3 Poradnika

lp.	Dotyczy	Opis
1.	errata, rozdział 1.1, tabela 1, punkt 13	<p>Jest: <i>opis wyjątku Reset</i></p> <p>A lepiej by było, gdyby było: <i>opis wyjątku HardFault</i></p>
2.	rozdział 16, tytuł rozdziału	<p>Jest: <i>16. Pamięć Flash („Variatio delectat”)</i></p> <p>A lepiej by było, gdyby było: <i>16. Pamięć Flash i bootowanie („Variatio delectat”)</i></p>
3.	rozdział 17.2, rozwiązanie zadania 17.1, czwarta linia kodu	<p>Jest: $RCC->CFG_R = RCC_CFR_PLLMULL9 RCC_CFR_PLLSRC_HSE RCC_CFR_ADCPRE_DIV6 ...$</p> <p>A lepiej by było, gdyby było: $RCC->CFG_R = RCC_CFR_PLLMULL9 RCC_CFR_PLLSRC RCC_CFR_ADCPRE_DIV6 ...$</p>
4.	rozdział 17.3, opis rozwiązania zadania 17.3, komentarz do piątej linijki kodu	<p>Jest: <i>konfiguracja Odchylacza Częstotliwości (Spread Spectrum),</i></p> <p>A lepiej by było, gdyby było (m.in. dodano przypis dolny): <i>konfiguracja Rozmycia Częstotliwości ^x (Spread Spectrum),</i></p> <p><small>^x chodzi o rozmycie widma generowanych zakłóceń - odsyłam do wątku poświęconego Poradnikowi (http://www.elektroda.pl/rtvforum/viewtopic.php?p=15177034#15177034)</small></p>

21.5. Zmiany dokonane w wersji 1.4 Poradnika

Tabela 21.5. Zmiany dokonane w wersji 1.4 Poradnika

lp.	Dotyczy	Opis
1.	rozdział 16	Dodano podrozdziały: 16.5. Kod w pamięci SRAM - po co? (F103 i F429) 16.6. Funkcja w pamięci sram - jak? (F103 i F429) 16.7. Cały program w pamięci sram - jak? (F103 i F429)
2.	rozdział 18	Dodano rozdział: 18. Hashing randomly encrypted CRC (“Contra facta non valent argumenta”) 18.1. Wstęp z niespodzianką 18.2. Nic nie dzieje się przypadkiem (F429) 18.3. Suma kontrolna CRC32 (F103 i F429) 18.4. Sprzętowe CRC8, CRC16 i dowolny wielomian (F103 i F429)
3.	rozdział 19	Dodano rozdział: 19. Interfejs SPI (“Potius sero quam numquam”) 19.1. Wstęp (F103 i F429) 19.2. Master SPI (F103 i F429) 19.3. Master i Slave w jednym SP(al)I domu (F103 i F429) 19.4. Pół-puplex na dwa mikrokontrolery (F103 i F429) 19.5. CRC w SPI (F103 i F429)

21.6. Zmiany dokonane w wersji 1.5 Poradnika

Tabela 21.6. Zmiany dokonane w wersji 1.5 Poradnika

lp.	Dotyczy	Opis
1.	rozdział 2.6, opis zdobywania pliku nagłówkowego mikrokontrolera	<p>Jest: <i>Resztę można wywalić.</i></p> <p>A lepiej by było, gdyby było <i>Resztę można wywalić (w pliku nagłówkowym należy zakomentować odwołanie do pliku system_stm32f..., nie jest on do niczego potrzebny).</i></p>
2.	rozdział 2.6, opis plików z paczki CMSIS pobranej ze strony ARM	<p>Jest:</p> <ul style="list-style-type: none"> - <i>core_cmFunc.h</i> – funkcje operujące na rejestrach specjalnych rdzenia, np. umożliwiające zmianę wskaźnika stosu, priorytetów przerwań, itd... - <i>core_cmInstr.h</i> – funkcje intrinsic czyli proste wstawki asm napisane w C umożliwiające wywołanie konkretnych rozkazów asm, np. <i>_WFI()</i>, <i>_NOP()</i>, itd... <p>A lepiej by było, gdyby było:</p> <ul style="list-style-type: none"> - <i>cmsis_gcc.h</i>^x – funkcje operujące na rejestrach specjalnych rdzenia (np. dostęp do wskaźnika stosu i konfiguracja przerwań) oraz funkcje intrinsic, czyli proste wstawki asm umożliwiające wywołanie, z poziomu języka C, konkretnych rozkazów asm (np. <i>_WFI()</i>, <i>_NOP()</i>, itd...) <hr/> <p><i>Uwaga! W starszych wersjach CMSIS (chyba do wersji 4.3) funkcje operujące na rejestrach specjalnych rdzenia i funkcje intrinsic, były rozdzielone na kilka plików (m.in. <i>core_cmFunc.h</i>, <i>core_cmInstr.h</i>). W najnowszym CMSISie nastąpiła mała reorganizacja. Funkcje zostały umieszczone we wspólnym pliku. Plik ten występuje w kilku wersjach, dla różnych kompilatorów, np.:</i></p> <ul style="list-style-type: none"> - <i>cmsis_armcc.h</i> - kompilator RealView - <i>cmsis_gcc.h</i> - kompilator GNU GCC - <i>cmsis_css.h</i> - kompilator TI CSS - ... <p><i>Pliki <i>core_cmFunc.h</i> oraz <i>core_cmInstr.h</i> dalej są obecne w paczce. Zawierają one jedynie kilka warunków preprocesora. Ich celem jest m.in. „zainkludowanie” odpowiedniego pliku z funkcjami, zależnie od używanego kompilatora. Polecam sobie poprzednio.</i></p> <p><i>Osobiście nie lubię jak mi się płczę w projekcie pliki, o wątpliwej przydatności. Korzystam tylko z kompilatora gcc, więc interesuje mnie tylko plik przygotowany dla tegoż kompilatora - <i>cmsis_gcc.h</i>. Jest tylko małe „ale”. Plik z definicjami dla rdzenia (<i>core_cm.h</i>) odwołuje się do plików <i>core_cmFunc.h</i> oraz <i>core_cmInstr.h</i> („inkluduje” je) - jeżeli nie będzie ich w projekcie to pojawi się błąd. Rozwiązań są dwa:</i></p> <ul style="list-style-type: none"> - <i>można jednak skopiować te pliki do projektu</i> - <i>można zamienić odwołania do nich, na odwołanie bezpośrednio do <i>cmsis_gcc.h</i></i> <hr/> <p>^x lub inna wersja, jeśli nie korzystasz z kompilatora gcc</p>

