

Mikrokontrolery STM32

Układy peryferyjne i nie tylko

(poradnik dla początkujących)

autor: szczywronek (szczywronek@gmail.com)

wersja: 1

ostatnia modyfikacja: 06.11.2015

Lódź, 31.07.2015

I n w e n t a r z

1. Wstęp formalny	6
1.1. Co to właściwie jest? („Quo vadis Domine?”)	6
1.2. Ja („Quicquid Latine dictum sit, altum videtur”)	7
1.3. Ty („Bene tibi!”)	7
1.4. Uwagi końcowe („Cogitationis poenam nemo patitur”)	9
2. Wstęp techniczny	10
2.1. Od przybytku głowa (nie) boli („Divitiae non sunt bonum”)	10
2.2. O co chodzi z tym rdzeniem? („Nil mirari, nil indignari, sed intelligere”)	12
2.3. Hardware („Nemo sine vitiis est!”)	15
2.4. Piśmiennictwo („Littera docet, littera nocet.”)	17
2.5. Software („Ex malis eligere minima oportet”)	22
2.6. Słowo o bibliotekach („Relata refero”)	23
2.7. Wskazówki przed startowe („Ave, Caesar, morituri te salutant”)	27
3. Porty I/O („Ardua prima via est”)	32
3.1. Ogarnąć nóżki w F103	32
3.2. Nieśmiertelnie Blinkający Hell of World (F103)	36
3.3. Atomowo macham nogą i nie tylko (F103 i F429)	39
3.4. Cortex-M4 wybieram Cię! (F429)	43
3.5. Wybór funkcji alternatywnej (F429)	48
3.6. Remapping funkcji alternatywnych (F103)	51
3.7. Elektryczna strona medalu (F103 i F429)	52
3.8. Skompensuj mi celę (F429)	54
4. Bit banding („Gemma gemmarum”)	55
4.1. Ale o co chodzi?	55
4.2. Jak to działa?	56
4.3. Jasne i proste jak cała matematyka	57
4.4. Makro do bit bandingu	58
4.5. Kiedy stosować bit banding	61
4.6. A gdzie jest haczyk?	62
4.7. Bit banding w STM32F429 (Cortex-M4)	62
5. Przerwania i wyjątki („Macte animo, iuvenis!”)	63
5.1. Trochę zbędnej teorii o trybach pracy rdzenia	63
5.2. Poznajmy wyjątki i przerwania w Cortexie	64
5.3. Mechanika działania wyjątków	67

5.4. Priorytety i wywłaszczanie	69
5.5. Funkcje pomocnicze	71
5.6. Priorytety przykład praktyczny	74
6. Licznik systemowy SysTick („Magnum opus”)	78
6.1. Blink me baby one more time (F103, F429)	78
7. Przerwania zewnętrzne („Facta sunt verbis difficilora”)	82
7.1. EXTI (F103)	82
7.2. EXTI (F429)	88
8. Liczniki („Festina lente!”)	90
8.1. Wstęp	90
8.2. Blok zliczający, prosty timer	91
8.3. Update Event (UEV), buforowanie rejestrów i One Pulse Mode	93
8.4. Schemat blokowy licznika	94
8.5. Licznik pędzony z zewnątrz	96
8.6. Filtrowanie sygnałów zewnętrznych	100
8.7. Tryb enkodera	102
8.8. Taktowanie licznika innym licznikiem	104
8.9. Podsumowanie źródeł taktowania	105
8.10. Blok porównujący - PWM	106
8.11. Blok porównujący - przerwania	113
8.12. Blok porównujący - podsumowanie	115
8.13. Blok przechwytyjący - Input Capture	116
8.14. Synchronizacja sygnałem zewnętrznym	118
8.15. PWM Input Mode	121
8.16. Synchronizacja kilku liczników	124
8.17. Liczniki ogólnego przeznaczenia i podstawowe	126
8.18. Luźne uwagi na koniec	127
8.19. Różnice między F103 i F429	129
9. Battery Backup Domain (“Non omnis moriar”)	132
9.1. Wstęp	132
9.2. Backup Registers (F103)	132
9.3. RTC (F103)	136
9.4. Backup Registers (F429)	139
9.5. Backup SRAM (F429)	142
9.6. RTC (F429)	144

10. Układy watchdog („Duo cum faciunt idem, non est idem”)	149
10.1. Watchdog niezależny IWDG	149
10.2. Watchdog okienkowy WWDG	153
10.3. Porównanie układów watchdog	159
11. Reset, zasilanie i tryby oszczędzania energii („Difficilis in otio quies”)	160
11.1. Reset	160
11.2. Zasilanie (F103)	162
11.3. Zasilanie (F429)	162
11.4. Debugowanie a tryby uśpienia (F103 i F429)	163
11.5. Tryby obniżonego poboru mocy (F103)	164
11.6. Tryby obniżonego poboru mocy (F429)	169
12. Mechanizm DMA („Annuntio vobis gaudium magnum: habemus DMA”)	177
12.1. Z czym to się je?	177
12.2. DMA (F103)	178
12.3. DMA (F429)	185
13. Przetwornik ADC („Superflua non nocent”)	195
13.1. ADC wstęp (F103)	195
13.2. Tryby pracy pojedynczego przetwornika ADC (F103)	198
13.3. Czas próbkowania (F103)	210
13.4. Tryby pracy dwóch przetworników ADC (F103)	213
13.5. Bajery (F103)	219
13.6. Ogólne spojrzenie na tryby pracy przetwornika ADC (F103 i F429)	224
13.7. Różnice w STM32F429 (F429)	225
13.8. Końcowe uwagi (F103 i F429)	240
14. Przetwornik DAC („Omne ignotum pro magnifico”)	242
14.1. Wstęp (parametry i tryby pracy)	242
14.2. Zadania praktyczne (F103)	247
14.3. Zadania praktyczne (F429)	255
14.4. Uwagi końcowe	256
15. Interfejs USART („Volenti nihil difficile”)	257
15.1. STM32F103	257
15.2. STM32F429	262
16. Pamięć Flash („Variatio delectat”)	265
16.1. Pamięć Flash	265
16.2. Tryby uruchamiania i bootloader	267

16.3. Bajty konfiguracyjne (F103)	269
16.4. Bajty konfiguracyjne (F429)	274
17. System zegarowy („Finita est comoedia”)	276
17.1. Wstęp	276
17.2. System zegarowy (F103)	278
17.3. System zegarowy (F429)	286
Dodatek 1: Funkcja konfiguruająca porty (F103)	291
Dodatek 2: Funkcja konfiguruująca porty (F429)	294
Dodatek 3: Makro do bit bandingu	300
Dodatek 4: To bit band or not to bit band	304
Dodatek 5: Atrybut interrupt (F103, GCC)	307
Dodatek 6: Przerwanie widmo	309
18. Changelog („Hominis est errare, insipientis in errore perseverare”)	311

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 FOAD³...

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.”

3 STFW

1.2. Ja („Quicquid Latine dictum sit, altum videtur”⁴)

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*⁵ czy *zakapturzenie*⁶. Jak się nie podoba to wiadomo... ja nikogo nie zmuszam.

Początki na pewno nie są proste - tyle nowości... Dotychczas bawiłem się tylko mikrokontrolerami AVR 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 tygodni⁷ „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 100zł⁸ (po spaleniu Atmegi cierpiała głównie duma, nie kieszeń). Wierzę jednak, że uda mi się przebrnąć przez ten galimatias.

1.3. Ty („Bene tibi!”⁹)

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 AVR, 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*?
- Co to jest rejestr mikrokontrolera, co to jest przerwanie?

4 „Cokolwiek powiesz po lacinie, brzmi mądrze.”

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

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

8 oczywiście nie trzeba kupować drogich płytaków. Za 15zł można mieć płytkę z mikrokontrolerem STM32F103 i darmową dostawą do domu (ebay)

9 „Dobrze Tobie! (pozdrawienie)”

- 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 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 :)

¹⁰ pytanie dotyczy tylko osób korzystających z tego IDE :)

¹¹ i nie wybuchało... od razu

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

W każdym rozdziale opisującym jakieś peryferium, zamieściłem „zadania domowe” z przykładowymi rozwiązaniami. Przyłoż 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, prześpij 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 STMachine 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 jestem egoistą i zbieram na pendrive'a). I tyle w temacie. W razie jakichś wątpliwości proszę śmiało pisać na podany adres mailowy.

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

14 gdybym nie był bezrobotny to nie miałbym czasu na pisanie takich elaboratów :)

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 STMicroelectronics¹⁷. 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 go produkuje

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 zmennoprzecinkowych)
- *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
- rejstry 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) 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:

- *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)

i na nich będę się dalej opierał. Dokładniej STM32F103 będzie (najczęściej) punktem wyjścia przy poznawaniu jakiegoś peryferium. A jeśli w STM32F429 wygląda ono inaczej, to zostaną omówione różnice. 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 płyta 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.

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.

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

2.4. Piśmiennictwo („*Littera docet, littera nocet.*”²⁹)

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 pomysłom, bazować na dokumentacji i odczepić 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 tajemniczych. Więc skąd?

Za darmo z Internetu³⁰ :) Przede wszystkim z najpewniejszego źródła – czyli dokumentacji producenta. I tu mała, ale bardzo ważna uwaga (skupić się proszę). Atmel przyzwyczaił Nas do dokumentacji jedno-pedestalowej. Tzn. wystarczyło ściągnąć notę konkretnego mikrokontrolera i było w niej wszystko. Totalnie wszystko. Informacje o rdzeniu, opisy instrukcji assembla, 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 spieszymy, więc jeśli nie czujesz to podumaj nad tym chwilę – jak teraz załapiesz to potem nie będziesz błędzić 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 dokumencie³¹. 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?

29 „*Littera uczy, littera szkodzi.*”

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

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

Rdzeń jest produkowany przez holding ARM³², stąd też i ta firma przygotowuje 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 ma³³ :)

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.

32 www.arm.com; ARM to jednocześnie nazwa producenta i architektury

33 przykład z życia wzięty

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 diametralne³⁴.

Krótki poradnik jak szukać dokumentów na stronie ST:

1. www.st.com
 2. z górnej belki wybieramy *Products → Microcontrollers*
 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 zacznesz cokolwiek robić z STMem zobacz w datasheetcie co Twój mikrokontroler oferuje! Przykład: *Reference Manual RM0008* (kodeks drogowy) dotyczy mikrokontrolerów:

³⁴ jak np. opis kalibracji ADC: <http://www.elektroda.pl/rtvforum/viewtopic.php?p=13790376#13790376>
(wątek: [STM32] Kalibracja ADC. Jak często?)

- *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 sięagniesz 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 kolei 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 pdf'y 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*”³⁵)

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 rodzaju, więc w nim tego nie będzie³⁶. Dlatego, że to nie ma sensu. Dokładny opis środowiska, konfiguracji i obsługi to byłoby kilkudziesiąt/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?*)

35 „Ze zlego należy wybierać mniejsze zło.”

36 tak - można lamentować

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 Linuksie³⁷
- 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 refero*”³⁸)

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

Temat bibliotek jest dosyć kontrowersyjny. O co chodzi? Sprawa wygląda następująco: firma ARM stworzyła standard *CMSIS*³⁹... 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 układow 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:

³⁷ i w ogóle działa :)

³⁸ „*Powtarzam, co usłyszałem.*”

³⁹ *Cortex Microcontroller Software Interface Standard*

- 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)*. 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)⁴⁰:

```

1. 1.  FSMC_NORSRAMInitTypeDef  FSMC_NORSRAMInitStructure;
2. 2.  /* FSMC */ 
3. 3.  FSMC_NORSRAMTimingInitTypeDef  FSMC_NORSRAMTimingInitStructure;
4. 4.  FSMC_NORSRAMTimingInitStructure.FSMC_AddressSetupTime = 10;
5. 5.  FSMC_NORSRAMTimingInitStructure.FSMC_AddressHoldTime = 0;
6. 6.  FSMC_NORSRAMTimingInitStructure.FSMC_DataSetupTime = 10;
7. 7.  FSMC_NORSRAMTimingInitStructure.FSMC_BusTurnAroundDuration = 0x00;
8. 8.  FSMC_NORSRAMTimingInitStructure.FSMC_CLKDivision = 0x00;
9. 9.  FSMC_NORSRAMTimingInitStructure.FSMC_DataLatency = 0x00;
10. 10. FSMC_NORSRAMTimingInitStructure.FSMC_AccessMode = FSMC_AccessMode_A;
11. 11. FSMC_NORSRAMInitStructure.FSMC_Bank = FSMC_Bank1_NORSRAM1;
12. 12. FSMC_NORSRAMInitStructure.FSMC_DataAddressMux = FSMC_DataAddressMux_Disable;
13. 13. FSMC_NORSRAMInitStructure.FSMC_MemoryType = FSMC_MemoryType_SRAM;
14. 14. FSMC_NORSRAMInitStructure.FSMC_MemoryDataWidth = FSMC_MemoryDataWidth_16b;
15. 15. FSMC_NORSRAMInitStructure.FSMC_BurstAccessMode = FSMC_BurstAccessMode_Disable;
16. 16. FSMC_NORSRAMInitStructure.FSMC_WaitSignalPolarity = FSMC_WaitSignalPolarity_Low;
17. 17. FSMC_NORSRAMInitStructure.FSMC_WrapMode = FSMC_WrapMode_Disable;
18. 18. FSMC_NORSRAMInitStructure.FSMC_WaitSignalActive = FSMC_WaitSignalActive_BeforeWaitState;
19. 19. FSMC_NORSRAMInitStructure.FSMC_WriteOperation = FSMC_WriteOperation_Enable;
20. 20. FSMC_NORSRAMInitStructure.FSMC_WaitSignal = FSMC_WaitSignal_Disable;
21. 21. FSMC_NORSRAMInitStructure.FSMC_AsynchronousWait = FSMC_AsynchronousWait_Disable;
22. 22. FSMC_NORSRAMInitStructure.FSMC_ExtendedMode = FSMC_ExtendedMode_Disable;
23. 23. FSMC_NORSRAMInitStructure.FSMC_WriteBurst = FSMC_WriteBurst_Disable;
24. 24. FSMC_NORSRAMInitStructure.FSMC_ReadWriteTimingStruct = &FSMC_NORSRAMTimingInitStructure;
25. 25. FSMC_NORSRAMInit(&FSMC_NORSRAMInitStructure);
26. 26. /* FSMC */
27. 27. FSMC_NORSRAMTimingInitStructure.FSMC_AddressSetupTime = 3;
28. 28. FSMC_NORSRAMTimingInitStructure.FSMC_AddressHoldTime = 0;
29. 29. FSMC_NORSRAMTimingInitStructure.FSMC_DataSetupTime = 3;
30. 30. FSMC_NORSRAMTimingInitStructure.FSMC_BusTurnAroundDuration = 0x00;
31. 31. FSMC_NORSRAMTimingInitStructure.FSMC_CLKDivision = 0x00;
32. 32. FSMC_NORSRAMTimingInitStructure.FSMC_DataLatency = 0x00;
33. 33. FSMC_NORSRAMTimingInitStructure.FSMC_AccessMode = FSMC_AccessMode_A; /* FSMC */
34. 34. FSMC_NORSRAMInitStructure.FSMC_WriteTimingStruct = &FSMC_NORSRAMTimingInitStructure;
35. 35. FSMC_NORSRAMInit(&FSMC_NORSRAMInitStructure);
36. 36.
37. 37. /* Enable FSMC Bank1_SRAM Bank */
38. 38. FSMC_NORSRAMCmd(FSMC_Bank1_NORSRAM1, ENABLE);

