P L U M

www.plum.pl

Systemy Mikroprocesorowe w Sterowaniu

Część I

ARM Cortex-M3

dr inż. Maciej Szumski

Ignatki, styczeń 2017

Mojej Żonie - Dorocie
Moim Dzieciom - Karolowi, Michałowi, Natalii
Synowej - Adriannie
i Wnukom - Tymkowi, ...

Spis treści

Oznaczenia . . . XIII

Wstęp . . . XV

Zalety regulacji cyfrowej . . . XV
Podstawowy układ regulacji . . . XVI
Płyty rozwojowe używane w książce . . . XIX
Nazewnictwo (terminologia) . . . XXI
Przedrostki dwójkowe i przedrostki SI . . . XXI

Rozdział 1. Historia architektury ARM . . . 1

1.1. Acorn - początki . . . 1
1.2. Narodziny procesorów ARM . . . 1
1.3. Advanced RISC Machines Ltd. . . . 2
1.4. ARM i sprzedawcy mikrokontrolerów . . . 2
1.5. Zalety procesorów Cortex-M . . . 2
1.5.1. Niski pobór energii . . . 2
1.5.2. Moc obliczeniowa . . . 3
1.5.3. Efektywność energetyczna . . . 3
1.5.4. Gęstość upakowania kodu . . . 4
1.5.5. Przerwania . . . 4
1.5.6. Łatwe w użyciu, zorientowane na język C . . . 4
1.5.7. Skalowalność . . . 4
1.5.8. Uruchamianie programów . . . 4
1.5.9. RTOS . . . 4
1.5.10. CMSIS . . . 5

Rozdział 2. Reprezentacja liczb całkowitych . . . 7

2.1. Półprzewodniki . . . 7
2.2. Bit czyli najmniejsza jednostka informacji . . . 12
2.2.1. Działania logiczne . . . 12
2.3. Reprezentacja Danych . . . 13
2.4. Liczby binarne, oktalne, dziesiętne i hexadecymalne . . . 14
2.5. Liczby całkowite bez znaku . . . 16
2.6. Liczby całkowite ze znakiem . . . 16
2.6.1. Kod uzupełnień do jedności U1 . . . 16
2.6.2. Kod uzupełnień do dwóch U2 . . . 17
2.7. Dodawanie i odejmowanie liczb bez znaku, flaga *Carry* . . . 18
2.8. Dodawanie i odejmowanie liczb ze znakiem, flaga *Overflow* . . . 19
2.9. Jak kompilator tłumaczy instrukcję `if (a < b)` . . . 21
2.10. Stringi (ciągi znaków) w pamięci . . . 21

Rozdział 3. Jak działa procesor . . . 23

3.1. Tłumaczenie kodu źródłowego C na kod maszynowy . . . 23

3.2. Ładowanie programu maszynowego do pamięci . . . 25
3.2.1. Architektura Harvard versus Von Neumann . . . 26
3.2.2. Rozmieszczenie obszarów pamięci . . . 28
3.3. Rejestry wewnętrzne procesora . . . 31
3.4. Wykonywanie programu maszynowego . . . 35