<p>3.</p> <p>rozdział 2.6, podsumowanie</p>	<p>Jest:</p> <ul style="list-style-type: none"> - na start należy zaopatrzyć się w pliki: - <i>core_cmx.h</i> - <i>core_cmFunc.h</i> - <i>core_cmInstr.h</i> - oraz plik nagłówkowy mikrokontrolera <i>stm32fxx.h</i> <p>A lepiej by było, gdyby było:</p> <ul style="list-style-type: none"> - na start należy zaopatrzyć się w pliki: - <i>core_cmx.h</i> - <i>cmsis_gcc.h</i> - oraz plik nagłówkowy mikrokontrolera <i>stm32fxx.h</i>
<p>4.</p> <p>rozdział 11.1, opis System Reset</p>	<p>Jest:</p> <ul style="list-style-type: none"> - nie ma <i>wcale</i> - <p>A lepiej by było, gdyby było (dodany nowy akapit na końcu opisu):</p> <p>Warto zwrócić uwagę na to, że nóżka NRST mikrokontrolera nie działa tylko jako wejście. Każde źródło resetu systemowego (np. reset wywołany programowo) powoduje pojawienie się na niej stanu niskiego. Z tego względu podłączenie nóżki na stałe do VCC (co było możliwe w AVR), jest kiepskim pomysłem. Wyprowadzenie NRST można wykorzystać jako wyjście sygnału resetu dla innych układów cyfrowych w systemie. Odsyłam do schematu <i>Simplified diagram of the reset circuit</i> w RMie.</p>

21.7. Zmiany dokonane w wersji 1.6 Poradnika

Tabela 21.7. Zmiany dokonane w wersji 1.6 Poradnika

lp.	Dotyczy	Opis
1.	rozdział 2.7, akapit „Quarto!”	<p>Jest: <i>Jedyny znany mi sposób zablokowania STMa na amen, to włączenie drugiego stopnia ochrony pamięci przed odczytem w STM32F429.</i></p> <p>A lepiej by było, gdyby było: <i>Jedyny znany mi sposób zablokowania STMa na amen, to włączenie drugiego stopnia ochrony pamięci przed odczytem w STM32F334 lub STM32F429.</i></p>
2.	rozdział 3.1, tytuł rozdziału	<p>Jest: <i>3.1. Ogarnąć nóżki w F103</i></p> <p>A lepiej by było, gdyby było: <i>3.1. Ogarnąć nóżki w F103 (F103)</i></p>
3.	rozdział 3.3, tytuł rozdziału	<p>Jest: <i>3.3. Atomowo macham nogą i nie tylko (F103 i F429)</i></p> <p>A lepiej by było, gdyby było: <i>3.3. Atomowo macham nogą i nie tylko</i></p>
4.	rozdział 3.5, mniej więcej czwarty akapit	<p>Jest: <i>Po szczegóły odsyłam do datasheets - rozdział...</i></p> <p>A lepiej by było, gdyby było: <i>Po szczegóły odsyłam do datasheets układu STM32F429 - rozdział...</i></p>
5.	rozdział 3.5, mniej więcej ósmy akapit	<p>Jest: <i>To znaczy np.: nóżka PA15 może być powiązany z funkcjami alternatywnymi: AF0, AF1, AF5, AF6, AF15.</i></p> <p>A lepiej by było, gdyby było: <i>To znaczy np.: nóżka PA15, w F429, może być powiązana z funkcjami alternatywnymi: AF0, AF1, AF5, AF6, AF15.</i></p>
6.	rozdział 3.7, tytuł rozdziału	<p>Jest: <i>3.7. Elektryczna strona medalu (F103 i F429)</i></p> <p>A lepiej by było, gdyby było: <i>3.7. Elektryczna strona medalu</i></p>
7.	rozdział 4.1, pierwszy akapit	<p>Jest: <i>BB umożliwia dokonywanie atomowych operacji bitowych na pamięci SRAM i rejestrach peryferiali.</i></p> <p>A lepiej by było, gdyby było: <i>BB umożliwia dokonywanie atomowych (z punktu widzenia programu, patrz rozdział 4.2) operacji bitowych na pamięci SRAM i rejestrach peryferiali.</i></p>
8.	rozdział 4.1, podsumowanie	<p>Jest: <i>bit banding jest mechanizmem umożliwiającym przeprowadzanie atomowych operacji na bitach</i></p> <p>A lepiej by było, gdyby było: <i>bit banding jest mechanizmem umożliwiającym przeprowadzanie atomowych (z punktu widzenia programu) operacji na bitach</i></p>

		Dodany fragment po pierwszym akapicie: W tym miejscu musimy się na chwilę zatrzymać i doprecyzować jedną rzecz. Jeżeli nie interesują Cię mechanizmy działania bit bandingu, a jedynie jego „użytkowa strona” to możesz ten kawałek opuścić. Ale, coś za coś, wtedy nie zrozumiesz haczyka opisywanego w rozdziale 4.6 i będziesz musiał przyjąć go na wiarę. Przy okazji omawiania bit bandingu, wiele razy pojawią się określenia: „atomowo”, „bez sekwencji read-modify-write”. Warto sobie zdawać sprawę, że ten brak sekwencji RMW występuje tylko z punktu widzenia programu. Bit banding z punktu widzenia pamięci, której zawartość modyfikujemy z wykorzystaniem BB, niczym nie różni się od „zwyczajnego” dostępu. Na pamięci wykonywana jest zwyczajna operacja RMW. Powyżej napisałem, że wpisanie 0 lub 1 do słowa aliasu uruchamia jakieś magiczne hokus-pokus, które ustawia bit w regionie BB. To „hokus-pokus” to sprzętowy mechanizm, który wykonuje na pamięci (regionie BB) sekwencję read-modify-write. Cała magia BB polega na tym, że zamiast robić w programie nie-atomową operację RMW, robi ją za nas sprzęt (jakiś System Bus Controller, czy coś takiego). Czyli: w programie wykonujemy (atomowy) zapis (zera lub jedynki) do pamięci aliasu, a to wyzwala sprzętową sekwencję read-modify-write. Ponadto sprzęt (System Bus Controller) pilnuje, aby w czasie wykonywania tego sprzętowego RMW, procesor nie modyfikował pamięci. Styknie tego, wracamy do głównego wątku :]
10.	rozdział 4.2, podsumowanie	Dodana nowa kropka: zapis do aliasu BB wyzwala sekwencję RMW realizowaną sprzętowo
11.	rozdział 4.5, ostatni akapit	Jest: Powyższe reguły podyktowane są subiektywną wygodą i przejrzystością zapisu. Jasnym jest, że jeśli zależy nam na uzyskaniu atomowości to nie ma o czym gadać i podejście klasyczne odpada w przedbiegach. A lepiej by było, gdyby było: Powyższe reguły podyktowane są subiektywną wygodą i przejrzystością zapisu. Jasnym jest, że jeśli zależy nam na uzyskaniu atomowości to nie ma o czym gadać i podejście klasyczne odpada w przedbiegach. Ponadto trzeba zwrócić uwagę na mały haczyk związany z bit bandingiem, opisany w nadchodzącym rozdziale (rozdział 4.6).
12.	rozdział 4.6, treść całego rozdziału	Jest: A może nie ma :) No może taki mały zadziorek na upartego. Żadnych wiedzy i szukania dziury w całym odsyłam do dodatku 4. A lepiej by było, gdyby było: - nowa treść rozdziału znajduje się pod niniejszą tabelką (dotyczy tylko erraty w osobnym pdfie) -
13.	rozdział 4.7, cały rozdział	Jest: - no właśnie jest - A lepiej by było, gdyby: - lepiej by było, gdyby go nie było - (z bólem żegnamy rozdział 4.7, na zawsze pozostanie w naszej pamięci i Erracie)
14.	rozdział 6.1, tytuł rozdziału	Jest: 6.1. Blink me baby one more time (F103, F429) A lepiej by było, gdyby było: 6.1. Blink me baby one more time
15.	rozdział 7, tytuł rozdziału	Jest: 7. Przerwania zewnętrzne („Facta sunt verbis difficilora”) A lepiej by było, gdyby było: 7. Blok EXTI i przerwania zewnętrzne („Facta sunt verbis difficilora”)
16.	rozdział 7, wprowadzenie	Jest: Mamy opanowane GPIO i przerwania – jak to połączyć? EXTI (External Interrupt / Event Controller) czyli przerwania zewnętrzne.