```

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

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

```

1. void FSMC_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_ASYNWAIT(FSMC_NORSRAMInitStruct->FSMC_AynchronousWait));
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_WAIT_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_AynchronousWait
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->BTCR[FSMC_NORSRAMInitStruct->FSMC_Bank] |= (uint32_t) BCR_FACCEEN_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 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-
55. >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_Bank1E->BWTR[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->BTCR[0] = FSMC_BCR1_MWID_0 | FSMC_BCR1_WREN;
3.
4. /* Timing Register, BTR1 */
5. FSMC_Bank1->BTCR[1] = 0x0a | (0x0a<<8);
6.
7. /* Write timing register, BWTR1 */
8. FSMC_Bank1E->BWTR[0] = 0x0fffffff;
9.
10. /* Enable FSMC Bank1_SRAM Bank */
11. FSMC_Bank1->BTCR[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 rejestrów⁴¹.

Jak i skąd to wszystko zdobyć? Plik nagłówkowy mikrokontrolera oczywiście ze strony ST. Dokładniej trzeba pobrać paczkę z biblioteką SPL lub czymś podobnym 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 → STM32 Standard Peripheral Libraries*
- *STM32F10X Standard Peripheral Library*
- *Get Software → Download*
- w rozpakowanym pliku: *Libraries → CMSIS → CM3 → DeviceSupport → ST → STM32F10x*
- i mamy nasz garniec złota na krańcu tęczy – plik nagłówkowy mikrokontrolera z definicjami rejestrów i bitów: *stm32f10x.h*

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 posiadanego mikrokontrolera

Pozostało jeszcze zdobycie kawałka od ARMa. Najnowszą wersję oczywiście pobieramy ze strony producenta (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*

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

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_cmx.h*⁴² – zawiera definicje nazw rejestrów i bitów konfiguracyjnych peryferiów rdzenia (SysTick, NVIC, etc..) oraz kilka bardzo niskopoziomowych funkcji związanych z obsługą np. kontrolera przerwań (NVIC)
- *core_cmFunc.h* – funkcje operujące na rejestrach specjalnych rdzenia, np. umożliwiające zmianę wskaźnika stosu, priorytetów przerwań, itd...
- *core_cmInstr.h* – funkcje *intrinsic* czyli proste wstawki asm napisane w C umożliwiające wywołanie konkretnych rozkazów asm, np. *_WFI()*, *_NOP()*, itd...

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*
 - *core_cmFunc.h*
 - *core_cmInstr.h*
 - oraz plik nagłówkowy mikrokontrolera *stm32fxx.h*

2.7. Wskazówki przed startowe („*Ave, Caesar, morituri te salutant*”⁴³)

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:

42 gdzie *x* to numer rdzenia

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

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

na przykład jeśli chcielibyśmy włączyć przetwornik ADC to będzie to mniej więcej coś takiego⁴⁴:

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

W STM konwencja jest inna (standard CMSIS). Przede wszystkim jest o wiele więcej peryferiów, które mają jeszcze więcej rejestrów konfiguracyjnych, a rejestyry więcej bitów. Zaprowadzono więc porządek i rejestyry 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;  
BLOK_PERYFERYJNY -> REJESTR (operator) BLOK_REJESTR_BIT;
```

Proste prawda? To magiczne „->” to zwyczajny operator języka C (dostęp do składnika struktury poprzez wskaźnik). W Poradniku, chcąc odnieść się do rejestrów *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.

Nazwa bitu składa się z: nazwy bloku peryferyjnego, nazwy rejestrów i właściwej nazwy bitu. Przykładowo *TIM_CR1_CEN*:

- chodzi o bit licznika (*TIM*)
- bit jest w rejestrze *CR1*
- 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:

⁴⁴ pomijam tu kwestię czy powinno to być przypisanie (=) czy suma bitowa (|=) bo nie o to się rozchodzi

- TIM1_CR1_CEN;
- TIM2_CR1_CEN;
- ...

gdyż wszystkie liczniki mają identyczne rejesty konfiguracyjne *CR1* (prawie, ale do wszystkiego dojdziemy w swoim czasie).

Przy nazwie rejestrów (*CR1*) numerek jest zawsze. Każdy licznik ma dwa rejesty konfiguracyjne *CR_* (*CR1 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 *OC1M*, 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 odczytaniu⁴⁵ 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”⁴⁶ (sumę bitową) na pałkę. Notabene w AVR też chyba część rejestrów miała domyślną wartość różną od zera – nie chce mi się szukać...

Quarto! Mikrokontrolery STM32 nie mają *fuse bitów* którymi można by 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 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* :)

45 np. debuggerem

46 idiotyczny termin

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)
- **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 konkrety 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 mikrokontrolerów F103 i F429, informacje dotyczące poszczególnych rodzin czasem się przeplatają. Zdecydowanie należy przeczytać oba, nawet jeśli nie zamierzasz korzystać np. z F429. 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 uważa się 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”⁴⁷)

3.1. Ogarnąć nóżki w 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 STMach 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 (GPIO⁴⁸) 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*

47 „Początki są trudne.”

48 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 sposób⁴⁹: 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”⁵⁰, 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 rejestrów⁵¹) 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.

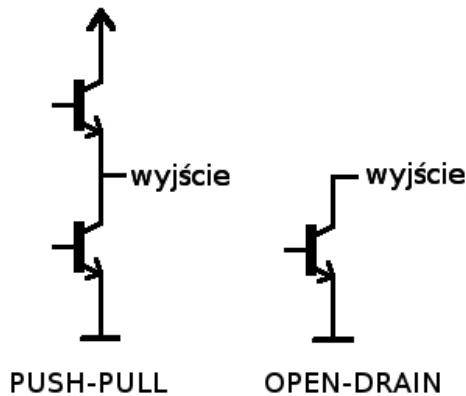
Wyjście – nowości w stosunku do AVRów są dwie. Po pierwsze wyjście może pracować w dwóch trybach: *push-pull* lub *open-drain*. Wyjście *push-pull* to dwa tranzystory (patrz rys. 3.1).

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

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

51 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 mikrokontroler⁵² 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-drain (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”⁵³ 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* kontrolera⁵⁴. Po co to? Zmniejszanie stromości zboczy redukuje zakłócenia indukow...

52 oczywiście w pewnych granicach – patrz *datasheet*

53 strasznie mylące to określenie

54 rozdział *Input/Output AC characteristics*

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 resecie⁵⁵. Możemy je również podciągnąć wewnętrznym 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 resecie są ustawione tak, aby wybrany był tryb wejściowy płynący. Czyli zgodnie z tabelką 3.2 bity CNF mają wartość 0b01. Po naszej operacji (CNF |= 0b10) wyszło więc: 0b01 | 0b10 = 0b11. Nie o to nam chodziło.

⁵⁵ wyjątkiem są m.in. piny JTAGa: PA15 (pull-up), PA14 (pull-down), PA13 (pull-up), PB4 (pull-up)

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 ;)

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 Praxis⁵⁶. No to lecimy na głęboką wodę. Tzn. Ty lecisz :P Czytelnika proszę o zasięście, stworzenie nowego *maina* w Twoim ulubionym środowisku i napisanie kodu do migania diodą :) Pomocy szukamy w *Reference Manualu* (dokumentacji periferii mikrokontrolera) w rozdziale dotyczącym GPIO i RCC⁵⁷ (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 obiecuje, 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 czytałem o STIMach 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 odbierzaj sobie tego, bo to jest bezcenna wiedza której nie da się przekazać żadnym poradnikiem.

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

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

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

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 (HSI⁵⁸) 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 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, rejstry RCC_AHBENR, RCC_APB1ENR, RCC_APB2ENR), np. bit włączający GPIOB nazywa się IOPBEN i leży w rejestrze RCC_APB2ENR

58 High Speed Internal, wszystko się wyjaśni w rozdziale 17

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_ARP2ENR*. Pod definicją bitu kryje się maska bitu włączającego zegar dla portu B (bit *IOPBEN*). Polecam pooglądać sobie co się kryje pod *RCC*, *AP2ENR* 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 resecie procka) był ustawiony (Tertio!).

W pętli głównej, z wykorzystaniem rejestru wyjściowego *GPIO_ODR*, cyklicznie zerowany i ustawiany jest jeden z bitów rejestru (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 pasuje⁵⁹. Ze swojej strony polecam funkcje udostępnione przez *Freddiego Chopina* w przykładach na jego stronie⁶⁰. Swego czasu próbowałem napisać coś po swojemu, jednak z każdą poprawką mojej funkcji coraz bardziej zbliżałem się do wersji *Freddiego*⁶¹ :) Tak czy siak w przykładach będę korzystał z funkcji *Freddiego* albo bardzo zbliżonej. Nie ma sensu jej teraz omawiać, kod źródłowy

59 błagam, tylko nie potwora z biblioteki SPL

60 www.freddiechopin.info

61 wszystkie drogi prowadzą do *Freddiego*

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

3.3. Atomowo macham nogą i nie tylko (F103 i F429)

Na początek krótki wstęp o atomowości. Określenie „operacja atomowa” oznacza operację niepodzielną. Założymy, 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      załaduj do r3 to co jest pod adresem foobar
orr r3, 0b1000       suma logiczna wartości z rejestru 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 rejestru peryferiala (lub pamięci) do „zmiennej pomocniczej” (rejestru ogólnego procesora - np. r3)
- operacji arytmetyczno-logicznej na tej zmiennej pomocniczej
- zapisania wartości zmiennej pomocniczej nazad do rejestru peryferiala (lub pamięci)

Dlaczego tak dziwnie? Dlatego, że operacje arytmetyczno-logiczne mogą być wykonywane tylko na rejestrach ogólnych procesora⁶². 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 rejestru:

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

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

```
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

62 architektura RISC tak ma

zmiennej pomocniczej (0b11) jest wpisywana do rejestru 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 rejestru. 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. STM32 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) odpowiadają 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 ustawienie⁶³ 9-go bitu rejestru 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**, jedynie dopuszczalne działanie to:

```
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 - wrzucimy jedynkę to coś się zadzieje (np.

⁶³ za pomocą jakiegoś sprzętowego *hokus-pokus*

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 rejestru 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. Jedną 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.    /* while(1) */
19. } /* 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)
- rejestr GPIO_BSRR i GPIO_BRR umożliwiają atomowe mechanizmy nóżkami

3.4. Cortex-M4 wybieram Cię! (F429)

Przyjrzyjmy się teraz jak to wygląda w STM32F429. Jest to o wiele bardziej rozbudowany mikrokontroler, 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, to inne rejesty konfiguracyjne portów:

- *MODER* – tryb pracy (wejście, wyjście, funkcja alternatywna, pin analogowy)
- *OTYPER* – typ wyjścia (*push-pull*, *open-drain*)
- *OSPEEDR* - nieszczesna „prędkość” wyjścia
- *PUPDR* – włączanie *pull-upu* lub *pull-downu*
- *IDR* – rejestr wejściowy
- *ODR* – rejestr wyjściowy
- *BSRR* – atomowe kasowanie, ustawianie bitów rejestru *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 góre 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 góre 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	PUPDR	Konfiguracja ⁶⁴	
01	0	00 – Low Speed	00	wyjście	PP
		01 – Medium Speed	01	wyjście	PP + PU
		10 – Fast Speed	10	wyjście	PP + PD
		11 – High Speed			
	1	00 – Low Speed	00	wyjście	OD
		01 – Medium Speed	01	wyjście	OD + PU
		10 – Fast Speed	10	wyjście	OD + PD
		11 – High Speed			
10	0	00 – Low Speed	00	f. alternat.	PP
		01 – Medium Speed	01	f. alternat.	PP + PU
		10 – Fast Speed	10	f. alternat.	PP + PD
		11 – High Speed			
	1	00 – Low Speed	00	f. alternat.	OD
		01 – Medium Speed	01	f. alternat.	OD + PU
		10 – Fast Speed	10	f. alternat.	OD + PD
		11 – High Speed			
00	bez znaczenia		00	wejście	pływające
			01	wejście	PU
			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.

64 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->BSRRH = GPIO_BSRR_BS_14;
15.        } else {
16.            GPIOG->ODR &= ~GPIO_ODR_ODR_13;
17.            GPIOG->BSRRL = GPIO_BSRR_BS_14;
18.        }
19.
20.    } /* while(1) */
21.
22. } /* main */
```

Proste? Dobra. Przyznać się, kto zauważył 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. Po wyłączeniu 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 (uprzedziałem, że warto do niej zaglądać). Jednym z podanych rozwiązań jest użycie rozkazu *DSB* (*Data Synchronization Barrier*) po wyłączeniu zegara. 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. Jedziemy z drugą dziwną rzeczą.

Uwaga pułapka! Rejestr GPIO_BSRR to 32-bitowy rejestr. Dolne 16-bitów umożliwia atomowe ustawianie bitów GPIO_ODR, zaś górne 16-bitów umożliwia atomowe kasowanie bitów ODR. To już znamy i nic się nie zmieniło. Kłopot polega na tym że w pliku nagłówkowym mikrokontrolera STM32F429, zamiast 32-bitowego rejestru BSRR są dwa rejesty 16-bitowe: BSRRRL i BSRRRH. Odpowiadają one odpowiednio dolnej i górnej połówce rejestru BSRR. Niby fajnie – jeden służy do ustawiania bitów, drugi do kasowania. Ale proszę zwrócić uwagę na to, że definicje bitów (GPIO_BSRR_BS/BR) nie są podzielone na dwa 16-bitowe rejesty! Maski bitów są „32-bitowe”. Przeanalizujmy taki (błędny) zapis „kasujący 10-ty bit ODR”:

```
GPIOA -> BSRRH = GPIO_BSRR_BR_10;
```

niby wszystko się zgadza:

- BSRRH to górną połową rejestru BSRR – czyli ta „kasująca”
- GPIO_BSRR_BR_10 – to definicja „kasującego bitu”⁶⁵ rejestru BSRR

i du... dużo czasu zajmie szukanie błędu :) Błąd polega na tym, że ta definicja bitu odnosi się do 32-bitowego rejestru. Maska wskazuje na 26-bit rejestru. I to „się prawie zgadza” bo 26-bit rejestru BSRR to bit BR10. Ale w pliku nagłówkowym nie mamy 32-bitowego rejestru BSRR tylko dwie połówki po 16-bitów. Próba ustawienia 26-tego bitu w 16-bitowym rejestrze jest raczej skazana na niepowodzenie!

Jak to rozwiązać? Możliwości jest sporo. Można np. zmienić plik nagłówkowy i wywalić te dwa rejesty połówkowe i zrobić jeden 32-bitowy:

Tabela 3.4 Korekta pliku nagłówkowego *stm32f4xx.h*

stary zapis	nowy zapis
<pre>typedef struct { __IO uint32_t MODER; /* ... */ __IO uint32_t OTYPER; /* ... */ __IO uint32_t OSPEEDR; /* ... */ __IO uint32_t PUPDR; /* ... */ __IO uint32_t IDR; /* ... */ __IO uint32_t ODR; /* ... */ __IO uint16_t BSRRRL; /* ... */ __IO uint16_t BSRRH; /* ... */ __IO uint32_t LCKR; /* ... */ __IO uint32_t AFR[2]; /* ... */ } GPIO_TypeDef;</pre>	<pre>typedef struct { __IO uint32_t MODER; /* ... */ __IO uint32_t OTYPER; /* ... */ __IO uint32_t OSPEEDR; /* ... */ __IO uint32_t PUPDR; /* ... */ __IO uint32_t IDR; /* ... */ __IO uint32_t ODR; /* ... */ // __IO uint16_t BSRRRL; /* ... */ // __IO uint16_t BSRRH; /* ... */ __IO uint32_t LCKR; /* ... */ __IO uint32_t AFR[2]; /* ... */ } GPIO_TypeDef;</pre>

Można zmienić definicje bitów, dopisać swoje nowe, pamiętać o tym i nie stosować definicji bitów z drugiej połówki (tak jest zrobione w moim przykładzie), można nic nie robić... itd. itd. Nie jest to jakiś wielki problem, tylko taki „trap for young players” :)

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
- czasem występują rozbieżności między nazwami rejestrów/bitów w dokumentacji i w plikach nagłówkowych

⁶⁵ jak ktoś nie załapał jeszcze lub się gubi: *BSRR* – Bit Set / Reset Register; *GPIO_BSRR_BR_...* – Bit Reset; analogicznie bity ..._BS_... to bity „ustawiające” (*Bit Set*)

3.5. Wybór funkcji alternatywnej (F429)

W F429 inaczej rozwiązano 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 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 rozwiązań 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 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.

Nastecną nowością w F429 są rejestr GPIO_AFRL 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 rejestr GPIO_AFRx. Umożliwiają one przyporządkowanie konkretnej nóżce, jednej z kilku możliwych funkcji

alternatywnych, np. aby uprościć projekt płytka. Po szczegóły odsyłam do datasheeta⁶⁶ - 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 SAI1⁶⁷ - funkcja alternatywna *AF6*
- kanałem pierwszym licznika TIM13 – funkcja alternatywna *AF9*
- linią NIOWR⁶⁷ 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 wybrać tryb alternatywny pinu i za pomocą rejestrów GPIO_AFRL/H wybrać odpowiedni numer funkcji alternatywnej zgodnie z tabelą z datasheeta. 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 wolałbym 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 może być powiązany 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

66 tak datasheeta nie *reference manuala!*

67 cokolwiek to jest

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 3.6: na podstawie dokumentacji zidentyfikować nóżki odpowiadające kanałom 1, 2, 3 licznika TIM1 i obczaić sposób konfiguracji (numery funkcji alternatywnych).

Przykładowe rozwiązanie:

Nóżek oczywiście szukamy w *datasheetcie*. Najwygodniej odpalić sobie tabelę: *STM32F427xx and STM32F429xx alternate function mapping*. Lokalizujemy kolumnę z potrzebnym peryferialem (*TIM1*⁶⁸) 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 wyklaruje :)

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

Co warto zapamiętać z tego rozdziału?

- w 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: tabela *STM32F427xx and STM32F429xx alternate function mapping*
- numery wyprowadzeń: tabela *STM32F427xx and STM32F429xx pin and ball definitions*

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 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. Popatrzmy 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 włączeniu częściowego remapu: USART3_TX (linia nadawcza) znajduje się na nóżce PC10
- po włączeniu 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

Ot i cała filozofia. Dla innych układów peryferyjnych działa to identycznie. Nie jest to niestety rozwiązanie tak elastyczne jak w 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 F429
- korzystając z remappingu należy włączyć taktowanie bloku AFIO

3.7. Elektryczna strona medalu (F103 i F429)

W ramach odskoczní, popatrzmy na podstawowe parametry elektryczne portów. Poniżej, w tabeli 3.5, podaję zestawienie najważniejszych wartości na podstawie datasheetów STM32F103 oraz STM32F429. 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 układu⁶⁹. Przykładowo zapis: -5mA oznacza, że dopuszczalny prąd wypływający z nóżki, gdy jej potencjał spadnie poniżej potencjału masy wynosi

⁶⁹ pytanie tylko czy to ma być suma geometryczna czy arytmetyczna?

5mA. Wartości z plusem odnoszą się do prądu wpływającego, gdy potencjał nóżki jest wyższy od V_{DD} . Na koniec jeszcze tylko dorzuć, że ujemny prąd wstrzykiwany zakłóca w jakimś tam stopniu pracę przetwornika ADC.

Tabela 3.5 Podstawowe parametry elektryczne portów GPIO

mikrokontroler	STM32F103	STM32F429
maksymalny prąd upływu wejścia	$\pm 1\mu A$ lub $\pm 3\mu A$ dla pinów FT ⁷⁰ przy napięciu 5V	
rezystory podciągające (pull-up, pull-down)	$40k\Omega$	$10k\Omega$ dla PA10 i PB12 $40k\Omega$ dla pozostałych
obciążalność wyjść	$\pm 8mA$ (lub $\pm 20mA$ jeśli nie zależy nam na utrzymaniu katalogowych poziomów napięć wyjściowych) $\pm 3mA$ dla PC13, PC14, PC15, PI8	
absolutnie maksymalny prąd wyjścia		$\pm 25mA$
absolutnie maksymalny prąd pobierany przez cały układ	$150mA$	$270mA$
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
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 datasheetie w tabelce *Pin Definitions* w rozdziale *Pinouts and pin descriptions*. Piny tolerujące 5V mają oznaczenie FT w kolumnie *I/O Level*. Na pinach 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*):

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

- piny FT są zabezpieczone czymś w rodzaju 4V zenerki do V_{dd}
- 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-drain⁷¹
- 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 podciagające 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.

⁷¹ oczywiście musimy pilnować natężenia prądu w stanie niskim i zadbać o jego ograniczenie w razie potrzeby

4. BIT BANDING („*GEMMA GEMMARUM*”⁷²)

4.1. Ale o co chodzi?

*Bit banding*⁷³ (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 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 którejś 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*”

72 „Klejnot nad klejnotami.”

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

Co warto zapamiętać z tego rozdziału?

- *bit banding* jest mechanizmem umożliwiającym przeprowadzanie atomowych 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”⁷⁴ powoduje automatyczne, sprzętowe i atomowe ustawienie odpowiadającego mu bitu w *BB regionie*. Analogicznie wpisanie wartości zero⁷⁴ do słowa aliasu, powoduje skasowanie bitu regionu. Proszę sobie to dobrze przemyśleć – mechanizm jest bardzo prosty :)

W poprzednim akapicie 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, np. w mikrokontrolerze STM32F303 BB nie obejmuje rejestrów portów GPIO i ADC

Co warto zapamiętać z tego rozdziału?

- BB nie ma dostępu do rejestrów rdzenia i jego peryferiów
- BB nie musi obejmować wszystkich rejestrów mikrokontrolera
- DMA nie ma dostępu do BB
- BB *alias* zawiera słowa, sprzętowo połączone z bitami *BB regionu*

⁷⁴ słowo ma 32 bity, ale w przypadku BB liczy się tylko najmłodszy bit, reszta jest bez znaczenia

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 dokumentację⁷⁵ 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
- drugiemu bitowi regionu będzie odpowiadało drugie słowo aliasu, o adresie 0x4200 0004⁷⁶
- trzeciemu bitowi regionu będzie odpowiadało trzecie słowo aliasu, o adresie 0x4200 0008
- itd...

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

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

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*:
$$\text{offset_rejestru} = \text{adres_rejestru_X} - \text{adres_poczatku_regionu}$$
- obliczamy ile bitów było od początku regionu do naszego rejestru *X* (czyli w obliczonym powyżej *offsecie*):
$$\text{ilość_bitów_od_poczatku_regionu_do_X} = \text{offset_rejestru} * 8$$
- dodajemy do powyższego, numer bitu który chcemy zmienić w rejestrze *X*:
$$\text{ilość_bitów_od_poczatku_regionu_do_bitu_n} = \text{ilość_bitów_od_poczatku_regionu_do_X} + n$$

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:

$$\text{adres_w_aliasie} = \text{ilość_bitów_od_poczatku_regionu_do_bitu_n} * 4 + \text{adres_poczatkowy_aliasu}$$

Suma summarum, po uporządkowaniu całości otrzymujemy:

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

Domyślam się, 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

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 *.lss 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 wstępu (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 rejestrów 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 tabelka 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.

Co warto zapamiętać z tego rozdziału?

- a trzeba tu coś specjalnie zapamiętywać?

4.6. A gdzie jest haczyk?

A może nie ma :) No może taki mały zadziorek na upartego. Żadnych wiedzy i szukania dziury w całym odsyłam do dodatku 4.

Co warto zapamiętać z tego rozdziału?

- że BB nie jest podstępnny

4.7. Bit banding w STM32F429 (Cortex-M4)

Żadnych zmian i nowości muszę zawieść. Nic się nie zmienia. Jedyne na co można się nadziać to to, że *bit banding* nie obejmuje pamięci *CCM (Core Coupled Memory)* dostępnej w F429. Nieśmiało strzelę że to, że ktoś poczatkujący 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 warto zapamiętać z tego rozdziału?

- że BB w CM4 działa tak samo jak w CM3 i jest Twoim przyjacielem!

5. PRZERWANIA I WYJĄTKI („*MACTE ANIMO, IUVENIS!*”⁷⁷)

Proszę wyłączyć blokadę psychiczną i zanurzyć się w lekturze :) Będzie sporo nowości i trochę teorii o tym jak to *Cortex* ogarnia przerwania. 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. Mięej lektury. Przy pierwszym czytaniu można ominąć pierwszy podrozdział.

Wszystko co przeczytasz w tym rozdziale, dotyczy zarówno F103 (*Cortex-M3*) jak i 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 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 (wyjątek to pojęcie zbliżone do przerwania, tylko szersze). Po wystąpieniu wyjątku (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 (przerwań)
- *thread mode* - wszystko co nie jest przerwaniem (np. funkcja *main*)

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

- w trybie obsługi wyjątków program zawsze ma poziom uprzywilejowany (*privileged*)
- w trybie obsługi wyjątków program zawsze korzysta z głównego stosu⁷⁸

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:

77 „*Bądź mężczyzną, młodzieńcze!*”

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

- 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 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?

- jeśli nie korzystamy z systemów operacyjnych to rozdzielenie stosów i zmiana poziomu uprzywilejowania rdzenia nie są nam potrzebne
- domyślnie po *resete* procesor jest na poziomie uprzywilejowanym a podział stosów jest wyłączony
- 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 i przerwania w Cortexie

W *Programming Manualu* jest informacja, że przerwanie to wyjątek pochodzący od peryferiala lub wywołany (z premedytacją) programowo. Z tego co ja rozumiem to:

- *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)

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.

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 szczegółu odsyłam do dokumentacji)
Reset	-3 (stały, najwyższy)	Reset to reset (np. po włączeniu zasilania czy zadziałaniu watchdoga). Procesor pobiera adres stosu i początku kodu z tablicy wektorów, rejestrzy przyjmują domyślne wartości ⁷⁹ . Szczegóły w rozdziale 11.1.
NMI	-2 (stały)	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.
HardFault	-1 (stały)	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.
MemManage	konfigurowalny	Naruszenie zasad ochrony pamięci zdefiniowanych w bloku MPU lub próba wykonywania instrukcji spod adresów „niewykonywalnych” - np. z przestrzeni rejestrów peryferiów.
BusFault	konfigurowalny	Błąd magistrali, np. próba dostępu do wyłączonej pamięci zewnętrznej.
UsageFault	konfigurowalny	Nieznana instrukcja lub dzielenie przez 0 ⁸⁰ .
SVCall	konfigurowalny	SuperVisor Call, wyjątek odpalany programowo (instrukcją svc)
PendSV	konfigurowalny	Jakieś czary mary przydatne przy korzystaniu z systemów operacyjnych
SysTick	konfigurowalny	Wyjątkuje jak zegarek systemowy (SysTick) zliczy do zera, takie przerwanie zegarowe.
Interrupts	konfigurowalny	Tu symbolicznie zawierają się wszystkie przerwania od peryferiów mikrokontrolera (liczników, interfejsów etc.) dołączonych do kontrolera NVIC. Pełna lista przerwań dostępna jest w Reference Manualu.

Omówmy z grubsza o co chodzi. Tabelka 5.1 zawiera nazwy wyjątków, informacje o priorytecie i krótki opis. 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. Szansa na to, że w naszym programie pojawi się np. nieznany rozkaz

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

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

jest... nikła. 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. 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. 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 całej 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 włączania 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 i przerwań (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 ISR⁸¹). Uwaga na przyszłość: jeśli nastąpi wywłaszczenie wyjątku to oba (wywłaszczony 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 wyjątek⁸² (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. 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 rejestrem LR i wartością EXC_RETURN (do doczytania w dokumentacji rdzenia we własnym 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

81 *Interrupt Service Routine* - procedura obsługi przerwania

82 przypominam, że przerwania też są wyjątkami

atrybutów, czy innych hocków-klocków⁸³. 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 ISR. Stan wyjątku zmienia się z *pending* na *active*. Opóźnienie od pojawienia się *tego co powoduje wyjątek* do rozpoczęcia wykonywania ISR 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 wyjątku zostanie on jeszcze raz wywołany (np. jeśli w trakcie wykonywania procedury obsługi przerwania zewnętrznego, pojawi się nowe przerwanie z tego źródła) to będzie miał jednocześnie stan *active* i *pending*. Gdy zakończy się obsługa wyjątku (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.

Po zakończeniu ISR 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.
- *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 takie przerwanie przerwania przerwaniem - zostanie omówione w następnym rozdziale

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

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 wywłaszczenie

Konfigurowalny priorytet wyjątku (przerwania) jest nowością w stosunku do AVRów. Każdy wyjątek i 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)

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

Uwaga! Można się zapłatać: **im niższa jest wartość priorytetu (liczba określająca priorytet) tym priorytet wyjątku jest wyższy (wyjątek jest „ważniejszy”)**. Reset⁸⁴ ma najwyższy priorytet (-3). Program może wyłączyć wszystkie wyjątki poza Resetem i NMI.

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*

⁸⁴ 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

Priorytet grupowy decyduje o wywłaszczeniach. Tzn. jeśli w czasie trwania obsługi przerwania pojawi się nowe o wyższym priorytecie grupowym to nastąpi wywłaszczenie. Wywłaszczenie jest to przerwanie przerwania przerwaniem. 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.

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. 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 przerwania, 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 ⁸⁵	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!

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

Na koniec ważna uwaga! Dokumentacja IMHO nie preczyje jednoznacznie czy podział na priorytet grupowy i pod-priorytet dotyczy tylko przerwań od peryferiów mikrokontrolera czy też przerwań od peryferiów 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 przerwań i wyjątków, bez różnicy czy pochodzą od peryferiów rdzenia czy mikrokontrolera. I tego będę się trzymał.

Z priorytetami jest związany jeszcze jeden ciekawy mechanizm. Możliwe jest mianowicie zablokowanie przerwań 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.

Co warto zapamiętać z tego rozdziału?

- w prostych programach nie ma potrzeby zmieniania priorytetów przerwań
- 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 ISR przerwania nie są blokowane jak w AVR

5.5. Funkcje pomocnicze

Do konfigurowania przerwań 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 uprzemy 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 przerwaniu:

```
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
void __enable_irq(void) void __disable_irq(void)	włączenie i wyłączenie wyjątków z konfigurowalnym priorytetem (nie łapią się: Reset, NMI, HardFault); trochę jak cli i sei z AVR
void __enable_fault_irq(void) void __disable_fault_irq(void)	jak wyżej ale obejmuje również wyjątek HardFault
uint32_t __get_BASEPRI (void) void __set_BASEPRI(uint32_t val)	funkcje pobierające i ustawiające wartość rejestru określającego minimalny priorytet jaki musi mieć wyjątek aby został obsłużony (rejestr BASEPRI)
void NVIC_EnableIRQ(IRQn_t ⁸⁶) void NVIC_DisableIRQ(IRQn_t ⁸⁶)	funkcje włączające i wyłączające konkretne przerwanie w kontrolerze NVIC; w kontrolerze włącza się tylko przerwania od peryferiów mikrokontrolera
uint32_t NVIC_GetPendingIRQ(IRQn_t ⁸⁶)	funkcja zwraca 1 jeśli przerwanie podane w argumencie jest w stanie oczekującym (pending) w kontrolerze NVIC
void NVIC_SetPendingIRQ(IRQn_t ⁸⁶)	funkcja powoduje przejście przerwania podanego w argumencie do stanu oczekiwania (ustawienie flagi pending w kontrolerze NVIC)
void NVIC_ClearPendingIRQ(IRQn_t ⁸⁶)	funkcja powoduje skasowanie flagi oczekiwania (pending) przerwania podanego w argumencie, w kontrolerze NVIC
uint32_t NVIC_GetActive(IRQn_t ⁸⁶)	zwraca 1 jeśli przerwanie podane w argumencie jest w stanie aktywnym
void NVIC_SetPriorityGrouping(uint32_t) uint32_t NVIC_GetPriorityGrouping(void)	funkcje ustawiające i pobierające wartość rejestru ustalającego sposób podziału priorytetu na priorytet grupowy i pod-priorytet (rejestr PRIGROUP)
uint32_t NVIC_EncodePriority (uint32_t PRIGROUP, uint32_t GroupPrio, uint32_t SubPrio)	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!
void NVIC_DecodePriority(uint32_t Priority, uint32_t PRIGROUP, uint32_t* GroupPrio, uint32_t* SubPrio)	funkcja wyłuskuje z podanego priorytetu wartość priorytetu grupowego i pod-priorytetu (uwzględniając podział priorytetu za pomocą PRIGROUP); nic nie modyfikuje w rejestrach!
void NVIC_SetPriority(IRQn_t IRQn ⁸⁶ , uint32_t priority)	funkcja ustawia priorytet (zakodowany np. za pomocą funkcji NVIC_EncodePriority) przerwania podanego w argumencie
uint32_t NVIC_GetPriority(IRQn_Type IRQn ⁸⁶)	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:
 - void NVIC_EnableIRQ(IRQn_t)
 - void NVIC_DisableIRQ(IRQn_t)
 - void NVIC_ClearPendingIRQ(IRQn_t)

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

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 wywalczyć przerwanie zewnętrzne, zaś przerwanie zewnętrzne mogło wywalczyć 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.         BB(TIM3->SR,TIM_SR UIF) = 0;
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.         BB(EXTI->PR, EXTI_PR_PR13) = 1;
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*”⁸⁷)

6.1. Blink me baby one more time (F103, F429)

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)
```

⁸⁷ „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/8⁸⁸)
- 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.
```

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

88 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 8MHz⁸⁹. 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 :)

⁸⁹ 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 CM4](#)

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 */
```

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. PRZERWANIA ZEWNĘTRZNE („*FACTA SUNT VERBIS DIFFICILORA*”⁹⁰)

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 (*External Interrupt / Event Controller*) czyli 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 HSE⁹¹. 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* → wywoływane przez piny o numerze 0 (*PA0, PB0, PC0, ...*)
- *EXTI1* → wywoywane przez piny o numerze 1 (*PA1, PB1, PC1, ...*)
- ...
- *EXTI15* → wywoywane 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 *USB Wakeup Event*
- *EXTI19* – przerwanie związane na sztywno z *Ethernet Wakeup Event* (tylko w mikrokontrolerach z linii *connectivity*)

90 „Czyny są trudniejsze niż słowa.”

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

Czyli np. piny PA7, PB7, PC7, PD7... mogą generować przerwanie na linii EXTI7, przy czym jednocześnie można włączyć 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 popatrzeć 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.     RCC->APB2ENR = RCC_APB2ENR_IOPBEN | RCC_APB2ENR_IOPCEN | RCC_APB2ENR_AFIOEN;
3.
4.     gpio_pin_cfg(GPIOB, PB0, gpio_mode_output_PP_2MHz);
5.     gpio_pin_cfg(GPIOB, PB2, gpio_mode_input_floating);
6.     gpio_pin_cfg(GPIOC, PC13, gpio_mode_input_floating);
7.
8.     AFIO->EXTICR[0] = AFIO_EXTICR1_EXTI2_PB;
9.     AFIO->EXTICR[3] = AFIO_EXTICR4_EXTI13_PC;
10.
11.    EXTI->IMR = EXTI_IMR_MR2 | EXTI_IMR_MR13;
12.    EXTI->RTSR = EXTI_RTSR_TR2;
13.    EXTI->FTSR = EXTI_FTSR_TR13;
14.
15.    NVIC_EnableIRQ(EXTI2_IRQn);
16.    NVIC_EnableIRQ(EXTI15_10_IRQn);
17.
18.    while (1);
19.
20. } /* main */
21.
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” portu⁹². Przy czym nie jest to takie proste jakby się mogło wydawać :) Linii związanych 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 haczyk⁹³, 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 rejestrów 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

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

⁹² piny portów dołączone są do multipleksera, z którego „wychodzi” sygnał EXTI; rejestr EXTICR steruje tym multiplekserem

⁹³ „Trap for young players” jak powiedziałby D. Jones

(EXTICR1). W programie odwołuję się do niego poprzez tablicę. Pierwszy element ma indeks zero, stąd zapis ...EXTICR[0]... wiem pokręcone. 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 wywoływane przez zbocze rosnące (jedynka w rejestrze RTSR⁹⁴), opadające (jedynka w rejestr FTSR⁹⁴) 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 moze specjalnego sensu, ale pamiętajmy że niektóre wektory przerwań są wspólne dla wielu linii. Wtedy musimy programowo sprawdzić, która linia wywołała przerwanie.

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

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.

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: Jeśli flaga nie zostanie skasowana to po zakończeniu obsługi przerwania będzie ona dalej ustawiona i procek ponownie wejdzie w obsługę przerwania i tak w kółko.

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 głowie⁹⁵, że przyda się też rozdział *System configuration controller*. Swoją drogą w RM0008 jest, na samym początku, taka tabelka⁹⁶, 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 szczegółów odsyłam do dokumentacji

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

⁹⁵ to są plusy wielu dupogodzin spędzonych na przeglądaniu for internetowych :)

⁹⁶ Sections related to each STM32F10xxx product

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

8. LICZNIKI („FESTINA LENTE!”⁹⁷)

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*⁹⁸
- 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

97 „Spiesz się powoli!”

98 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 zliczający⁹⁹ 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 przeładowania⁹⁹ (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)¹⁰⁰
- sygnały z zewnątrz, np. zbocza z jakiegoś pinu mikrokontrolera
- impulsy z innego licznika, np. licznik 2 może zliczać przepelnienia 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!

⁹⁹ to moja prywatna nazwa, więc proszę się nie przywiązywać i nie traktować jej zbyt poważnie :)

¹⁰⁰ 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 dokumentacji¹⁰¹, 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.

101 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 zdarzenie¹⁰², które może być generowane:

- przy przepełnieniu 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 przepełnieniu 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 przepełnieniu. 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

102 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) CCMR1¹⁰³. 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.

¹⁰³ 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_CH2¹⁰⁴. 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*)¹⁰⁵ 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

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

105 *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 = (uint16_t)~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

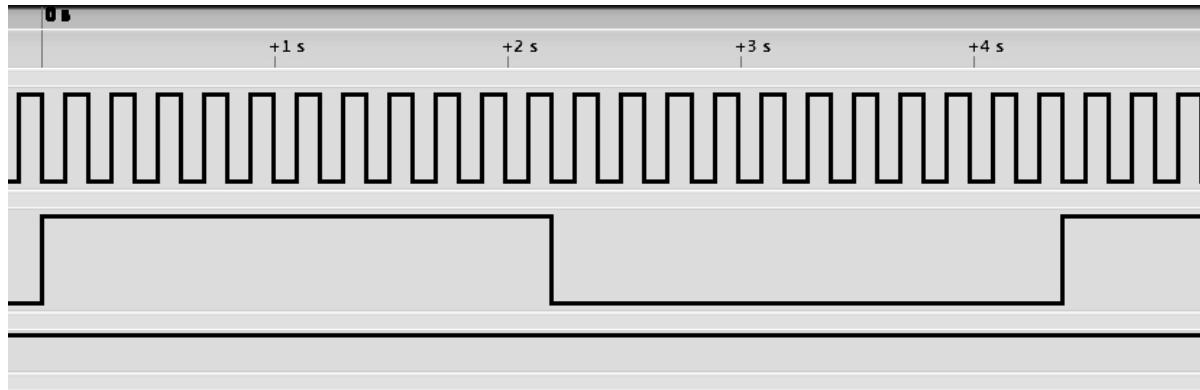
12, 13, 14) ustawiamy górną wartość zliczania¹⁰⁶ i włączamy przerwanie od UEV jak w poprzednim przykładzie

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

106 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

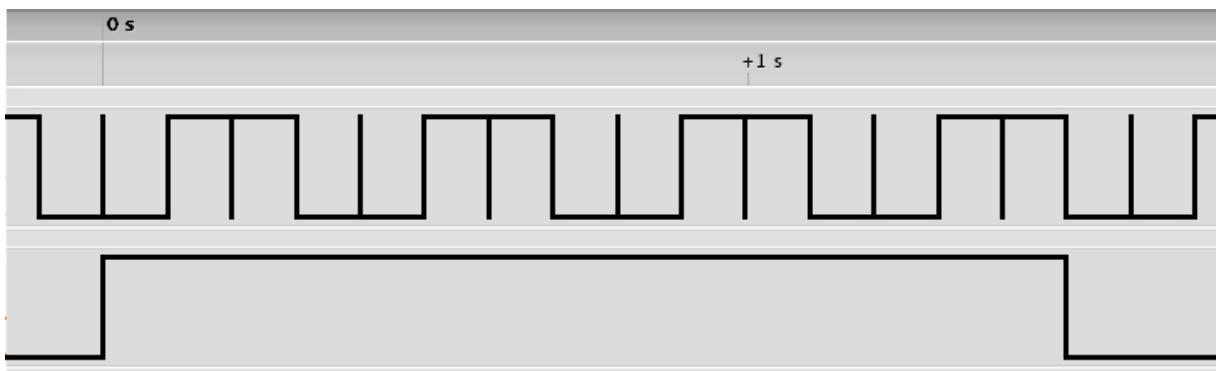
Jeszcze słówko o filtrowaniu. Uprzedzam, konkretów nie będzie bo średnio ogarniam ten temat. Generalnie chodzi o to, że poprzez bloki filtrujące mamy wpływ na częstotliwość

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. }
```

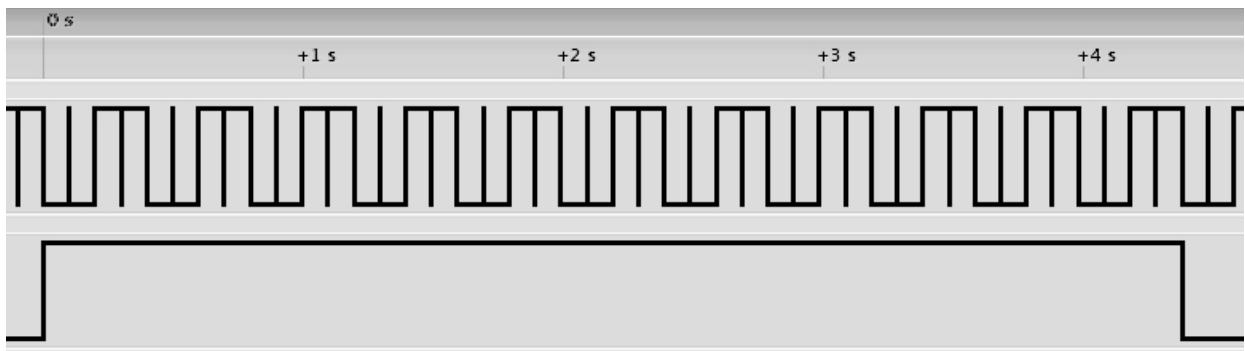
Tak to wygląda na analizatorze:



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 = (uint16_t)~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 skokowy¹⁰⁷ 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?

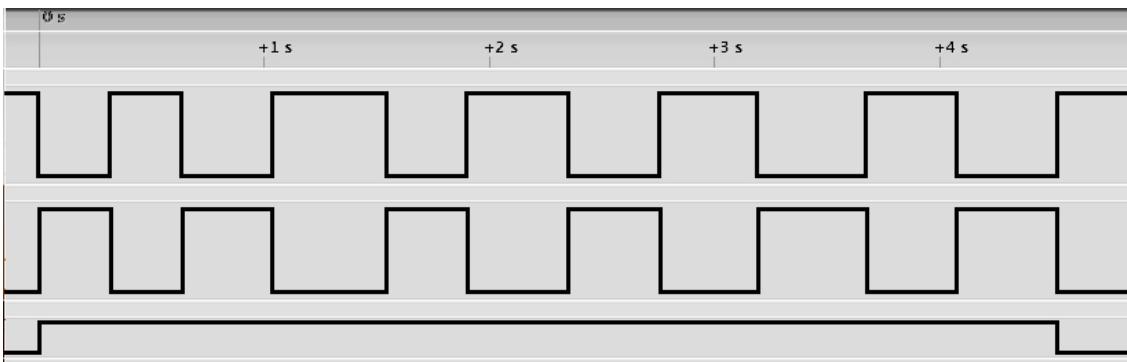
¹⁰⁷ czy jak to się tam fachowo nazywa, taki trykający przy kręceniu (aretowany?)