Rozdział 4. Zestaw uruchomieniowy z procesorem STM32F103VB . . . 39

4.1. Wprowadzenie . . . 39
4.2. Rodzina mikrokontrolerów STM32 . . . 39
4.3. Zestaw uruchomieniowy . . . 40
4.4. Dokumentacja Cortex-M3 i STM32F103 . . . 42
4.5. Programator . . . 42
4.6. Środowisko IDE Keil μVision . . . 44
4.6.1. Zintegrowane środowisko programistyczne . . . 44
4.6.2. Instalacja środowiska μVision . . . 44
4.7. Cortex Microcontroller Software Interface CMSIS . . . 45
4.8. Nasz pierwszy: Projekt00 . . . 46
4.8.1. Opcje projektu . . . 49
4.8.2. Symulacja komputerowa . . . 56
4.9. Reset and Clock Control (RCC) . . . 60
4.9.1. Reset . . . 60
4.9.2. Taktowanie mikrokontrolera - sygnały zegarowe . . . 60
4.10. Porty GPIO . . . 62
4.10.1. Wejście: oporniki *Pull Up* i *Pull Down* . . . 63
4.10.2. Wejście, przerzutnik Schmitta . . . 64
4.10.3. Wyjście GPIO: komplementarne lub otwarty dren . . . 64
4.10.4. Szybkość wyjścia GPIO . . . 67
4.10.5. Jak dołączyć diodę LED do mikrokontrolera ? . . . 68
4.11. Projekt01_LED, migająca dioda . . . 69
4.11.1. Tworzymy nowy projekt . . . 69
4.11.2. Plik *stm32f10x_conf.h* . . . 70
4.11.3. Inicjalizacja RCC i GPIO . . . 70
4.11.4. Z jakim parametrem należy wywołać *Delay()* aby uzyskać 1 sek ? . . . 72
4.11.5. Dołączamy zewnętrzną diodę LED . . . 73
4.11.6. Oscyloskop: szybkość wyjścia GPIO . . . 74

Rozdział 5. Zestaw instrukcji ARM ISA . . . 75

5.1. Zestawy instrukcji ARM . . . 75
5.2. Budowa procesora Cortex-M . . . 76
5.3. Format instrukcji asemblerowej . . . 78
5.4. Struktura programu asemblerowego . . . 79
5.5. Dyrektywy asemblera . . . 80

Rozdział 6. Instrukcje arytmetyczno - logiczne . . . 85

6.1. Rejestr Flag . . . 85
6.2. Modyfikowanie flag . . . 86
6.3. Instrukcje przesunięcia i obrotu . . . 87
6.4. Instrukcje arytmetyczne . . . 88
6.4.1. Instrukcje dodawania i odejmowania . . . 88
6.4.2. Instrukcje krótkiego dzielenia i mnożenia . . . 90
6.4.3. Instrukcje długiego mnożenia . . . 90
6.5. Instrukcje nasycenia . . . 91

6.6. Dwuargumentowe działania logiczne . . . 91
6.6.1. Maski bitowe . . . 92
6.6.2. Definiowanie masek za pomocą dyrektywy `EQU` . . . 93
6.6.3. Operatory logiczne w C . . . 93
6.7. Kolejność bitów i bajtów . . . 93
6.8. Zmiana typu liczby całkowitej . . . 95
6.9. Instrukcje porównań . . . 95
6.10. Przesyłanie danych pomiędzy rejestrami . . . 96
6.11. Przepisz określony fragment rejestru . . . 97

Rozdział 7. Instrukcje przesyłania danych . . . 99

7.1. Wpisanie danej do rejestru . . . 99
7.1.1. Instrukcje przesyłania danych `MOV` i `MVN` . . . 99
7.1.2. Pseudoinstrukcje `LDR` i `ADR` . . . 100
7.2. Big Endian oraz Little Endian . . . 101
7.3. Dostęp do danych w pamięci . . . 102
7.4. Adresowanie pamięci . . . 103
7.4.1. *Pre-index*, *Post-index* oraz *Pre-index with Update* . . . 103
7.4.2. Instrukcje `Load` i `Store` . . . 103
7.5. `Load` i `Store` z listą rejestrów . . . 104

Rozdział 8. Instrukcje rozgałęzień *Branch* . . . 107

8.1. Testowanie warunków . . . 107
8.2. Instrukcje rozgałęzień . . . 109
8.3. Warunkowe wykonanie instrukcji . . . 111
8.4. Instrukcja kompilatora `If-then` . . . 111
8.4.1. Złożone wyrażenie logiczne z OR . . . 112
8.4.2. Złożone wyrażenie logiczne z AND . . . 112
8.5. Instrukcja kompilatora `If-then-else` . . . 113
8.6. Instrukcje pętli . . . 113
8.6.1. Pętla `For` . . . 113
8.6.2. Pętla `While` . . . 114
8.6.3. Pętla `Do while` . . . 115
8.7. Instrukcja kompilatora `Switch` . . . 115