		<p>A lepiej by było, gdyby było: Mamy opanowane GPIO i przerwania – jak to połączyć? EXTI (Extended Interrupt / Event Controller) czyli układ obejmujący m.in. przerwania zewnętrzne.</p>
17.	rozdział 8.19, tytuł rozdziału	<p>Jest: 8.19. Różnice między F103 i F429</p> <p>A lepiej by było, gdyby było: 8.19. Różnice między F103 i F429 (F103 i F429)</p>
18.	rozdział 20, tytuł rozdziału	<p>Jest: 20. Errata („Hominis est errare, insipientis in errore perseverare”)</p> <p>A lepiej by było, gdyby było: 20. Errata i changelog („Hominis est errare, insipientis in errore perseverare”)</p>

A gdzie jest haczyk? (F103, F334 i F429)

Kiedyś myślałem, że w sumie to haczyka nie ma. Całkiem niedawno, jakoś po publikacji Poradnika w wersji 1.5 :), zupełnie przypadkiem nadziałem się na coś co mnie zupełnie zaskoczyło i pokazało, że haczyk jest i to całkiem zacny. Nie wiem jak mogłem to wcześniej przegapić...

W rozdziale 4.2, we fragmencie wydzielonym za pomocą odcinków horyzontalnych, opisałem odrobinę sposób działania bit bandingu. Tzn. że modyfikacja bitu, z pomocą bb, to sekwencja *read-modify-write* realizowana na poziomie sprzętu. No i jest ona „atomowa”, bo *System Bus Controller* wstrzymuje procesor, jeśli ten próbuje dobrać się do pamięci w trakcie trwania operacji RMW.

Haczyki (z powyższym związanego) na jakie możemy się nadziać są dwa. I są bardzo podobne do siebie. Oba dotyczą dostępu do wszelkiej maści rejestrów statusowych układów peryferyjnych. Ten rozdział, niestety, troszkę wyprzedza nasz „stan wiedzy”. To czego na razie nie rozumiesz, przyjmuj na wiarę :) Potem tu wróćisz, gdy dzięki dalszej lekturze Poradnika i wiedzy zeń płynącej, rozkwitniesz niczym... niczym coś co rozkwita... dajmy temu spokój.

Pierwszy haczyk dotyczy rejestrów zawierających bity kasowane poprzez wpisanie doń jedynki. W dokumentacji (RM) bity takie oznaczone są skrótem³²⁸ *rc_w1*. Skrót oznacza, że bit może być czytany (*read*) oraz kasowany (*clear*) poprzez wpisanie jedynki (*write 1*). Przykładem takiego rejestru jest регистр zawierający flagi przerwań zewnętrznych (EXTI_PR). Wyobraźmy sobie teraz, że chcemy skasować jedną z flag tego rejestru i, niezbyt szczęśliwie, postanowiliśmy wykorzystać do tego bit banding. Posiłkując się naszym makrem, piszemy więc na przykład coś takiego:

```
BB(EXTI->PR, EXTI_PR_PR5) = 1;
```

zgadza się? No zgadza - flaga kasowana wpisaniem jedynki, wpisujemy jedynkę na odpowiednią pozycję. Skasowaliśmy flagę, pozamiatane, w czym problem? Problem w tym, że jeżeli w rejestrze były ustawione jeszcze jakieś flagi *rc_w1*, to je też pozamiataliśmy przy okazji. Wiesz czemu?

BB to sekwencja RMW, tylko sprzętowa. Zawartość rejestru EXTI_PR została odczytana. „Coś sprzętowego”, co realizuje operacje bit bandingowe, ustawiło w tej wartości bit EXTI_PR_PR5 po czym wynik został nazad wpisany do rejestru. A co jeżeli kilka flag w rejestrze 328 lista wszystkich oznaczeń znajduje się na początku RMa (*List of abbreviations for registers*)

było ustawionych? Wtedy w odczytanej wartości było kilka jedynek i teraz, gdy została ona wpisana do EXTI_PR, wszystkie te flagi zostały skasowane! Olaboga! Mocium Panie, dramat! Przemyśl sobie dobrze ten przypadek, bo łatwiej zapamiętać coś co się rozumie :) Rada? Rada jest prosta: nie modyfikować rejestrów zawierających pola *rc_wI* za pomocą bb a jeśli już, to po dogłębnym przemyśleniu sprawy. I tyle. Na szczęście nie ma zbyt wielu takich rejestrów.

Czas na drugi haczyk. Wynika on z tego, że z punktu widzenia pamięci, operacje bb niczym nie różnią się od zwyczajnych (programowych) sekwencji RMW. BB też wywołuje RMW, jeno realizowane przez sprzęt. I tak jak wspominałem, w rozdziale 4.2, procesor jest blokowany jeśli w czasie sekwencji RMW będzie próbował uzyskać dostęp do tej pamięci. Ale! Nie tylko procesor może modyfikować „pamięć”. Jeżeli za pomocą bb będziemy modyfikowali rejesty układowego peryferyjnego, np. licznika, to ten licznik nie zostanie zablokowany na czas operacji RMW. Czyli jest ryzyko, że zawartość jego rejestrów (np. rejestru statusowego) się zmieni. W tym momencie to nie będzie operacja atomowa! A czym grozi zmiana zawartości rejestru modyfikowanego nieatomową sekwencją RMW już wiemy z rozdziału 3.3.

Rada? Trzeba myśleć co się robi :) To nie jest problem wynikający z bit bandingu! W takim samym stopniu dotyczy on bezpośrednich operacji na pamięci, ale korzystając z bb trudniej dostrzec zagrożenie! Za każdym razem, gdy modyfikowane są rejesty zawierające pola których stan zmienia się sprzętowo, należy przemyśleć, czy ewentualna operacja RMW czegoś nam nie nadpisze. Innej rady nie ma znam :)

Zdaję sobie sprawę, że w tym momencie nie wszystko rozumiesz. Luz. Wszystko się z czasem ułoży. Być może pokazałem bb w trochę złym świetle. Ale tak nie jest! To naprawdę użyteczny mechanizm i nie trzeba się go bać. Serio!

No w sumie to znam jeszcze trzy pułapki związane z bb... O jednej już wspomniałem w rozdziale 4.2. Chodzi o to, że w niektórych mikrokontrolerach bb nie obejmuje wszystkich układów peryferyjnych. Jak się tego nie sprawdzi i z rozpoczętu zacznie się stosować bb to wychodzą bzdury. Na szczęście to jest stosunkowo łatwy do wykrycia błąd.

Druga pułapka dotyczy pamięci CCM (*Core Coupled Memory*) występującej w mikrokontrolerach z rdzeniem Cortex-M4. Bit banding nie obejmuje pamięci CCM. Nieśmiało strzelić że to, że ktoś poczynający będzie zaraz kombinował z bb i pamięcią CCM na początku swojej przygody z STMami, jest równie prawdopodobne jak to, że komuś uda się pomalować amelinium.