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 enkodera¹⁰⁸ 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.

¹⁰⁸ 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_CCMR1¹⁰⁹.

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

¹⁰⁹ 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 channel¹¹⁰*. 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

¹¹⁰ schemat ogólny licznika też się przydaje - warto go sobie wydrukować żeby był zawsze pod ręką!

- multiplekser jest konfigurowany bitami CC1E¹¹¹ 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 sygnał¹¹² - 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:

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

$$d_{PWM} = \frac{CCRx}{ARR + 1} \cdot 100 \quad [\%]$$

Do dzieła mój Szogunie!

¹¹¹ bity CC1NE dotyczą nóżki komplementarnej, olewamy je na razie

¹¹² 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 AFIO¹¹³. 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 gledzenie. Będzie za chwilę ([funkcja break](#)).

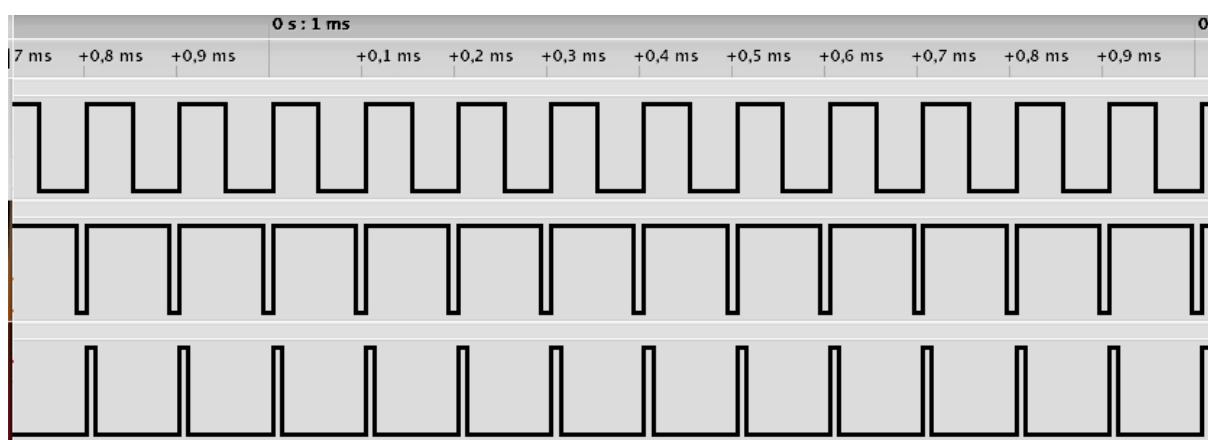
113 niby blok *Alternate Function...*

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 :)

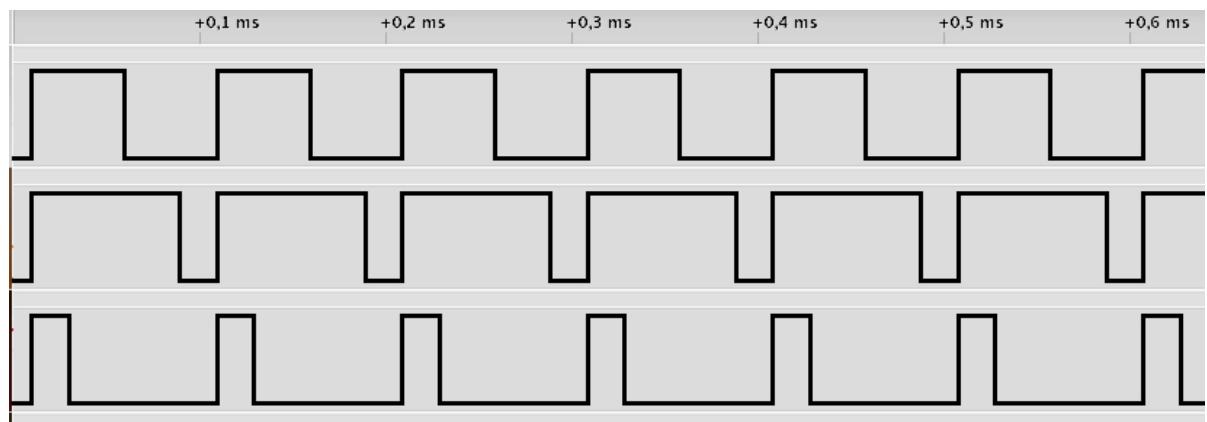
Funkcja BREAK znowu jest ukłonem w stronę energoelektroniki. To taki wyłącznik awaryjny generowania sygnałów. Po odebraniu sygnału BREAK następuje sprzętowe wyłączenie wyjść licznika i wymuszenie na nich bezpiecznych stanów. Bezpieczne stany wyjść konfiguruje się gdzieś w opcjach licznika w zależności 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

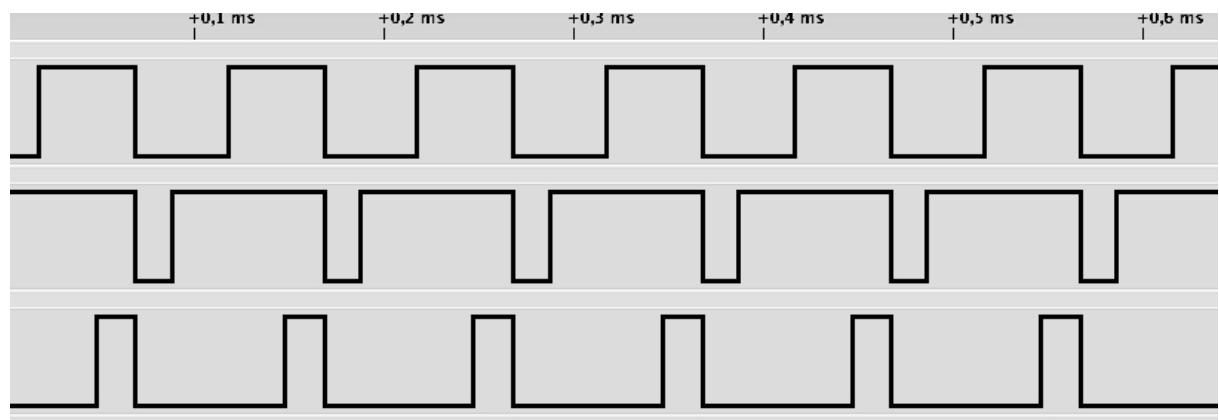
I teraz dochodzimy do bitu **TIM_BDTR_MOE**. To jest taki główny wyłącznik wyjść licznika. Działanie funkcji break polega właśnie na skasowaniu tego bitu, co wyłącza wyjścia. Jest on również **domyślnie skasowany po resecie**. Właśnie dlatego musimy go ustawić, mimo że w naszym przykładzie nie korzystamy 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.

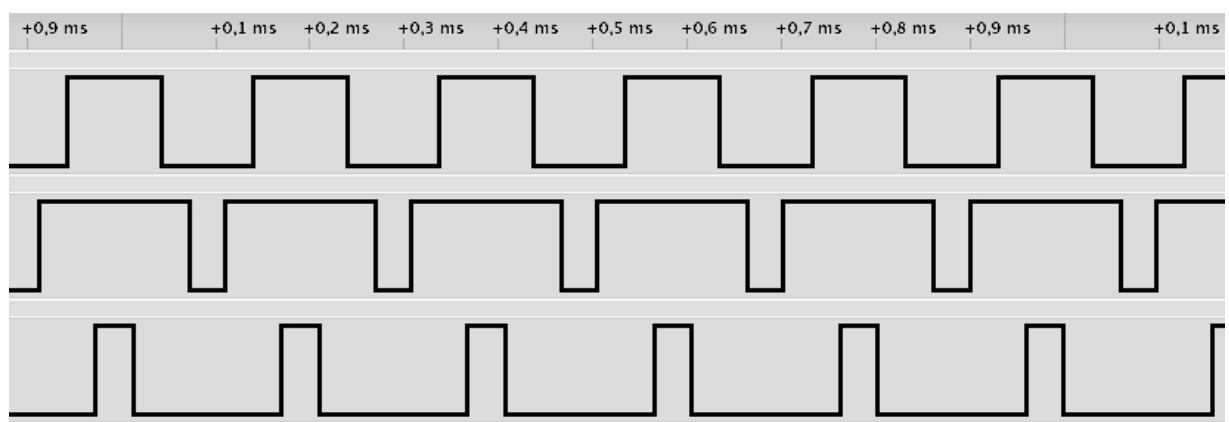
Żeby wykręcić się z dokładnego omawiania trybów pracy PWM, obiecałem pokazać przebiegi PWM dla różnych trybów. No i dotrzymuję słowa :) Trzy kanały PWM (50%, 80%, 20%) i różne tryby. Międz sobą zabawy w szukanie różnic na obrazkach :)



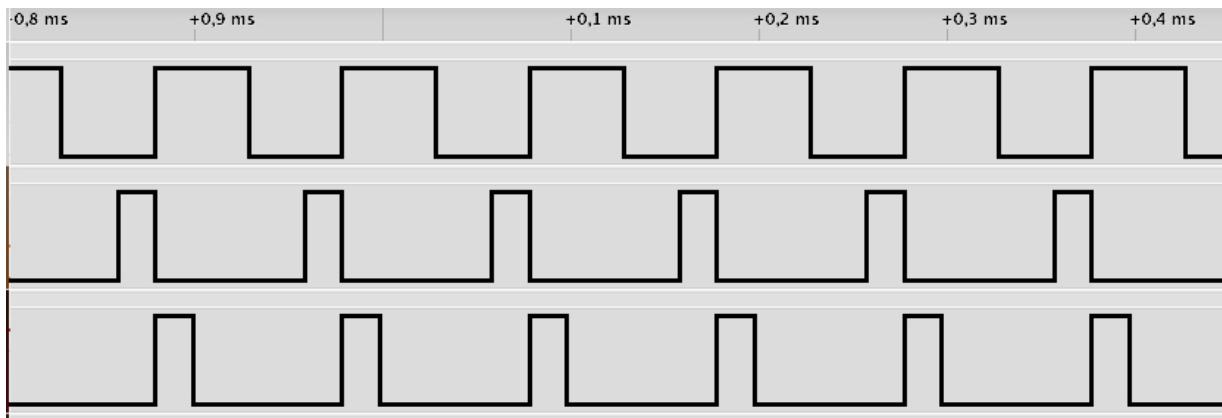
Rys. 8.6 Licznik zliczający do góry, wszystkie kanały w trybie *PWM Mode 1*
(współczynniki wypełnienia 50%, 80%, 20%)



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.         BB(TIM1->SR, TIM_SR_CC1IF) = 0;
44.         BB(GPIOB->ODR, PB0) = 1;
45.     }
46.
47.     if (TIM1->SR & TIM_SR_CC2IF){
48.         BB(TIM1->SR, TIM_SR_CC2IF) = 0;
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 przerwanie¹¹⁴, 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)

¹¹⁴ 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_CCMR1¹¹⁵)
- 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 ICxPS¹¹⁶ aktualna wartość licznika CNT jest zatrzykowana 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 STMachine, bo operacje na 32 bitowych rejestrach są trudne... Jak ktoś się nie czuje na siłach to czyta jeszcze kawałek :)

¹¹⁵ przypominam, że rejesty CCMR mają w RMie dwa odrębne opisy: jeden dla trybu *Output Compare*, drugi dla *Input Capture*

¹¹⁶ *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 licznika¹¹⁷ 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 przykłady¹¹⁸... 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)

117 *Slave Mode Controller* - nie wiem jak to zgrabnie spolszczyć - nadzorca niewolnika?

118 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 kodu¹¹⁹:

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

¹¹⁹ ach te skomplikowane STMy

Drugi punkt programu: opóźniona reakcja na sygnał z zewnątrz. Plan działania jest taki:

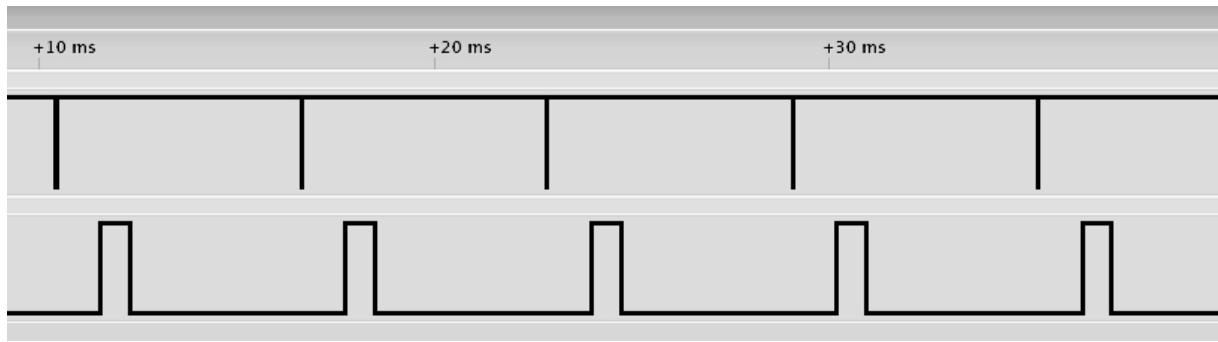
- konfigurujemy licznik w trybie *Slave Trigger Mode* - czyli będzie czekał na sygnał zewnętrzny który go wystartuje
- dodatkowo dorzucamy *One Pulse Mode* - żeby działał tylko raz po wyzwoleniu
- licznik ma generować krótki impuls na wyjściu kanału numer 2 (tryb PWM)

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 SysTicka 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 wymuszonemu 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 połączone¹²⁰: 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

120 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 ¹²¹	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 przydatność¹²². 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*

121 Capture / Compare

122 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 wyróst 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

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 proku (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 = (uint16_t)~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) mechanie pinem poprzez zmianę podciągania (góra / dół)

Co warto zapamiętać z tego rozdziału?

- a trzeba tu coś zapamiętywać?

9. BATTERY BACKUP DOMAIN (“*NON OMNIS MORIAR*”¹²³)

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*¹²⁴... 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* (F429)
- zegar czasu rzeczywistego - *RTC*
- zewnętrzny oscylator zegarkowy - *LSE*
- rejestr kontrolny domeny baterijnej - *RCC_BDCR*

Co do kwestii elektrycznych:

- napięcie baterii: 1,8 - 3,6V
- pobór prądu (orientacyjnie): $\sim 1,5\mu A$

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 podtrzymowania.

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:

123 „*Nie wszyscy umierają.*”

124 co to kur(sy)wa jest *dominium*!?

- 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 SPL, 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 peryferyjnymi¹²⁵ 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.