Rozdział 9. Wyświetlacz LCD 1602 . . . 117

9.1. Wymiana danych z wyświetlaczem (4 bity danych) . . . 118
9.2. Instrukcje . . . 119
9.3. Projekt02_LCD . . . 122

Rozdział 10. Programowanie strukturalne . . . 133

10.1. Podprogramy . . . 133
10.2. Stos . . . 135
10.3. Ochrona rejestrów w podprogramie . . . 135
10.4. Przekazywanie parametrów przez rejestry . . . 137
10.5. Projekt03_asm . . . 138
10.5.1. Bit parzystości - przekazywanie wartości . . . 138
10.5.2. Wyszukanie największej liczby - przekazywanie wskaźnika . . . 139

Rozdział 11. Kontroler przerwań NVIC, SysTick i RTC . . . 141

11.1. Przerwania . . . 141
11.2. Tryby i stany pracy procesora . . . 141
11.3. Program obsługi przerwania ISR . . . 142

11.4. Układ zarządzania przerwaniami NVIC . . . 145
11.4.1. Priorytety przerwań . . . 146
11.4.2. Funkcje CMSIS do obsługi NVIC . . . 147
11.5. Zegar systemowy SysTick . . . 148
11.6. Projekt04_SysTick . . . 149
11.7. Przerwania zewnętrzne . . . 151
11.8. Klawiatura matrycowa 4 × 4 . . . 153
11.9. Projekt05_keypad . . . 155
11.10. Projekt06_scanf . . . 159
11.11. Zegar czasu rzeczywistego RTC . . . 160
11.11.1. Czas UNIXowy . . . 160
11.11.2. Konfiguracja zegara RTC . . . 161
11.12. Projekt07_RTC . . . 162

Rozdział 12. Liczniki i sterowanie PWM . . . 163

12.1. Budowa licznika i tryby zliczania . . . 163
12.2. Odmierzanie odcinków czasu . . . 165
12.3. Projekt08_toggleLED . . . 166
12.4. Pomiar długości impulsu . . . 169
12.5. Projekt09_TIM2switch, eliminacja drgań styków . . . 172
12.6. Projekt10_RadioControl . . . 175
12.7. Projekt11_6WheelDrive . . . 180
12.8. Projekt12_DistanceSensor, dalmierz ultradźwiękowy . . . 184

Rozdział 13. Płyta STM3210E-EVAL, silniki krokowe . . . 187

13.1. Katalog ARM2CortexM3-STM3210E-EVAL . . . 188
13.2. Projekt01_LED . . . 189
13.3. Projekt14_TFTLCD . . . 189
13.3.1. Kolory . . . 191
13.3.2. Tekst . . . 191
13.3.3. Grafika . . . 192
13.3.4. Zdjęcia i obrazki . . . 192
13.4. 4-Pin Trace ETM . . . 193
13.4.1. Opóźnienie programowe . . . 194
13.5. Silnik krokowy . . . 196
13.5.1. Silnik o zmiennej reluktancji VR – *variable reluctance* . . . 196
13.5.2. Silnik z magnesem trwałym PM – *permanent magnet* . . . 197
13.5.3. Silnik hybrydowy HB – *hybrid motor* . . . 197
13.5.4. Silniki krokowe unipolarne i bipolarne . . . 198
13.5.5. Sterowanie falowe (jednofazowe) . . . 199
13.5.6. Sterowanie pełnokrokowe (dwufazowe) . . . 200
13.5.7. Sterowanie półkrokowe . . . 200
13.6. Projekt15_StepperMotor . . . 201
13.6.1. Płyta ZL27ARM . . . 202
13.6.2. Płyta STM32F10E-EVAL . . . 204
13.7. Indukcyjność . . . 204
13.7.1. Stała czasowa obwodu RL . . . 204
13.7.2. Sterowanie L/nR . . . 208
13.7.3. Sterowanie z kluczowaniem prądu . . . 209
13.7.4. Sterowanie bipolarne silników krokowych . . . 209
13.7.5. Wolne i szybkie rozładowanie energii uzwojenia . . . 211
13.8. Projekt16_StepperMotor . . . 211