Co do ostatniej pułapki... pułapka to złe słowo... tak czy siak - żadnych wiedzy i szukania dziury w całym odsyłam do dodatku 4.

Co warto zapamiętać z tego rozdziału?

- należy diabelnie uważać modyfikując za pomocą bb rejesty zawierające pola *rc_wI*
 - w ogóle należy uważać z rejestrami, których pola są modyfikowane sprzętowo (bb nie jest dla nich atomowy!)
 - nie wszystkie rejesty układów peryferyjnych muszą być objęte bb
 - bb wbrew pozorom nie jest podstępny i jest naszym przyjacielem
-

21.8. Zmiany dokonane w wersji 1.7 Poradnika

Tabela 21.8. Zmiany dokonane w wersji 1.7 Poradnika (przede wszystkim dodanie wsparcia dla F334)

lp.	Dotyczy	Opis
1.	rozdział 2.3	Dodano: <i>opis płytka Nucleo-F334R8</i>
2.	rozdział 2.7, akapit „Sexto!”	Dodana nowa kropka: - F334 w odniesieniu do mikrokontrolera STM32F334R8T6 z zestawu Nucleo-F334R8
3.	rozdział 2.7, akapit „Octava!”	Jest: <i>Ponadto, pomimo podziału na rozdziały dotyczące mikrokontrolerów F103 i F429, informacje dotyczące poszczególnych rodzin czasem się przeplatają. Zdecydowanie należy przeczytać oba, nawet jeśli nie zamierzasz korzystać np. z F429.</i> A lepiej by było, gdyby było: <i>Ponadto, pomimo podziału na rozdziały dotyczące poszczególnych mikrokontrolerów (F103, F334, F429), informacje dotyczące różnych rodzin czasem się przeplatają. Zdecydowanie należy przeczytać wszystkie, nawet jeśli nie zamierzasz korzystać np. z F429. Jeżeli przy tytule rozdziału nie ma wyszczególnionej konkretnej rodziny mikrokontrolerów, to dotyczy on wszystkich omawianych rodzin.</i>
4.	rozdział 3.4, tytuł rozdziału	Jest: 3.4. Cortex-M4 wybieram Cię! (F429) A lepiej by było, gdyby było: 3.4. Cortex-M4 wybieram Cię! (F334 i F429)
5.	rozdział 3.4, pierwszy akapit	Jest: <i>Przyjrzymy się teraz jak to wygląda w STM32F429. Jest to o wiele bardziej rozbudowany mikrokontroler, więc i portom dostało się więcej opcji.</i> A lepiej by było, gdyby było: <i>Przyjrzymy się teraz jak to wygląda w STM32F334 i STM32F429. Najpierw omówimy F429, a F334 zostawimy sobie na deser. Sq to o wiele bardziej rozbudowane mikrokontrolery, więc i portom dostało się więcej opcji.</i>
6.	rozdział 3.4, drugi akapit	Jest: <i>To co z miejsca rzuca się w oczy, po otwarciu RMa mikrokontrolera STM32F429, to inne rejestrze konfiguracyjne portów</i> A lepiej by było, gdyby było: <i>To co z miejsca rzuca się w oczy, po otwarciu RMa mikrokontrolera STM32F429 (mikrokontrolerem F334 zajmiemy się za chwilę), to inne rejestrze konfiguracyjne portów</i>
7.	rozdział 3.4, nowe akapity na końcu rozdziału	Dodany: <i>opis różnic między GPIO w F334 i F29 oraz zadanie domowe bazujące na F334</i>
8.	rozdział 3.4, nowa kropka w podsumowaniu	Nowa kropka w podsumowaniu rozdziału: - porty w F334 i F429 działają z grubsza tak samo
9.	rozdział 3.5, tytuł rozdziału	Jest: 3.5. Wybór funkcji alternatywnej (F429) A lepiej by było, gdyby było: 3.5. Wybór funkcji alternatywnej (F334 i F429)

10.	rozdział 3.5, cały rozdział	Zmieniono: Wszystkie odwołania do mikrokontrolera F429 (z wyjątkiem wyjątków wynikających z dalszych punktów erraty) zamieniono na odwołania do F334 i F429.
11.	rozdział 3.5, mniej więcej piąty akapit	<p>Jest: Sposób konfiguracji funkcji alternatywnych portu zostanie omówiony szczegółowo jak poznamy jakieś peryferia :) Na razie tylko sygnalizuję zagadnienie. Przy konfiguracji nóżki w trybie alternatywnym należy wybrać tryb alternatywny pinu i za pomocą rejestrów GPIO_AFRL/H wybrać odpowiedni numer funkcji alternatywnej zgodnie z tabelą z datasheeta.</p> <p>A lepiej by było, gdyby było: Sposób konfiguracji funkcji alternatywnych portu zostanie omówiony szczegółowo jak poznamy jakieś peryferia :) Na razie tylko sygnalizuję zagadnienie. Przy konfiguracji nóżki w trybie alternatywnym należy (i to jest trzecia z różnic między portami GPIO w F334 i F429):</p> <ul style="list-style-type: none"> - w przypadku mikrokontrolera F334 skonfigurować kolejno: <ul style="list-style-type: none"> - numer wybranej funkcji alternatywnej (rejestry GPIO_AFRL/H) zgodnie z tabelą z datasheeta - konfigurację wyprowadzenia (rejestry GPIO_OTYPER, GPIO_OSPEEDR, GPIO_PUPDR) - tryb alternatywny pinu (w rejestrze GPIO_MODER), - w przypadku mikrokontrolera F429 RM zaleca inną kolejność konfiguracji: <ul style="list-style-type: none"> - tryb alternatywny pinu (w rejestrze GPIO_MODER) - konfigurację wyprowadzenia (rejestry GPIO_OTYPER, GPIO_OSPEEDR, GPIO_PUPDR) - numer wybranej funkcji alternatywnej (rejestry GPIO_AFRL/H) zgodnie z tabelą z datasheeta <p>Przypuszczam, że powyższe różnice nie mają większego znaczenia, ale dla świętego spokoju można zmienić funkcję konfigurującą porty (z dodatku 2). Wersja dla F334 wygląda praktycznie tak samo, zmienia się kolejność konfiguracji rejestrów.</p>
12.	rozdział 3.5, zadanie 3.7	<p>Jest: Zadanie domowe 3.7: na podstawie dokumentacji zidentyfikować nóżki odpowiadające kanałom 1, 2, 3 licznika TIM1 i obczać sposób konfiguracji (numery funkcji alternatywnych).</p> <p>A lepiej by było, gdyby było: Zadanie domowe 3.7: na podstawie dokumentacji mikrokontrolerów F334 i F429 zidentyfikować nóżki odpowiadające kanałom 1, 2, 3 licznika TIM1 i obczać sposób konfiguracji (numery funkcji alternatywnych).</p>
13.	rozdział 3.5, rozwiązywanie zadania 3.7	Zmieniono: Rozwiązywanie zostało podzielone na dwie wersje, osobno dla F334 i F429.
14.	rozdział 3.5, podsumowanie	<p>Jest:</p> <ul style="list-style-type: none"> - numery funkcji alternatywnych: tabela STM32F427xx and STM32F429xx alternate function mapping - numery wyprowadzeń: tabela STM32F427xx and STM32F429xx pin and ball definitions <p>A lepiej by było, gdyby było:</p> <ul style="list-style-type: none"> - numery funkcji alternatywnych: <ul style="list-style-type: none"> - F334: datasheet → tabela Alternate functions - F429: datasheet → tabela STM32F427xx and STM32F429xx alternate function mapping - numery wyprowadzeń: <ul style="list-style-type: none"> - F334: datasheet → tabela STM32F334x4/6/8 pin definitions - F429: datasheet → tabela STM32F427xx and STM32F429xx pin and ball definitions