9) włączenie funkcji tamper i ustalenie polaryzacji wejścia

125 GPIO configurations for device peripherals

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 szcześliwe rozwiązanie bo wymusza aktywowanie funkcji tamper stanem niskim (po wciśnięciu przycisku). Po wyłączeniu zasilania płytki 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 konfiguracja¹²⁶ 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 dobrąć tak aby licznik tykał co sekundę, dzięki temu licznik będzie zliczał czas w sposób wygodny do obróbki¹²⁷ a przerwanie *second* będzie się odpalało co sekundę
 - *Overflow Interrupt* – przerwanie od przekręcenia licznika¹²⁸
- funkcję alarmu można wykorzystać do wybudzenia procka z najgłębszego trybu uśpienia (*Standby*) – patrz rozdział o uśpieniach (11.5)

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.

126 tylko rejestrów RTC_PRL, RTC_ALR, RTC_CNT i RTC_DIV

127 hasło klucz: *Unix Time, POSIX Time*

128 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 zegara¹²⁹ 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 konfiguracji¹³⁰ (CNF = 1)
- zapisać nowe wartości rejestrów¹³¹ 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

129 dotyczy również sytuacji, w której zegar był wyłączony ze względu na uśpienie procka

130 tryb konfiguracyjny jest wymagany przy zmianie wartości rejestrów RTC_PRL, RTC_CNT, RTC_ALR

131 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 11.5).

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 PI8¹³²). 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

132 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 (F429¹³³, 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

133 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 rozwiązanie (429¹³⁴, 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.

¹³⁴ płytka 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/PM¹³⁵) przechowywany jest w rejestrze: RTC_TR (*Time Register*)
- data (dzień, miesiąc, dzień tygodnia, rok) przechowywana jest w rejestrze: RTC_DR (*Data Register*)

135 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)¹³⁶. 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:

136 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 RTCCLK¹³⁷ (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_WPR¹³⁸, 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.

¹³⁷ dostępny jest osobny preskaler

¹³⁸ 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 ;)

10. UKŁADY WATCHDOG („DUO CUM FACIUNT IDEM, NON EST IDEM”¹³⁹)

10.1. Watchdog niezależny IWDG

Parafrując definicję *konia* z pierwszej polskiej encyklopedii powszechniej: „*Watchdog jaki jest, każdy widzi*”. Watchdog niezależny¹⁴⁰ (*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 rejesty:

- 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 jakiekolwiek innej wartości (np. przy kasowaniu watchcata)

139 „*Gdy dwóch robi to samo, to nie jest to samo.*”

140 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 79), 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*¹⁴¹ - 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

141 *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 przerwań¹⁴² (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

¹⁴² 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 inaczej¹⁴³: 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:

$$t_{\text{WWDG_reset}} = \frac{1}{36e6} \cdot 4096 \cdot 2^2 \cdot (0x3F + 1) \approx \frac{1,05e6}{36e6} \approx 30 \text{ ms}$$

¹⁴³ 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 rozwiązań (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 wywołyany 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 „zera¹⁴⁴”. W związku z tym flaga

144 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 DUŚ¹⁴⁵. 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

145 Do Usianej Śmierci

10.3. Porównanie układów watchdog

Mnogość narzędzi, zawsze rodzi pytanie o to, którego użyć. Spróbujmy porównać oba układy nadzorcze:

Tabela 10.1 Porównanie układów *watchdog*

Właściwość	IWDG	WWDG
<i>reset</i> 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 ¹⁴⁶	<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 <i>resetem</i>	<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)

146 Power on Reset

11. RESET, ZASILANIE I TRYBY OSZCZĘDZANIA ENERGII („*DIFFICILIS IN OTIO QUIES*”¹⁴⁷)

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*¹⁴⁸

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, ...)

Power reset działa tak samo jak *system reset*. Wywoływany jest przy:

- zadziałaniu układu monitorującego napięcie zasilania mikrokontrolera (POR, PDR lub BOR¹⁴⁹)
- wyjściu z trybu uśpienia Standby

¹⁴⁷ „Bezczynność nie daje odpoczynku.”

¹⁴⁸ nie bardzo potrafię wyobrazić sobie sens stosowania tego mechanizmu... a Ty?

¹⁴⁹ układ BOR nie występuje w mikrokontrolerze F103

Backup domain reset jest jedynym, który resetuje wszystkie rejestrze 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*¹⁵⁰ 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 ustawienie 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 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 rejestrzu RCC_CSR
- funkcja *NVIC_SystemReset()* pozwala wymusić programowy reset mikrokontrolera
- domena baterijna nie jest zerowana przy resecie

¹⁵⁰ przypominam, że w pliku nagłówkowym ten bit nazywa się inaczej: RCC_CSR_WDGRSTF (nie wiem czemu)

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 baterijne podtrzymuje pracę RTC i pamięć BKP. Jeżeli nie korzysta się z podtrzymywania baterijnego 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 biegun 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 programem ustalanym 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 datasheet¹⁵¹). Maksymalny pobór prądu wynosi coś koło 150mA. Dodatkowym źródłem zasilania jest bateria (1,65–3,6V). Zasilanie

151 tabela *Limitations depending on the operating power supply range*

batteryjne podrzynieje pracę RTC i pamięć BKP. Pobór prądu z baterii nie przekracza $1,5\mu\text{s}$. 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. Debugowanie a tryby uśpienia (F103 i F429)

Pewien problem pojawia się przy próbie połączenia debugowania i trybów obniżonego poboru mocy (uśpienia). Przekonałem się o tym dość boleśnie przy próbie uruchomienia gotowego 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 czołości. 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

11.5. Tryby obniżonego poboru mocy (F103)

Po resecie mikrokontroler pracuje w trybie *run*¹⁵². 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.

152 Run Forrest Run!

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 ustawiony 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 ustawiony bit SEVONPEND w SCB_SCR. Po wybudzeniu należy ręcznie wyczyścić flagę przerwania w peryferialu i „pendingu” w NVICu
- jeśli pojawi się zewnętrzne przerwanie (EXTI) i będzie ono skonfigurowane w trybie *Event*¹⁵³, 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 peryferii 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)!

153 patrz rejestr EXTI_EMR

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*)

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
	opóźnienie wybudzenia	1,8μs	LPDS = 1: 3,6μs LPDS = 0: 5,4μs
	regulator 1,8V	pracuje	zależnie od LPDS w PWR_CR
	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
	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 ustalony 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.

154 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 = (uint16_t)~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 11.6). Zdecydowanie proszę go przeczytać nawet jeśli ktoś nie zamierza korzystać z mikrokontrolera innego niż STM32F103 :)

Co warto zapamiętać z tego rozdziału?

- patrz rozdział 11.6

11.6. 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 AHB¹⁵⁵ wynosi 120MHz; regulator **zawsze** pracuje na tym poziomie jeśli wyłączona jest pętla PLL¹⁵⁵
- *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 PLL¹⁵⁵ jest wyłączony a procesor taktowany jest wewnętrznym lub zewnętrznym oscylatorem dużej częstotliwości. Nowo ustawiony poziom aktywuje się, gdy zostanie włączony układ PLL¹⁵⁵. 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

¹⁵⁵ 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 rozbudzanie 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 rozbudzaniu
<i>normal mode</i>	-	-	0	-	0	-
	<i>FPD</i> ¹⁵⁶	-	0	-	1	<i>flash</i>
	<i>LP</i> ¹⁵⁷	-	0	0	0	<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 UD¹⁵⁸, logika rdzenia</i>
	<i>LP + FPD</i>	3	-	1	1	<i>flash, regulator z LP i UD, logika rdzenia</i>

156 *Flash Power Down* - uśpienie pamięci Flash,

157 *Low Power* - tryb obniżonego poboru mocy regulatora napięcia

158 *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 pojedziemy 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 CPU¹⁵⁹). Brzmi całkiem sensownie.

159 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 wybudziło¹⁶⁰.

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 końca¹⁶¹...! SLEEPONEXIT powoduje, że gdy tylko procesor

160 żeby nie było, sam na to nie wpadłem :) jest to opisane w *Generic User Guide* rdzenia :)

161 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 zdarzenie¹⁶² (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,

¹⁶² 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 Yiu¹⁶³ sugeruje, aby zamiast pojedynczej instrukcji *wfe* używać zawsze sekwencji rozkazów *sev + wfe + wfe*. Po szczegółach 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*¹⁶⁴. 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

163 autor serii książek *Definitive Guide to ARM Cortex-Mx*

164 wyłączyliśmy przerwania, więc flagi same się nie skasują

12. MECHANIZM DMA („*ANNUNTIO VOBIS GAUDIUM MAGNUM: HABEMUS DMA*”¹⁶⁵)

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 STM¹⁶⁶. 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...

¹⁶⁵ „Ogłaszam wam radość wielką: mamy DMA” ;)

¹⁶⁶ 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_CPAR¹⁶⁷ i DMA_CMAR¹⁶⁸ 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

167 Channel Peripheral Address Register

168 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 register 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)** register 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 rejesty 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
- rejesty 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 rozwiążanie (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 działać<sup>169</sup> :) Kierunek przesyłu oczywiście wybieramy sobie bitem DIR.

**23, 24)** reszta konfiguracji i włączenie kanału DMA.

**26, 27)** włączamy oba liczniki

<sup>169</sup> 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 docelowego<sup>170</sup>. 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

---

<sup>170</sup> 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 FIFO<sup>171</sup>, 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 FIFO<sup>171</sup>. 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.

<sup>171</sup> 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*<sup>172</sup> 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 tabela: *FIFO threshold configurations*. Ograniczeń przy korzystaniu z trybu *burst* jest niestety trochę więcej, podsumujmy w skrócie najważniejsze:

---

172 jak przetłumaczyć *burst mode*? Może być *tryb seryjny*?

- zapis w trybie *burst* nie może przekroczyć 1kB address boundary<sup>173</sup>
- 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

---

173 przepraszam, ale nie wiem jak to zgrabnie przetłumaczyć...

A co do samych flag (każda może generować przerwanie):

- flaga zakończenia transmisji<sup>174</sup> DMA\_xISR\_TCIF
- flaga połowy transmisji DMA\_xISR\_HTIF
- flaga błędu transferu DMA\_xISR\_TEIF, ustawiana gdy wystąpi:
  - błęd szyny danych<sup>175</sup>
  - 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\_FEIF, 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*)<sup>175</sup>

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)

---

174 w trybie *double buffered* jest ustawiana co zmianę buforu

175 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!

## 13. PRZETWORNIK ADC („*SUPERFLUA NON NOCENT*”<sup>176</sup>)

### 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*<sup>177</sup> (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 datasheetcie 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

176 „*Nadmiar nie szkodzi.*”

177 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 ( $V_{ref-}$ )<sup>178</sup> powinno być połączone z masą analogową ( $V_{ssa}$ )
- dodatnie napięcie odniesienia ( $V_{ref+}$ )<sup>178</sup> 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 >100pin, 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 regularna<sup>179</sup> (*Regular Group*)
- grupa wstrzykiwana<sup>180</sup> (*Injected Group*)

---

<sup>178</sup> jeśli jest wyprowadzone w danej obudowie

<sup>179</sup> to tłumaczenie nie do końca oddaje sens nazwy... ale przynajmniej łatwo zapamiętać

<sup>180</sup> lub wtryskiwana... lub pieszczotliwiej 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 ostrzeże<sup>181</sup>! 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 strzykawka 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 strzykawkową<sup>182</sup>. 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

---

181 mogli dorzucić jakąś flagę nadpisania wyników, nie?

182 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 przytomnym<sup>183</sup>. 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 :)

183 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 kanału<sup>184</sup>, 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:

---

<sup>184</sup> 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. Działa<sup>185</sup>!

**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.

---

<sup>185</sup> 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 program<sup>186</sup>. 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!

---

186 „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. BB(ADC1->SR, ADC_SR_JEOC) = 0;
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 wielokanałowy<sup>187</sup> 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

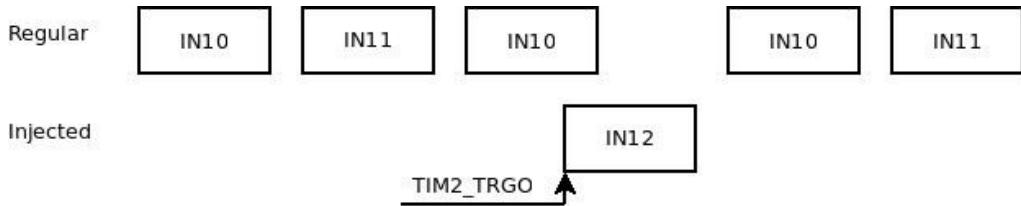
**41 - 48)** procedura obsługi przerwania: sprawdzenie źródła przerwania, skasowanie flagi, zapisanie wyniku, mignięcie diodą, koniec :)

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

---

<sup>187</sup> 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                 |

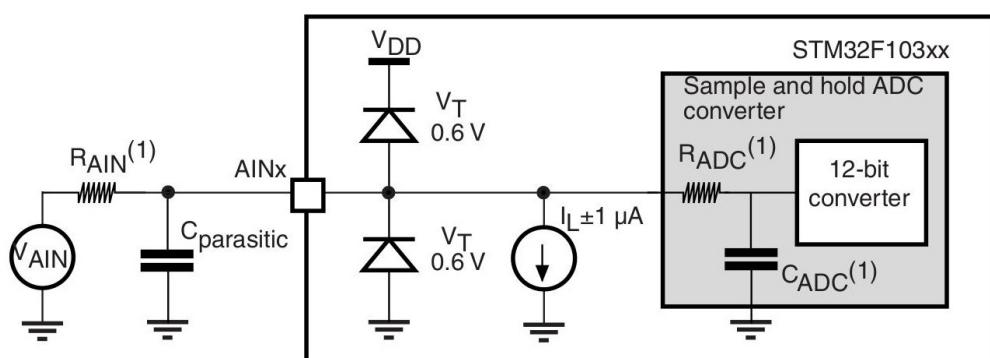
### Co warto zapamiętać z tego rozdziału?

- 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 ( $1\text{k}\Omega$ )
- $C_{ADC}^{(1)}$  - pojemność kondensatora układu próbkującego ( $8\text{pF}$ )

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 zegara<sup>188</sup> ADC:

$$T_{conv} = T_s + 12.5$$

- dzięki Panom: Harremu N. i Claude'owi S.<sup>189</sup> 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):

---

<sup>188</sup> 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

<sup>189</sup> 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  $f_{ADC} = 14MHz$ ):

$$T_{conv} = 1.5 + 12.5 = 14 \text{ cykli zegara ADC} \rightarrow 14/14 MHz = 1 \mu s$$

- 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  $f_{ADC} = 14MHz$ ):

$$T_{conv} = 239.5 + 12.5 = 252 \text{ cykle zegara ADC} \rightarrow 252/14 MHz = 18 \mu s$$

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  $380 \Omega$ . 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  $219 k\Omega^{190}$ . 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/wolnozmienne<sup>191</sup> i nie zależy nam na częstych pomiarach to mamy to wszystko głęboko w... ustawiamy np. maksymalny czas próbkowania i olewamy sprawę

---

<sup>190</sup> datasheet podaje, że maksymalna rezystancja źródła nie powinna przekraczać  $50 k\Omega$

<sup>191</sup> 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*<sup>192</sup>
- 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*

192 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*<sup>193</sup>, 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!