Rozdział 14. Liczby wymierne stałopozycyjne . . . 215
14.1. Binarna arytmetyka stałopozycyjna . . . 216
14.1.1. Liczby stałopozycyjne bez znaku . . . 216
14.1.2. Liczby stałopozycyjne ze znakiem . . . 217
14.1.3. Konwersja liczby wymiernej do postaci stałopozycyjnej . . . 217
14.1.4. Kompromis pomiędzy rozdzielczością a zakresem . . . 217
14.1.5. Arytmetyka stałopozycyjna z nasyceniem . . . 218
14.2. Działania niezależne od formatu . . . 218
14.2.1. Negacja i moduł liczby . . . 218
14.2.2. Dodawanie i odejmowanie . . . 219
14.3. Działania zależne od formatu (Q15.16) . . . 220
14.3.1. Mnożenie stałopozycyjne . . . 221
14.3.2. Dzielenie stałopozycyjne Q15.16 . . . 223
14.3.3. Dzielenie liczb 64-bitowych ze znakiem . . . 225
14.3.4. Operacje pomocnicze na liczbach 64-bitowych . . . 227
14.3.5. Dzielenie liczb całkowitych dodatnich *long division* . . . 228
14.4. Projekt17_FixedPoint_sin . . . 230
14.5. Projekt18_cube3D . . . 232
14.5.1. Obroty na płaszczyźnie . . . 232
14.5.2. Macierze obrotu . . . 233
14.5.3. Złożenie obrotów na płaszczyźnie . . . 234
14.5.4. Obroty w przestrzeni $\mathcal{R}^3$ . . . 235
14.5.5. Złożenie obrotów w przestrzeni $\mathcal{R}^3$ . . . 236
14.5.6. Projekt18_cube3D . . . 237
14.6. Projekt19_cube3D_FixedPoint . . . 238

Rozdział 15. Bezpośredni dostęp do pamięci DMA . . . 239
15.1. DMA Bus Matrix . . . 239
15.2. Kanały DMA . . . 241
15.3. Programowanie DMA . . . 243
15.4. Projekt20_DMA_M2M . . . 244

Rozdział 16. Przetwornik analogowo–cyfrowy ADC . . . 247
16.1. Przetwornik ADC z sukcesywną aproksymacją . . . 247
16.1.1. Próbkowanie . . . 248
16.1.2. Kwantyzacja . . . 249
16.1.3. Kodowanie . . . 251
16.2. Kanały wejściowe ADC . . . 251
16.3. Wyzwalanie ADC i okres próbkowania . . . 252
16.4. Autokalibracja . . . 253
16.5. Przetwarzanie ADC z wykorzystaniem przerwań SysTick . . . 253
16.5.1. Projekt21_ADC1 . . . 253
16.6. Pętla prądowa 4 - 20 mA . . . 255
16.6.1. Źródło napięciowe DC: stałe napięcie na zaciskach . . . 255
16.6.2. Źródło prądowe DC: stały prąd w obwodzie . . . 256
16.6.3. Parametryzowane źródło prądowe . . . 256
16.6.4. Układ pomiarowy pętli prądowej . . . 257
16.7. Projekt22_4-20mA przemysłowy pomiar temperatury . . . 258

Rozdział 17. Przetwornik cyfrowo–analogowy DAC . . . 261
17.1. Budowa przetwornika DAC . . . 261
17.2. DAC w procesorach STM32F103 . . . 267
17.3. Wzmacniacz wyjściowy . . . 267

17.4. Projekt23_DAC_sin . . . 268

Rozdział 18. Transmisja Szeregowa . . . 271

18.1. USART Universal Synchronous/Asynchronous Receiver-Transmitter . . . 272
18.1.1. Ramka . . . 272
18.1.2. Prędkość bodowa . . . 273
18.1.3. Warstwa fizyczna . . . 274
18.2. Warstwa łącza danych . . . 275
18.2.1. INTEL HEX Format . . . 275
18.2.2. Modbus . . . 275
18.2.3. Master (klient) – slave (serwer danych) . . . 276
18.2.4. CRC w Modbus RTU . . . 277
18.3. Projekt24_1_Modbus_Master . . . 278
18.4. Projekt25_Modbus_slave . . . 282
18.5. Projekt24_2_Modbus_Master . . . 284
18.6. I^2C Inter-Integrated Circuit . . . 287
18.6.1. Czujnik temperatury STLM75 . . . 289
18.6.2. Projekt26_I2C . . . 290
18.7. SPI Serial Peripheral Interface . . . 291
18.7.1. Projekt27_SPI . . . 292