15.	rozdział 3.6, cały rozdział	Zmieniono: Wszystkie odwołania do mikrokontrolera F429 zamieniono na odwołania do F334 i F429. Chyba, że z erraty wynika co inszego.
16.	rozdział 3.7, pierwszy akapit	Jest: Poniżej, w tabeli 3.5, podaję zestawienie najważniejszych wartości na podstawie datasheetów STM32F103 oraz STM32F429. A lepiej by było, gdyby było: Poniżej, w tabeli 3.5, podaję zestawienie najważniejszych wartości na podstawie datasheetów STM32F103VC, STM32F334R8 oraz STM32F429ZI.
17.	rozdział 3.7, tabela 3.5	Dodano: nową kolumnę dla mikrokontrolera F334.
18.	rozdział 4.2, uwagi na końcu rozdziału (przed podsumowaniem)	Jest: - bit band region nie musi obejmować wszystkich peryferiów mikrokontrolera, np. w mikrokontrolerze STM32F303 BB nie obejmuje rejestrów portów GPIO i ADC A lepiej by było, gdyby było: - bit band region nie musi obejmować wszystkich peryferiów mikrokontrolera Ta ostatnia uwaga chyba wymaga małego komentarza. Za pomocą bb można modyfikować ^x pamięć, która leży w zakresie adresów objętych regionem bb. Region bb dla układów peryferyjnych, zarówno w Cortexie M3 jak i M4, zaczyna się od adresu 0x4000 0000 kończy zaś pod adresem 0x400F FFFF (szczególnie nadchodzące w rozdziale 4.3). BB ma dostęp tylko do peryferiów leżących w tym obszarze! No to teraz, posługując się mapką pamięci (Memory Map) z RMA posiadanego mikrokontrolera sprawdź, które układy nie załapały się na bb. Nie czytaj dalej póki sam nie sprawdzisz, ćwiczeń nigdy za dużo! - F103: USB OTG FS i FSMC - F334: GPIOA, GPIOB, GPIOC, GPIOD, GPIOF, ADC1, ADC2 - F429: USB OTG FS, DCMI, CRYP, HASH, RNG, FSMC, FMC Układami USB, DCMI i kontrolerami pamięci F(S)MC i tak nie będziemy się na razie zajmować. Ale o portach I/O i przetwornikach ADC w F334 radzę pamiętać :) ^x z rozpetto piszę o modyfikowaniu, ale proszę nie zapominaj, że bb pozwala też odczytywać pamięć :)
19.	rozdział 5, wprowadzenie	Jest: Wszystko co przeczytasz w tym rozdziale, dotyczy zarówno F103 (Cortex-M3) jak i F429 (Cortex-M4). Chyba, że wyraźnie napisano inaczej :) A lepiej by było, gdyby było: Wszystko co przeczytasz w tym rozdziale, dotyczy zarówno F103 (Cortex-M3) jak i F334/F429 (Cortex-M4). Chyba, że wyraźnie napisano inaczej :)
20.	rozdział 5.2, tabela 5.1, opis wyjątku NMI	Jest: Przerwania nie-maskowalne (Non Maskable Interrupts). Nie można ich wyłączyć! W STMach przerwanie NMI jest odpalane jeśli system kontroli sygnału zegarowego (Clock Security System) wykryje awarię zewnętrznego rezonatora (patrz zadanie 17.2). A lepiej by było, gdyby było: Przerwania nie-maskowalne (Non Maskable Interrupts). Nie można ich wyłączyć! W STMach przerwanie NMI jest odpalane jeśli system kontroli sygnału zegarowego (Clock Security System) wykryje awarię zewnętrznego rezonatora (patrz zadanie 17.2) lub w przypadku wykrycia błędu bitu parzystości pamięci SRAM (dotyczy tylko F334).
21.	rozdział 6.1, przed podsumowaniem	Dodano sympatyczne zadanie (wraz z rozwiązaniem i głębokim komentarzem): Zadanie domowe 6.5: napisać program migający diodą, oparty o przerwanie SysTicka, dla... kto zgadnie? F334 :]

22.	rozdział 7.1, opis rozwiązania zadania 7.1, opis 25 linii kodu	Jest: Reszta kodu nie zawiera już nic nowego. A lepiej by było, gdyby było: Uwaga! Koniecznie przeczytaj opis rozwiązania analogicznego zadania, z rozdziału poświęconego mikrokontrolerowi F334 (zadanie 7.4; 26 i 27 linia kodu).
23.	rozdział 7.1, opis rozwiązania zadania 7.2	Jest: Odpowiedź: [...] A lepiej by było, gdyby było: Odpowiedź: (czcionka w kolorze tła - aby odczytać skopiuj do innego edytora, ale najpierw sprawdź sam! Wszystko wydaje się proste jak się zna odpowiedź.) [...]
24.	rozdział 7.3	Dodany nowy rozdział: 7.3. EXTI (F334)
25.	rozdział 8.20	Dodany nowy rozdział: 8.20. Licznikowe indywidualum (F334)
26.	rozdział 9.1, wypunktowanie pod drugim akapitem	Jest: Co do kwestii elektrycznych: napięcie baterii: 1,8 - 3,6V pobór prądu (orientacyjnie): ~1,5μA A lepiej by było, gdyby było: Co do kwestii elektrycznych: - napięcie baterii: - F103: 1,8 - 3,6V - F334: 1,65 - 3,6V - F429: 1,65 - 3,6V - pobór prądu (orientacyjnie): - F103: ~1,5μA - F334: ~1μA - F429: ~10μA
27.	rozdział 9.1, trzeci akapit	Jest: W zestawie HY-mini jest gniazdo na baterię CR1220. Swoją włożylem z dwa lata temu i dalej działa :) W STM32F429i-Disco nóżka VBAT jest połączona z głównym zasilaniem. Bez drobnych modyfikacji nie jest możliwe dołączenie zewnętrznego źródła podtrzymywania. A lepiej by było, gdyby było: W zestawie HY-mini jest gniazdo na baterię CR1220. Swoją włożylem z dwa lata temu i dalej działa :) W STM32F429i-Disco nóżka VBAT jest połączona z głównym zasilaniem. Bez drobnych modyfikacji nie jest możliwe dołączenie zewnętrznego źródła podtrzymywania. W zestawie Nucleo-F334R8 w ogóle nie ma możliwości podłączenia źródła do Vbat. To wyprowadzenie jest na stałe połączone z Vcc i nie przewidziano możliwości zmiany konfiguracji.
28.	rozdział 9.7	Nowy rozdział: 9.7. Backup Registers i RTC (F334)
29.	rozdział 10.4	Nowy rozdział: 10.4. Układy watchdog w mikrokontrolerze F334 (F334)
30.	rozdział 11.3, pierwszy akapit	Jest: Pobór prądu z baterii nie przekracza 1,5μs. A lepiej by było, gdyby było: Maksymalny pobór prądu z baterii, nie przekracza 10μA.