---

<sup>193</sup> coś chyba pokręciłem z tą nazwą... ale nawet ST się w tym gubi i w każdym dokumencie nazywa tryby inaczej :)

Przykładowe rozwiązanie (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 proceek 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 ( $0^{194}, 2, 4, 6, 8, \dots$ ) 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  $10\text{k}\Omega$  (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?

---

194 zero jest liczbą parzystą?

Policzmy co nie co:

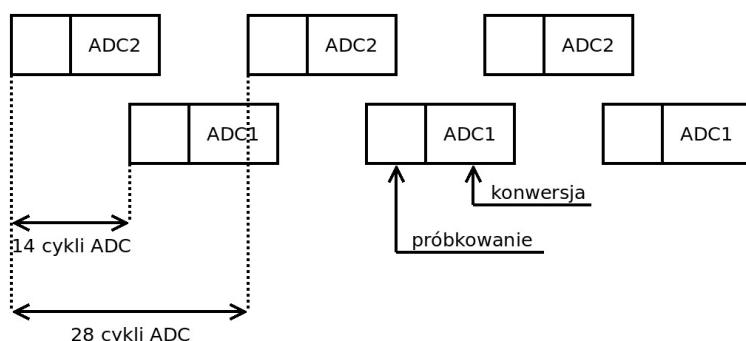
- minimalny czas konwersji (dla zegara ADC równego 14MHz) wynosi  $1\mu s$ , stąd maksymalna częstotliwość próbkowania jest równa 1MHz - czyli ok!
- maksymalna rezystancja źródła (patrz rozdział 13.3):  $380\Omega$  - czyli nie jest ok!

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 grupę <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. BB(ADC1->SR, ADC_SR_AWD) = 0;
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 $\mu$ 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 układu<sup>195</sup>. 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...

---

195 nie wiem czy gdzieś to wyczytałem czy mi się samo uroipo :)

- 21)** wyłączamy ADeCe, żeby przestał mierzyć (tzn. wprowadzamy go w tryb *power down*)<sup>196</sup>
- 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  
(średnia<sup>197</sup>: 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 zasilania<sup>198</sup>
- do kontroli wyników konwersji ADC mamy układ (pseudo) *analogowego watchdoga*
- jest coś takiego jak *offset*, czujnik temperatury, źródło napięcia odniesienia

---

196 właściwie to nie wiem czy trzeba koniecznie to robić... ale tak mi się wydaje logicznie

197 średnia jest jak bikini: bardzo dużo pokazuje, tylko nie to co istotne :)

198 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...

Połączeniu 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 ustalony w konfiguracji grupy.

Jeżeli chcemy przeprowadzić konwersję większej liczby kanałów to trzeba włączyć bit SCAN. Przetwornik wykona wtedy tyle konwersji, ile ustalono w konfiguracji danej grupy. Poza tym nic to nie zmienia! I bez sensu tworzyć otoczkę z nowym *trybem*. Po prostu będzie tyle konwersji ile ustalono. 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 samo<sup>199</sup>! Zaraz pojawi się *triple mode*<sup>200</sup>. 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” :)

---

<sup>199</sup> oczywiście są jakieś wyjątki kiedy dana konfiguracja nie ma sensu: np. wyzwalanie grupy wstrzykiwanej zewnętrznym sygnałem i jednocześnie JAUTO

<sup>200</sup> 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: 19/24<sup>201</sup> (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 kiepski) i źródła referencyjnego:

- nachylenie charakterystyki:  $Avg\_slope \ typ = 2,5mV/^{\circ}C$
- napięcie przy  $25^{\circ}C$ :  $V_{25 \ typ} = 0,76V$
- 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:

- $R_{AIN \ max}$  - maksymalna wartość rezystancji źródła sygnału, ścieżek, etc...

---

201 RM podaje liczbę 19, datasheet 24...

- $k$  - 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* ( $6k\Omega$ )

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)
- rejestyry 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*

| <b>bity konfiguracyjne</b> |             | <b>overrun</b> | <b>opis</b>                   |
|----------------------------|-------------|----------------|-------------------------------|
| <b>DMA</b>                 | <b>EOCS</b> |                |                               |
| 0                          | 0           | nieaktywne     | odbiór wyników na piechotę :) |
| 0                          | 1           | aktywne        | odbiór wyników w przerwaniu   |
| 1                          | x           | aktywne        | odbiór wyników przez DMA      |

W momencie wystąpienia *overruna* z automatu:

- wyłączany jest strumień DMA (jeśli korzystamy z DMA)
- ADC przestaje reagować na kolejne trygierze

Zadziałanie 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 ciekawiej<sup>202</sup>. Włączyłem do tego bufor FIFO.
- w konfiguracji DMA wyłączyłem tryb kołowy (bo tablica ma się zapełnić tylko jeden raz - takie założenie zadania)

202 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 rozwiązań - 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 DUS<sup>203</sup>? 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*

---

203 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 błędów<sup>204</sup>. Także ten...

**18)** do konfiguracji wielokrotnych ADeCe, 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!

204 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. Końcowe uwagi (F103 i F429)

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 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*

### **Co warto zapamiętać z tego rozdziału?**

- nic nie zapamiętywać tylko czytać noty i ćwiczyć!

## 14. PRZETWORNIK DAC („*OMNE IGNOTUM PRO MAGNIFICO*”<sup>205</sup>)

### 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) [-]
- $V_{\text{ref}}$  - 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  $V_{\text{SSA}}$  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  $\pm 1$  powoduje zmianę wyjściowej wartości analogowej o:  $\pm 3,3V / 4096 = \pm 0,81mV$ .

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

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.

---

<sup>205</sup> „*Wszystko, co nieznane, wydaje się wspaniałe.*”

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 wyjściowego (można gołącznie i wyłączyć programowo). Włączenie buforowania pogarsza, niestety, właściwości dynamiczne wyjścia.

**Tabela 14.1** Najważniejsze dane elektryczne przetworników DAC (F103 i 429):

| wielkość             | wartość                          | opis                                                                                             |
|----------------------|----------------------------------|--------------------------------------------------------------------------------------------------|
| $R_{LOAD \ min}$     | $5k\Omega$                       | minimalna wartość rezystancji obciążającej wyjście DAC przy włączonym buforowaniu <sup>206</sup> |
|                      | $1,5M\Omega$                     | minimalna wartość rezystancji obciążającej wyjście DAC przy wyłączeniu bufora <sup>206</sup>     |
| $C_{LOAD \ max}$     | $50pF$                           | maksymalna wartość pojemności obciążającej wyjścia DAC przy włączonym buforze                    |
| $Z_{OUT \ max}$      | $15k\Omega$                      | maksymalna wartość impedancji wyjścia DAC przy wyłączeniu bufora                                 |
| $V_{OUT \ min}$      | $0,5mV$                          | minimalna wartość napięcia wyjściowego przy wyłączeniu bufora                                    |
|                      | $200mV$                          | minimalna wartość napięcia wyjściowego przy włączonym buforze                                    |
| $V_{OUT \ max}$      | $V_{ref+} - 1LSB$                | maksymalna wartość napięcia wyjściowego przy wyłączeniu bufora                                   |
|                      | $V_{DDA} - 0,2V$                 | maksymalna wartość napięcia wyjściowego przy włączonym buforze                                   |
| $t_{settling \ max}$ | $F103: 4\mu s$<br>$F429: 6\mu s$ | maks. czas ustalania się napięcia po wyjściu po "diametralnej" zmianie                           |
| $t_{wakeup \ max}$   | $10\mu s$                        | maks. czas wybudzania przetwornika z trybu uśpienia                                              |
| update rate          | $1MS/s$                          | maksymalna częstotliwość małych zmian napięcia wyjściowego (do 1LSB)                             |

Przetworniki DAC oczywiście mają parę baderów. Pierwsza sprawa dotyczy zadawania wartości napięcia wyjściowego. Są trzy możliwości podawania tej wartości różniące się rozdzielczością (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:

- wartość 8b wyrównana do prawej strony: rejestr ADC\_DHR8Rx<sup>207</sup>
- wartość 12b wyrównana do prawej strony: rejestr ADC\_DHR12Rx
- wartość 12b wyrównana do lewej strony: rejestr ADC\_DHR12Lx

206 zwiększenie obciążenia (zmniejszenie rezystancji) spowoduje odjechanie wartości napięcia wyjściowego od zadanej (przetwornik nie będzie w stanie wymusić odpowiedniej wartości napięcia), względnie coś się sfajczy  
207 x to numer przetwornika

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 nadjęciu trygierza (+ 3 cykle zegara APB<sup>208</sup>)

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:

- 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
- zapętlaj

---

208 lub po 1 cyklu w przypadku wyzwalania bitem SWTRIG, taki wyjątek :)

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*<sup>209</sup>, 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*
- *simultaneous trigger with different LFSR*
- *simultaneous trigger with same triangle*
- *simultaneous trigger with different triangle*

Nie będziemy omawiać tego po kolei, 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

---

<sup>209</sup> jest schemat w RMie, jak ktoś chce to niech sobie analizuje :)

niezależny trygierz. Dane nie muszą być identyczne bo w rejestrach z „D” są wydzielone odrębne 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 register 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 ~1/3 napięcia odniesienia, drugi ~2/3 napięcia odniesienia. Zmierzyć napięcia bez obciążenia i po obciążeniu wyjść rezystorami 10kΩ. 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<br>(PA4) | -               | 2,328V | 1,835V   | 3,311V | -              | 1,108V | 124mV    | 3,263V |
| OUT2<br>(PA5) | 2,205V          | 0,972V | 0,9mV    | 3,309V | 2,211          | 2,209V | 65,5mV   | 3,262V |

Prawie mądre wnioski i obserwacje:

- na PA4, bez bufora, wychodzą bzdury (!) bo w zestawie HY-mini ten pin jest podcięgnięty do V<sub>CC</sub> przez rezistor 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
- po włączeniu buforowania, DAC radzi sobie na obu kanałach
- włączenie buforowania spowodowało jakąś tam zmianę napięcia na wyjściu
- po włączeniu 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.

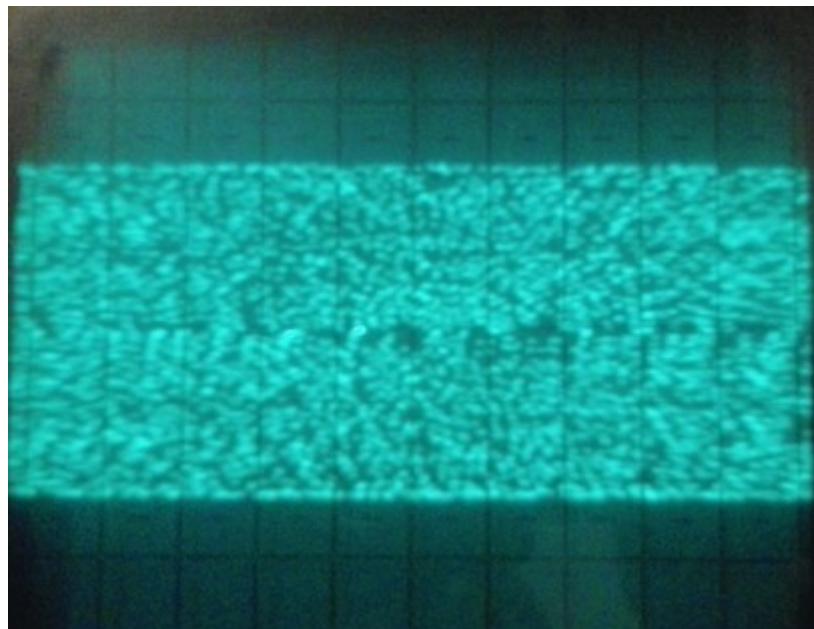
Przykładowe rozwiązanie (F103, wyjście analogowe PA5):

```
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. }
```

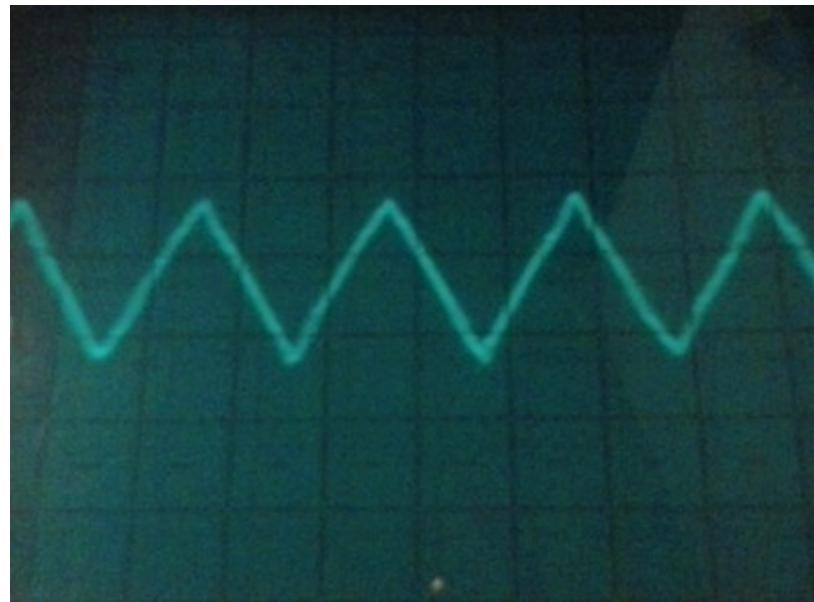
Programu nie ma co opisywać, bo nic tu specjalnego nie ma. Przypominam, że generator szumu i trójkąta działa tylko kiedy DAC jest wyzwalany sygnałem zewnętrznym i nie jest to wyzwalanie bitem SWTRIG. Efekty działania programu przedstawiam na „zrzutach”<sup>210</sup> z oscylowizora.

---

<sup>210</sup> obiecuje, że jak mi ktoś podaruje cyfrowy oscyloskop to podmienię zrzuty na ładniejsze :)



Rys. 14.1. Przebieg uzyskany z wykorzystaniem generatora szumu



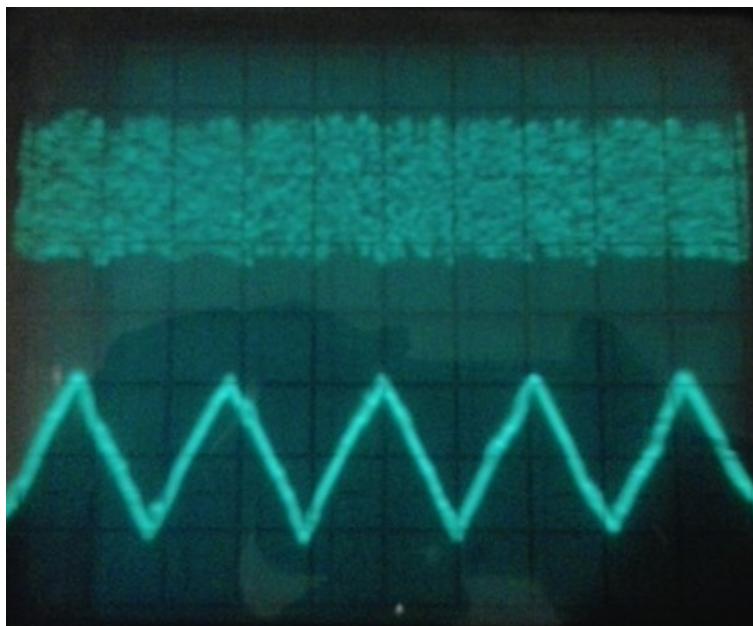
Rys. 14.2. Przebieg uzyskany z wykorzystaniem generatora trójkąta

**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 zapętlaj

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.