Rozdział 19. Obiekt regulacji . . . 295

19.1. Obiekt oscylacyjny . . . 295
19.2. Obiekt wieloinercyjny z opóźnieniem . . . 296
19.3. Projekt28_obiekt . . . 296

Rozdział 20. Cyfrowy regulator PID . . . 299

20.1. Podstawowy schemat układu regulacji . . . 299
20.1.1. Transmitancja operatorowa regulatora PID . . . 300
20.2. Cyfrowa realizacja algorytmu PI . . . 300
20.2.1. Składowa proporcjonalna algorytmu . . . 301
20.2.2. Składowa całkowa algorytmu . . . 301
20.2.3. Czas zdwojenia T_I . . . 303
20.2.4. Projekt29_PI . . . 303
20.3. Cyfrowa realizacja algorytmu PID . . . 304
20.3.1. Czas wyprzedzenia T_D . . . 305
20.3.2. Składowa różniczkowa algorytmu . . . 305
20.3.3. Projekt30_PID . . . 307
20.4. Dobór nastaw regulatora PID . . . 310

Rozdział 21. Lokowanie biegunów . . . 313

21.1. Transmitancja dyskretna i transformata $\mathcal{Z}$ (Laurenta) . . . 313
21.1.1. Operator różnicy wstecznej Δ . . . 316
21.1.2. Bieguny transmitancji (*poles*) . . . 316
21.1.3. Bieguny zespolone sprzężone . . . 318
21.1.4. Twierdzenie o wartości końcowej . . . 319
21.2. Prosty projekt regulatora . . . 320
21.2.1. Wzory Viéte'a . . . 322
21.2.2. Projekt31_1_pole_placement . . . 325
21.3. Zwiększenie odporności na zakłócenia . . . 326
21.3.1. Zakłócenie stałe odniesione do wejścia obiektu . . . 327
21.3.2. Szum pomiarowy . . . 328
21.3.3. Projekt31_2_pole_placement . . . 333

21.3.4. Projekt31_3_pole_placement, obiekt oscylacyjny 334
Rozdział 22. RTOS Real Time Operating Sysytem 337
22.1. Podwójny wskaźnik stosu: MSP i PSP 338
22.2. Projekt32_Multitasking 339
Dodatek A. Schemat zestawu uruchomieniowego ZL27ARM 349
Dodatek B. Znaki ASCII7 i operatory języka C 353
Dodatek C. Karty katalogowe silników krokowych 357
Dodatek D. Schemat płyty SMSboard 361
Dodatek E. Schemat płyty KP_board 363
Dodatek F. Schemat płyty RS232/485 365
Bibliografia 367
Elektronika 367
Mikrokontrolery 367
Automatyka 367
Skorowidz 369

Oznaczenia

Skróty

ADK	*AMBA Design Kit*
AHB	*Advanced High-Performance Bus*
AHB-AP	*AHB Access Port*
AMBA	*Advanced Microcontroller Bus Architecture*
APB	*Advanced Peripheral Bus*
API	*Application Programming Interface*
ARM ARM	*ARM Architecture Reference Manual*
ASIC	*Application Specific Integrated Circuit*
ATB	*Advanced Trace Bus*
BE8	*Byte Invariant Big Endian Mode*
CMSIS	*Cortex Microcontroller Software Interface Standard*
CPI	*Cycles Per Instruction*
CPU	*Central Processing Unit*
DAP	*Debug Access Port*
DSP	*Digital Signal Processor / Digital Signal Processing*
DWT	*Data WatchPoint and Trace*
EABI/ABI	*Embedded Application Binary Interface*
ELF	*Executable and Linkable Format*
ETM	*Embedded Trace Macrocell*
FPB	*Flash Patch and Breakpoint*
FPGA	*Field Programmable Gate Array*
FPU	*Floating Point Unit*
FSR	*Fault Status Register*
GCC	*GNU Compiler Collection*
GNU	*GNU is Not Unix*
GPIO	*General Purpose Input Output*
ICE	*In-Circuit Emulator*
IDE	*Integrated Development Environment*
IRQ	*Interrupt Request (external interrupts)*
ISA	*Instruction Set Architecture*
ISR	*Interrupt Service Routine*
ITM	*Instrumentation Trace Macrocell*
JTAG	*Joint Test Action Group*
JTAG-DP	*JTAG Debug Port*
LR	*Link Register*
LSB	*Least Significant Bit*
LSU	*Load / Store Unit*
MAC	*Multiply Accumulate*