31.	rozdział 11.4	Nowy rozdział: 11.4. Zasilanie (F334)
32.	rozdział 11.5 (11.4 przed zmianą 31) tytuł rozdziału	Jest: 11.5. Debugowanie a tryby uśpienia (F103 i F429) A lepiej by było, gdyby było: 11.5. Debugowanie a tryby uśpienia
33.	rozdział 11.6 (11.5 przed zmianą 31), tabela 11.1, opis stop mode	Jest: LPDS = 1: $3,6\mu s$ LPDS = 0: $5,4\mu s$ A lepiej by było, gdyby było: LPDS = 1: $5,4\mu s$ LPDS = 0: $3,6\mu s$
34.	rozdział 11.8	Nowy rozdział: 11.8. Tryby obniżonego poboru mocy (F334)
35.	rozdział 12.4	Nowy rozdział: 12.4. DMA (F334)
36.	rozdział 13.7, opis parametrów czujnika temperatury	Jest: Parametry czujnika temperatury (dalej ostrzegaję że jest kiepski) i źródła referencyjnego: A lepiej by było, gdyby było (dodany przypis dolny): Parametry czujnika temperatury (dalej ostrzegaję że jest kiepski ^x) i źródła referencyjnego: ^x w mikrokontrolerze zaszyte są dane kalibracyjne czujnika temperatury i źródła odniesienia, patrz zadanie 13.12
37.	rozdział 13.8	Nowy rozdział: 13.8. Różnice w STM32F334 (F334)
38.	rozdział 13.9 (13.8 przed zmianą 36), tytuł rozdziału	Jest: 13.8. Końcowe uwagi (F103 i F429) A lepiej by było, gdyby było: 13.9. Końcowe uwagi
39.	rozdział 13.9 (13.8 przed zmianą 36)	Jest: Dodatkowo chciałbym jeszcze zwrócić uwagę na trzy zapisy z errat, dotyczące przetworników ADC: - F103: Voltage glitch on ADC input 0 - F429: Internal noise impacting the ADC accuracy - F429: ADC sequencer modification during conversion A lepiej by było, gdyby było: Dodatkowo chciałbym jeszcze zwrócić uwagę na kilka zapisów z errat, dotyczących przetworników ADC: - F103: Voltage glitch on ADC input 0 - F334: DMA Overrun in dual interleaved mode with single DMA channel - F334: Sampling time shortened in JAUTO autodelayed mode - F334: Load multiple not supported by ADC interface - F334: Possible voltage drop caused by a transitory phase when the ADC is switching from a regular channel to an injected channel Rank 1 - F429: Internal noise impacting the ADC accuracy - F429: ADC sequencer modification during conversion

		Jest: Omawiane mikrokontrolery STM32 posiadają po dwa przetworniki DAC. Każdy przetwornik ma tylko jeden kanał wyjściowy. W obu mikrokontrolerach: - DAC_OUT1 to nóżka PA4 - DAC_OUT2 to nóżka PA5
40.	rozdział 14.1, trzeci akapit	A lepiej by było, gdyby było: Mikrokontrolery STM32 F103 i F429 posiadają po dwa przetworniki DAC. Każdy przetwornik ma tylko jeden kanał wyjściowy. W obu mikrokontrolerach: - DAC_OUT1 to nóżka PA4 - DAC_OUT2 to nóżka PA5
		<i>Pan indywidualista (F334) też ma dwa przetworniki, ale pierwszy z nich ma dwa kanały wyjściowe:</i> - DAC1_OUT1 - nóżka PA4 - DAC1_OUT2 - nóżka PA5 - DAC2_OUT1 - nóżka PA6
41.	rozdział 14.1, akapit przed tabelą 14.1	Jest: W razie potrzeby należy stosować zewnętrzne układy wzmacniające sygnał. Sytuację odrobinę poprawia wbudowany w mikrokontroler układ buforowania sygnału wyjściowego (można go wyłączyć i wyłączyć programowo). A lepiej by było, gdyby było (dodany przypis dolny): W razie potrzeby należy stosować zewnętrzne układy wzmacniające sygnał. Sytuację odrobinę poprawia wbudowany w mikrokontroler układ buforowania sygnału wyjściowego ^x (można go wyłączyć i wyłączyć programowo).
		^x w przypadku mikrokontrolera F334 bufor jest dostępny tylko na wyjściu DAC1_OUT1
42.	rozdział 14.1, tabela 14.1, wartość t _{settling max}	Jest: F103: 4μs F429: 6μs A lepiej by było, gdyby było: F103: 4μs F334: 4μs F429: 6μs
43.	rozdział 14.4	Dodany nowy rozdział: 14.4. Odrobina odmiany (F334)
44.	rozdział 15.1, tytuł rozdziału	Jest: 15.1. STM32F103 A lepiej by było, gdyby było: 15.1. USART (F103)
45.	rozdział 15.2, tytuł rozdziału	Jest: 15.2. STM32F429 A lepiej by było, gdyby było: 15.2. USART (F429)
46.	rozdział 15.3	Dodany nowy rozdział: 15.3. USART (F334)
47.	rozdział 16.1, po opisie pamięci mikrokontrolera F429	Jest: - nie ma - A lepiej by było, gdyby było (nowy akapit):

		Zaś w F334 sprawy mają się następująco: <ul style="list-style-type: none"> - Main Memory - pamięć programu i danych - 0x0800 0000 - 0x0800 FFFF - Information Block - tutaj znajdują się: <ul style="list-style-type: none"> - System Memory (0x1FFF D800 - 0x1FFF F7FF) - firmowy bootloader - Option Bytes (0x1FFF F800 - 0x1FFF F80F) - bajty konfiguracyjne
48.	rozdział 16.1, opis opóźnień kontrolera pamięci flash	Jest: dla STM32F103 wielkość opóźnień jest związana z częstotliwością zegara systemowego SYSCLK A lepiej by było, gdyby było: dla STM32F103 i STM32F334 wielkość opóźnień jest związana z częstotliwością zegara systemowego SYSCLK
49.	rozdział 16.1, tabela 16.1, tytuł tabeli	Jest: Tabela 16.1 Wymagane opóźnienia przy dostępie do pamięci flash w STM32F103 A lepiej by było, gdyby było: Tabela 16.1 Wymagane opóźnienia przy dostępie do pamięci flash w F103 i F334
50.	rozdział 16.1, ostatni akapit	Jest: Czas kasowania pamięci flash, może dochodzić do 40ms w przypadku F103 i ponad 30s (!) w przypadku F429 A lepiej by było, gdyby było: Czas kasowania pamięci flash, może dochodzić do 40ms w przypadku F103/F334 i ponad 30s (!) w przypadku F429
51.	rozdział 16.2, drugi akapit	Jest: Działa to w ten sposób, że tuż po resecie lub wybudzeniu z trybu standby (dokładniej 4 cykle zegarowe po) sprawdzany jest stan nóżek mikrokontrolera oznaczonych BOOT0 i BOOT1. A lepiej by było, gdyby było (dodany przypis dolny): Działa to w ten sposób, że tuż po resecie lub wybudzeniu z trybu standby (dokładniej 4 cykle zegarowe po) sprawdzany jest stan nóżek mikrokontrolera oznaczonych BOOT0 i BOOT1 ^x . ^x F334 nie posiada nóżki BOOT1, zamiast niej odczytywany jest stan bitu konfiguracyjnego nBOOT1
52.	rozdział 16.2, tabela 16.3	Dodano: osobną kolumnę dla mikrokontrolera F334
53.	rozdział 16.3, drugi akapit pod tabelą 16.3	Jest: Jeśli konfiguracja nóżek będzie inna, np. BOOT0 = 1 i BOOT1 = 0, to odczytując pamięć od adresów 0 procesor będzie dostawał zawartość pamięci System Memory A lepiej by było, gdyby było: Jeśli konfiguracja nóżek będzie inna, np. BOOT0 = 1 i BOOT1 = 0 (dotyczy F103 i F429), to odczytując pamięć od adresów 0 procesor będzie dostawał zawartość pamięci System Memory
54.	rozdział 16.3, wykropkowanie pod tabelą 16.3	Jest: <ul style="list-style-type: none"> - w F103: USART1 - w F429: USART1, USART3, CAN2 i USB (klasa DFU) A lepiej by było, gdyby było: <ul style="list-style-type: none"> - w F103: USART1 - w F334: USART1, USART2, I2C1 - w F429: USART1, USART3, CAN2 i USB (klasa DFU)
55.	rozdział 16.5	Dodany nowy rozdział: 16.5. Bajty konfiguracyjne (F334)