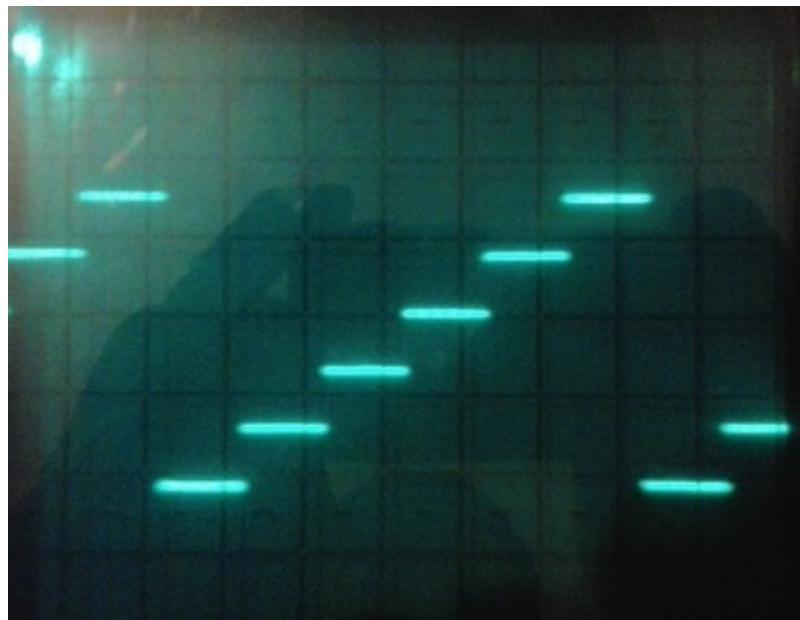
**5)** tablica z kolejnymi wartościami „napięć” (wartości wpisywane do rejestru DACa). Zgodnie z założeniem, że przykłady mają być szybkie w pisaniu i niekoniecznie eleganckie, wprowadziłem tam takie paskudztwo nazwane *WSP* (współczynnik). Sposób wyliczania wartości nikogo nie powinien dziwić, zwykła 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:

$$f_{\sin} = \frac{f_{UEV}}{n_{samples}} = \frac{f_{tim}}{(ARR + 1) \cdot (PSC + 1) \cdot n_{samples}}$$

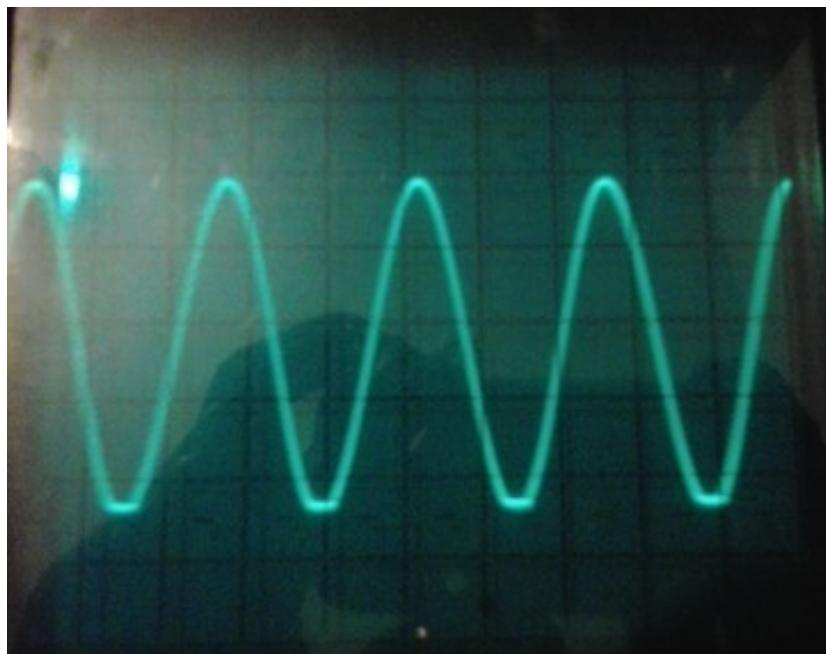
gdzie:

- $f_{tim}$  - 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. 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*”<sup>211</sup>)

### 15.1. STM32F103

Można szaleć F103 ma pięć USARTów. Na płytce HY-mini sprawa 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.

---

<sup>211</sup> „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 SPL<sup>212</sup>:

---

```

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. }

```

---

212 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 rozwiązań (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-versacze<sup>213</sup>. A! Jedną ciekawostkę dydaktyczną wcisnąłłem 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

213 albo *versucze* jak ktoś woli :)

## 15.2. STM32F429

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 S<sup>214</sup>. „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

---

214 jak Jarosław Psikuta z pewnego polskiego filmu

obrazujące do jakich układów mają dostęp poszczególne kontrolery DMA<sup>215</sup>, to możemy zobaczyć<sup>216</sup>, 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

---

215 System implementation of the two DMA controllers w RMie

216 ale trzeba się wpatrzyć

## 16. PAMIĘĆ FLASH („*VARIATIO DELECTAT*”<sup>217</sup>)

### 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)

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 *prefetch*, *cache* w nazwie). Te opcje generalnie można włączyć i zapomnieć<sup>218</sup>. 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:

---

217 „*Odmiana sprawia przyjemność.*”

218 warto sobie o nich przypomnieć jeśli program modyfikuje pamięć Flash, wszelkie bufore należą wówczas wyczyścić tak aby nie zawierały starych danych

- dla STM32F103 wielkość opóźnień jest związana z częstotliwością zegara systemowego SYSCLK<sup>219</sup> <sup>220</sup>.

**Tabela 16.1** Wymagane opóźnienia przy dostępie do pamięci flash w STM32F103

| SYSCLK [MHz] |    | wymaga liczba WAITSTATES |
|--------------|----|--------------------------|
| od           | do |                          |
| 0            | 24 | 0                        |
| 24           | 48 | 1                        |
| 48           | 72 | 2                        |

- dla STM32F429 wielkość opóźnień wynika z częstotliwości HCLK<sup>219</sup> 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 | -                | -                |

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*

219 zaraz się wyjaśni co to :) w rozdziale 17

220 właściwie to nie wiem czemu SYSCLK a nie HCLK...

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 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 (dokładniej 4 cykle zegarowe po) sprawdzany jest stan nóżek mikrokontrolera

oznaczonych BOOT0 i BOOT1. W zależności od stanu tych nóżek, następuje zmapowanie pamięci<sup>221</sup> 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 mikrokontrolera

| BOOT0 | BOOT1 | uruchomienie z |
|-------|-------|----------------|
| 0     | x     | pamięci flash  |
| 1     | 0     | bootloadera    |
| 1     | 1     | pamięci SRAM   |

Przykładowo jeśli  $\text{BOOT0} = 0$ , 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.  $\text{BOOT0} = 1$  i  $\text{BOOT1} = 0$ , to odczytując pamięć od adresów 0 procesor będzie dostawał zawartość pamięci *System Memory*, w której znajduje się firmowy bootloader.

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. Powódź 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 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 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 F429 bo nigdy nawet nie włączyłem tam bootloadera. W zestawie HY-mini wbudowana 221 tak to się fachowa nazywa?

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 rozwiązywanie ze względu na konieczność zabawy z nóżkami BOOT. No i debugować się nie da przez bootloader. Bootloader ma jednak 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... mniejszego.

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 adresu<sup>222</sup>) lub poprzez rejesty 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.

**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*.

---

222 w pliku nagłówkowym są odpowiednie definicje (OB)

## Przykładowe rozwiązań (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 rejestry 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 (zanegowana). Stąd rozmiar 16b.

9) kasuję pierwszy bit (nRST\_STOP), co odpowiada włączeniu funkcji

10) zapalam diode

**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!

## 17. SYSTEM ZEGAROWY („*FINITA EST COMOEDIA*”<sup>223</sup>)

### 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*<sup>224</sup> 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*<sup>225</sup> 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 mikrokontrolerem<sup>226</sup>. 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

223 „Komedii skończona.”

224 przynajmniej dobrze, że nie *inicjację*

225 ta „o siebie”? ja Ci dam że „o siebie” :)

226 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 Migotki<sup>227</sup>” 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

---

227 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_HSE | 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 licznika<sup>228</sup>! 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

<sup>228</sup> 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 obserwacjach<sup>229</sup>. 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 *default handlera*

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

---

<sup>229</sup> „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)
- rejestry RCC\_xxxENR pozwalają 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
- rejestry 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 dzielniakiem 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:

$$f_{vco} = f_{PLL\ in} \cdot N = f_M \cdot \frac{N}{M}$$

- 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, RNG<sup>230</sup>
- częstotliwości tych sygnałów są wyznaczane przez dwa preskalery (/P i /Q):

$$f_{PLLCLK} = f_{VCO} \cdot \frac{1}{P}$$

$$f_{PLL48CK} = f_{VCO} \cdot \frac{1}{Q}$$

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:

$$MODPER = \text{round} \left( \frac{f_{PLL\text{ in}}}{4 \cdot f_{Mod}} \right)$$

gdzie:

- $f_{PLL\text{ in}}$  - częstotliwość wejściowa PLL (za dzielnikiem M) [Hz]
- $f_{Mod}$  - częstotliwość zaburzania częstotliwości zegara (do 10kHz) [Hz]

---

230 USB wymaga zegara równo 48MHz zaś RNG i SDIO wymagają zegara o częstotliwości do 48MHz

Drugi wzorek dotyczy amplitudy zmian częstotliwości<sup>231</sup>. 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:

- $f_{PLL\ in} = 1\text{MHz}$
- $f_{Mod} = 1\text{kHz}$
- $md = 2\%$
- $f_{VCO} = 240\text{MHz}$

$$MODPER = \text{round} \frac{f_{PLL\ in}}{4 \cdot f_{Mod}} = \text{round} \frac{1\text{e}6}{4 \cdot 1\text{e}3} = \text{round} (250) = 250$$

$$INCSTEP = \text{round} \frac{(2^{15} - 1) \cdot md \cdot f_{VCO}}{100 \cdot 5 \cdot MODEPER} = \text{round} \frac{(2^{15} - 1) \cdot 2 \cdot 240}{500 \cdot 250} = \\ \text{round} (125,825) = 126$$

No to jedziemy:

**Zadanie domowe 17.3:** rozburzyć F429 na full (bez funkcji *over-drive* opisanej odrobinę w rozdziale 11.6). Tzn:

- HCLK - 168MHz
- PCLK1 - 42MHz
- PCLK2 - 84MHz

Nie zapomnieć o opóźnieniach flasha i ograniczeniach związanych z napięciem zasilania i *scalingiem*. (patrz rozdział 11.6). Standardowo: pełgający led na potwierdzenie :)

---

231 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...

Przykładowe rozwiązanie (F429, dioda na PG13):

```
1. int main(void){
2.
3. RCC->CR |= RCC_CR_HSEON;
4. RCC->PLLCFGR = 4ul<<0 | 168ul<<6 | 7ul<<24 | RCC_PLLCFGR_PLLSRC_HSE | 1ul<<29;
5. RCC->SSCGR = 500ul<<0 | 44ul<<13 | RCC_SSCGR_SSCGEN;
6. while (!(RCC->CR & RCC_CR_HSERDY));
7. RCC->CR |= RCC_CR_PLLON;
8. RCC->CFGR = RCC_CFGR_PPREG1_DIV4 | RCC_CFGR_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->CFGR |= RCC_CFGR_SW_PLL;
16. while ((RCC->CFGR & RCC_CFGR_SWS) != RCC_CFGR_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

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 ( $f_{PLL48CK} = f_{VCO} / Q = 336MHz / 7 = 48MHz$ )
- dzielnik P =  $2^{232}$  ( $f_{PLLCLK} = f_{VCO} / P = 336MHz / 2 = 168MHz$ )
- 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 (!)

---

232 to jest domyślna wartość, więc w rejestrze nic nie zmieniam

**5)** konfiguracja *Odchylacza Częstotliwości (Spread Spectrum)*, założyłem sobie odchyłki o 1% z częstotliwością 1kHz<sup>233</sup>:

$$MODPER = \text{round} \frac{f_{\text{PLL in}}}{4 \cdot f_{\text{Mod}}} = \text{round} \frac{2\text{e}6}{4 \cdot 1\text{e}3} = \text{round} (250) = 500$$

$$INCSTEP = \text{round} \frac{(2^{15} - 1) \cdot md \cdot f_{\text{VCO}}}{100 \cdot 5 \cdot MODEPER} = \text{round} \frac{(2^{15} - 1) \cdot 1 \cdot 336}{500 \cdot 500} = \text{round} (44,039) = 44$$

**6)** czekamy na HSE

**7)** włączam PLL, niech się rozpędza :)

**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.

Dalej już nie ma nic ciekawego... no może poza SLEEPONEXIT i brakiem pętelki głównej. Jak coś to odsyłam do rozdziału 11.6. 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

---

233 luz! nie przejmuj się, ja też nie mam bladego pojęcia jak się dobiera te wartości :)

## 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*<sup>234</sup> 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;
```

234 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. }
```

## 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, |
| <br>                                        | <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,    | <br>                                         |
| <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, |
| <br>                                        | <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,    | <br>                                         |
| <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, |
| <br>                                        | <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,    | <br>                                         |
| <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, |
| <br>                                        | <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,    | <br>                                         |
| <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, |
| <br>                                        | <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,    | <br>                                         |
| <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, |
| <br>                                        | <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 sposób<sup>235</sup>.

235 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 potworka<sup>236</sup>:

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 rozwinie się do postaci:

236 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 SRAḾowe (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* rejestru peryferyjnego. 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 ST<sup>237</sup>. 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

237 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. W związku z tym porównywanie czasów dostępu do pamięci aliasu i regionu nie ma w praktyce sensu :)

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 genialności<sup>238</sup> 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

---

238 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](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 zewnętrznych<sup>239</sup> 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):

---

<sup>239</sup> 239 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 :)

## **18. CHANGELOG („*HOMINIS EST ERRARE, INSIPIENTIS IN ERRORE PERSEVERARE*”<sup>240</sup>)**

**Tabela 3.1** Opis zmian dokumentu

| Data       | Versja | Komentarz  |
|------------|--------|------------|
| 06.11.2015 | 1      | publikacja |

---

<sup>240</sup> „Ludzką rzeczą jest błądzić, głupców trwać w błędzie.”