MCU	*Microcontroller Unit*
MMU	*Memory Management Unit*
MPU	*Memory Protection Unit*
MSB	*Most Significant Bit*
MSP	*Main Stack Pointer*
NaN	*Not a Number*
NMI	*Non-Mascable Interrupt*
NVIC	*Nested Vectored Interrupt Controller*
OS	*Operating System*
PC	*Program Counter*
PMU	*Power Management Unit*
PSP	*Process Stack Pointer*
PPB	*Private Peripheral Bus*
PSR	*Program Status Register*
RTOS	*Real-Time Operating System*
SCB	*System Control Block*
SCS	*System Control Space*
SIMD	*Single Instruction, Multiple Data*
SoC	*System-on-a-Chip*
SP	*Stack Pointer*
SRPG	*State Retention Power Gating*
SW	*Serial-Wire*
SW-DP	*Serial-Wire Debug Port*
SWJ-DP	*Serial-Wire JTAG Debug Port*
TPA	*Trace Port Analyzer*
TPIU	*Trace Port Interface Unit*
TRM	*Technical Reference Manual*
UAL	*Unified Assembly Language*
WIC	*Wakeup Interrupt Controller*
V_{DD}	napięcie zasilające, dodatnie w odniesieniu do potencjału masy
V_{SS}	potencjał masy

Wstęp

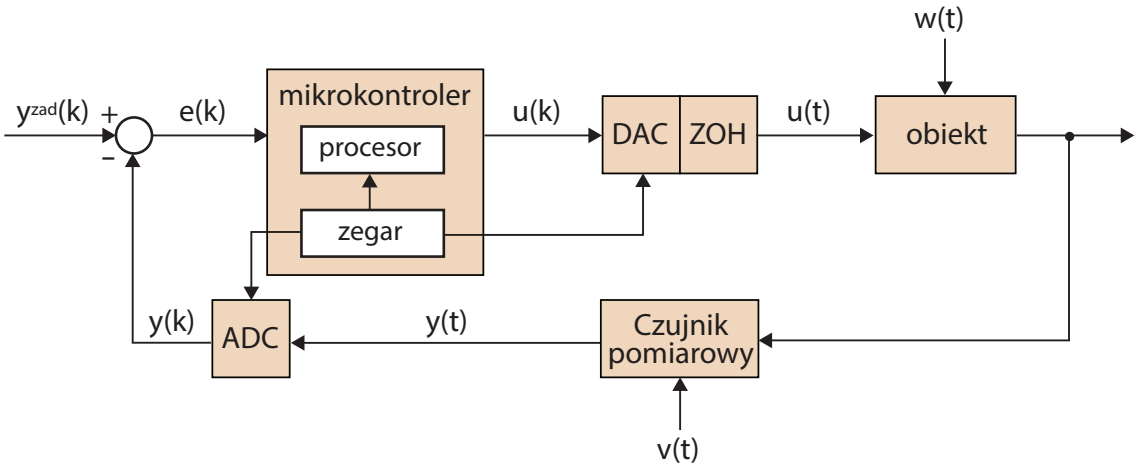
Książka została napisana jako podręcznik do przedmiotu SMS *Systemy Mikroprocesorowe w Sterowaniu*. Autor zakłada, że Czytelnik zna podstawy automatyki i posiada umiejętność programowania w języku C na poziomie średnio - zaawansowanym. W książce opisane są układy regulacji obiektów liniowych, niezmiennych w czasie (*Linear Time-Invariant*) LTI z jednym wejściem i jednym wyjściem (*Single Input Single Output*) SISO.