56.	rozdział 16.6 (16.5 przed zmianą 54), tytuł rozdziału	Jest: 16.6. Kod w pamięci SRAM - po co? (F103 i F429) A lepiej by było, gdyby było: 16.6. Kod w pamięci SRAM - po co?
57.	rozdział 16.7 (16.6 przed zmianą 54), tytuł rozdziału	Jest: 16.7. Funkcja w pamięci sram - jak? (F103 i F429) A lepiej by było, gdyby było: 16.7. Funkcja w pamięci sram - jak?
58.	rozdział 16.8 (16.7 przed zmianą 54), tytuł rozdziału	Jest: 16.8. Cały program w pamięci sram - jak? (F103 i F429) A lepiej by było, gdyby było: 16.8. Cały program w pamięci sram - jak?
59.	rozdział 16.8 (16.7 przed zmianą 54), trzeci akapit pod listingiem „Program uruchamiający program z pamięci sram”	Jest: Został jeszcze jeden, finalny, punkt programu - zmuszenie mikrokontrolera, żeby sprzętowo uruchomił się z pamięci sram. Tak jak opisywałem w rozdziale 16.2, za pomocą nóżek BOOTx, można zmienić mapowanie pamięci tak, aby od adresu 0 dostępna była pamięć sram. A lepiej by było, gdyby było (dodany przypis dolny): Został jeszcze jeden, finalny, punkt programu - zmuszenie mikrokontrolera, żeby sprzętowo uruchomił się z pamięci sram. Tak jak opisywałem w rozdziale 16.2, za pomocą nóżek BOOTx ^x , można zmienić mapowanie pamięci tak, aby od adresu 0 dostępna była pamięć sram. ^x lub bitu konfiguracyjnego nBOOT1 w przypadku mikrokontrolera F334
60.	rozdział 16.8 (16.7 przed zmianą 54), na końcu, przed posumowaniem	Dodano: nowy akapit poświęcony mikrokontrolerowi F334. (w skrócie: nic nowego)
61.	rozdział 16.8 (16.7 przed zmianą 54), podsumowanie	Jest: - można wymusić sprzętowo (nóżki BOOT0 i BOOT1 w stanie wysokim) bootowanie z pamięci sram: - bootowanie z pamięci sram w F429 jest cacy - nie trzeba się martwić o stos i przerwania (przy czym, nie wiadomo po co, RM zaleca jednak ustawienie SCB_VTOR) A lepiej by było, gdyby było: - można wymusić sprzętowo (nóżki BOOT0 i BOOT1 w stanie wysokim lub bit nBOOT1 w stanie niskim w przypadku F334) bootowanie z pamięci sram: - bootowanie z pamięci sram w F429 i F334 jest cacy - nie trzeba się martwić o stos i przerwania (przy czym, nie wiadomo po co, RM zaleca jednak ustawienie SCB_VTOR)
62.	rozdział 17.4	Dodany nowy rozdział: 17.4. System zegarowy (F334)
63.	rozdział 18.1, pierwszy akapit	Jest: W mikrokontrolerze F103, z wymienionych, występuje jedynie blok CRC. Dodatkowo działa on identycznie jak w F429. Z tego względu w tym rozdziale spuścimy na F103 zasłonę milczenia i skupimy się na bogatszym F429. A lepiej by było, gdyby było: W mikrokontrolerze F103, z wymienionych, występuje jedynie blok CRC. Dodatkowo działa on identycznie jak w F429. Z tego względu w tym rozdziale spuścimy na F103 zasłonę milczenia i skupimy się na bogatszym F429. No i na F334, gdyż w nim wszystko musi być inaczej.

64.	rozdział 18.5	Dodany nowy rozdział: <i>18.5. Suma kontrolna CRC (F334)</i>
65.	rozdział 19.2, dygresja wydzielona liniami	Jest: <i>Mikrokontrolerów F103 i F429 „problem” nie dotyczy. Koniec OT!</i> A lepiej by było, gdyby było: <i>Mikrokontrolerów F103 i F429 „problem” nie dotyczy, ale już niedługo się nań nadziejmy w rozdziale 19.6. Koniec OT!</i>
66.	rozdział 19.6	Dodany nowy rozdział: <i>19.6. Outsider (F334)</i>
67.	dodatek 7	Dodany nowy rozdział: <i>Dodatek 7: Gwałt na Nucleo w dwóch aktach</i>

21.9. Zmiany dokonane w wersji 1.8 Poradnika

Tabela 21.9. Zmiany dokonane w wersji 1.8 Poradnika

lp.	Dotyczy	Opis
1.	rozdział 3.1, rysunek 3.1	<i>Poprawiono rysunek. Zamiast dwóch tranzystorów typu N jest para komplementarna.</i>
2.	rozdział 3.1	Dodany nowy akapit na końcu: <i>Przy konfigurowaniu pinów warto zwrócić uwagę na to, aby nie zmienić konfiguracji (domyślnej) wyprowadzeń związanych z wykorzystywanyym interfejsem programowania/debugowania (np. JTAG albo SWD). Wgranie programu, który zmienia konfigurację tych wyprowadzeń może uniemożliwić połączenie się z mikrokontrolerem. Bez paniki! Nic się w ten sposób nie zablokuje. W razie czego wystarczy przytrzymać reset podczas łączenia z mikrokontrolerem albo uruchomić go w trybie bootloadera (więcej na ten temat w rozdziale 16.2). Upozdrawiam, żeby potem nie było niespodzianki :)</i>
3.	rozdział 3.1, podpis rysunku 3.1	Jest: <i>Rys. 3.1 Wyjścia push-pull i open-drain (paintCAD)</i> A lepiej by było, gdyby było (dodany przypis dolny): <i>Rys. 3.1 Wyjścia push-pull i open-drain^x (paintCAD)</i> ^x w rzeczywistości są to tranzystory polowe, na rysunku użyto symboli bipolarnych dla zwiększenia czytelności
4.	rozdział 3.4, tabela 3.3, rejestr OSPEEDR	Jest: OSPEEDR <i>00 – Low Speed 01 – Medium Speed 10 – Fast Speed 11 – High Speed</i> A lepiej by było, gdyby było (m.in. dodany przypis dolny): OSPEEDR^x <i>00 – Low Speed 01 – Medium Speed 10 – High Speed (Fast Speed) 11 – Very High Speed (High Speed)</i> ^x w nawiasach podano nazwy „obowiązujące” do (chyba) 10 wersji Reference Manuala, potem nazwy zostały zmienione. I dobrze, zawsze my się myliło czy High Speed jest szybsze od Fast Speed czy odwrotnie :)
5.	rozdział 3.4, akapit przed tabelą 3.4	Jest: <i>Otoż w F334 jest mniej prędkości niż w F429. Ubyło nam fast speed. Szczegóły zebrane w poniższej tabelce:</i> A lepiej by było, gdyby było: <i>Otoż w F334 jest mniej prędkości niż w F429. Ubyło nam jedno high speed. Szczegóły zebrane w poniższej tabelce:</i>
6.	rozdział 3.4, tabela 3.4, kolumna F429	Jest: F429 <i>Low Speed Medium Speed Fast Speed High Speed</i> A lepiej by było, gdyby było:

		F429 <i>Low Speed</i> <i>Medium Speed</i> <i>High Speed</i> <i>Very High Speed</i>
7.	rozdział 5.6, zadanie 5.1, linijka 46 listingu	Jest: $BB(TIM3->SR, TIM_SR_UIF) = 0;$ A lepiej by było, gdyby było: $TIM3->SR = \sim TIM_SR_UIF;$
8.	rozdział 5.6, zadanie 5.1, linijka 53 listingu	Jest: $BB(EXTI->PR, EXTI_PR_PR13) = 1;$ A lepiej by było, gdyby było: $EXTI->PR = EXTI_PR_PR13;$
9.	rozdział 7.3, rozwiązywanie zadania 7.4, dwa ostatnie akapity	Jest (pomyłona nazwa bitu): $\dots MR13 \dots$ A lepiej by było, gdyby było: $\dots PR13 \dots$
10.	rozdział 8.5, rozwiązywanie zadania 8.4, linijka 27 listingu	Jest: $TIM1->SR = (uint16_t)\sim TIM_SR_UIF;$ A lepiej by było, gdyby było: $TIM1->SR = \sim TIM_SR_UIF;$
11.	rozdział 8.7, rozwiązywania zadania 8.7, 26 linijka listingu	Jest: $TIM1->SR = (uint16_t)\sim TIM_SR_UIF;$ A lepiej by było, gdyby było: $TIM1->SR = \sim TIM_SR_UIF;$
12.	rozdział 8.11, zadanie 8.9, linijka 43 listingu	Jest: $BB(TIM1->SR, TIM_SR_CC1IF) = 0;$ A lepiej by było, gdyby było: $TIM1->SR = \sim TIM_SR_CC1IF;$
13.	rozdział 8.11, zadanie 8.9, linijka 48 listingu	Jest: $BB(TIM1->SR, TIM_SR_CC2IF) = 0;$ A lepiej by było, gdyby było: $TIM1->SR = \sim TIM_SR_CC2IF;$
14.	rozdział 8.19, rozwiązywania zadania 8.16, 28 linijka listingu	Jest: $TIM1->SR = (uint16_t)\sim TIM_SR_UIF;$ A lepiej by było, gdyby było: $TIM1->SR = \sim TIM_SR_UIF;$
15.	rozdział 11.1, podsumowanie	Nowa kropka: każde źródło resetu systemowego, powoduje ściągnięcie nóżki NRST do masy
16.	rozdział 11.6, rozwiązywanie zadania 11.1, 26 linijka listingu	Jest: $TIM1->SR = (uint16_t)\sim TIM_SR_UIF;$ A lepiej by było, gdyby było: $TIM1->SR = \sim TIM_SR_UIF;$
17.	rozdział 13.2, zadanie 13.4, linijka 44 listingu	Jest: $BB(ADC1->SR, ADC_SR_JEOC) = 0;$

		A lepiej by było, gdyby było: $ADC1->SR = (\sim ADC_SR_JEOC) \& 0x1f;$
18.	rozdział 13.5, zadanie 13.6, linijka 28 listingu	Jest: $BB(ADC1->SR, ADC_SR_AWD) = 0;$ A lepiej by było, gdyby było: $ADC1->SR = (\sim ADC_SR_AWD) \& 0x1f;$
19.	rozdział 17.3, rozwiązywanie zadania 17.3, linijka 3 listingu	Jest: $RCC->CR = RCC_CR_HSEON;$ A lepiej by było, gdyby było: $RCC->CR = RCC_CR_HSEON RCC_CR_HSEBYP;$
20.	rozdział 17.3, rozwiązywanie zadania 17.3, opis linii 3	Jest: 3) włączenie HSE A lepiej by było, gdyby było: 3) włączenie HSE. Uwaga! W domyślnej konfiguracji płytki Discovery, do głównego mikrokontrolera, doprowadzony jest sygnał zegarowy z programatora ST-Link. Mikrokontroler pracujący w ST-Linku ma włączone „wyprowadzenie” sygnału zegarowego (8MHz) na zewnątrz poprzez wyjście MCO (Master Clock Output). Sygnał ten przez rezystor R28 i mostek SB18 jest doprowadzony do wejścia HSE układu F429. Czyli, żeby nie było wątpliwości: oscylator HSE nie współpracuje z zamontowanym na płytce kwarcem (X3), tylko dostaje „gotowy” sygnał zegarowy z ST-Linka! W takiej sytuacji należy włączyć bit HSEBYP.
21.	rozdział 19.3, zadanie 19.3, linijka 63 listingu	Jest: $BB(EXTI->PR, EXTI_PR_PR12) = 1;$ A lepiej by było, gdyby było: $EXTI->PR = EXTI_PR_PR12;$
22.	rozdział 19.4, zadanie 19.4, linijka 40 listingu (program dla slave'a wersja podstawowa)	Jest: $BB(EXTI->PR, EXTI_PR_PR4) = 1;$ A lepiej by było, gdyby było: $EXTI->PR = EXTI_PR_PR4;$
23.	rozdział 19.4, zadanie 19.4, linijka 42 listingu (program dla slave'a wersja rozszerzona)	Jest: $BB(EXTI->PR, EXTI_PR_PR4) = 1;$ A lepiej by było, gdyby było: $EXTI->PR = EXTI_PR_PR4;$
24.	rozdział 20	Dodany nowy rozdział (z dwoma podrozdziałami): 20. Komparator i wzmacniacz („Nec temere, nec timide”) 20.1. Komparator analogowy (F334) 20.2. Wzmacniacz operacyjny (F334)
25.	dodatek 2, po listingu funkcji	Dodany nowy akapit: Uwaga! Tak jak wspomniano w rozdziale 3.4, od (chyba) 10 wersji dokumentacji (RM) nastąpiła zmiana w nazewnictwie „prędkości” wyjść. Z racji uwagi jednak na lenistwo własne moje, nie chce mi się zmieniać definicji wykorzystywanych przez omawianą funkcję. W związku z powyższym, pozostawiam „stare” określenia prędkości: LS - Low Speed MS - Medium Speed FS - Fast/Full Speed (w „nowym” RMie ta nazwa została zmieniona na: High Speed) HS - High Speed (w „nowym” RMie ta nazwa została zmieniona na: Very High Speed)