Zalety regulacji cyfrowej

Praktycznie, wszystkie współczesne regulatory to systemy cyfrowe. Można je budować jako cyfrowe przybliżenie regulatorów analogowych, ale nie pozwala to w pełni wykorzystać możliwości jakie daje technika cyfrowa. Uzyskamy co najwyżej regulator prawie tak dobry, jak najlepszy analogowy. Dlatego warto poznać tajniki regulacji cyfrowej. Regulacja cyfrowa ma wiele wyraźnych zalet w stosunku do analogowej, co tłumaczy jej niezwykłą popularność:

- **Dokładność**. Sygnały są reprezentowane cyfrowo. Zastosowanie liczb zmiennopozycyjnych umożliwia przetwarzanie sygnałów z bardzo małymi błędami w porównaniu do sygnałów analogowych obciążonych szumem, dryftem temperaturowym i dryftem wynikającym ze starzenia elementów. Reprezentacja cyfrowa umożliwia także rejestrację (archiwizację) przebiegu regulacji i zdarzeń (alarmów) wewnątrz regulatora, co nie było możliwe w przypadku regulatorów analogowych.
- **Implementacja prawa regulacji**. Przetwarzanie cyfrowe umożliwia wykonywanie obliczeń (mnożenie, dzielenie, całkowanie, różniczkowanie, ...) z dokładnością nieporównywalnie większą niż w przypadku sygnałów analogowych. Regulatory cyfrowe umożliwiają realizację zaawansowanych, nieliniowych algorytmów predykcyjnych (np. opartych o sieci neuronowe), których realizacja analogowa byłaby niemożliwa.
- **Elastyczność**. Regulator analogowy wymaga gruntownej przebudowy w celu zmiany prawa regulacji. W regulatorze cyfrowym, najczęściej wystarczy zmienić oprogramowanie, bez zmiany sprzętu. Możliwe jest także pobieranie ze strony producenta aktualizacji oraz wgrywanie różnych algorytmów (predykcyjnych, adaptacyjnych), które można dobrać do obiektu już w miejscu instalacji.
- **Zdalny nadzór i diagnostyka**. Regulator cyfrowy może być połączony z internetem, co pozwala na zbieranie danych o procesie w rozległych układach regulacji takich jak krajowa sieć gazowa, czy sieci wodociągowe. Takie połączenie ułatwia także pracę serwisu, który może na przykład określić, czy konieczny jest wyjazd, czy wystarczy interwencja na miejscu, personelu o niższych kwalifikacjach.

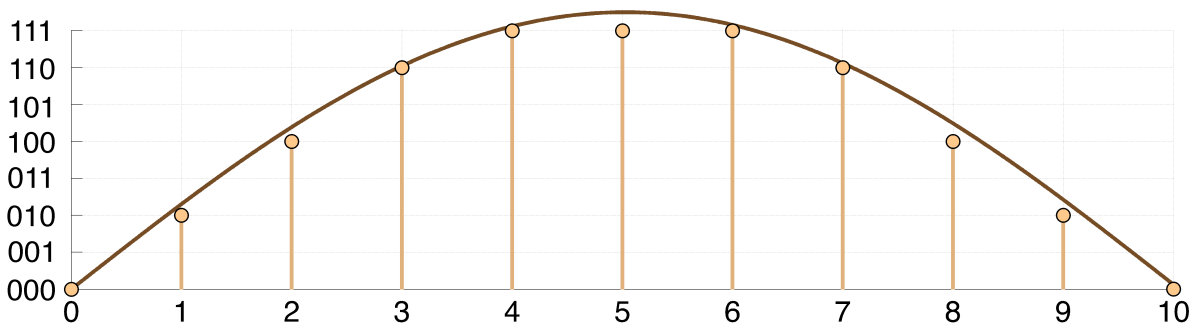
- **Szybkość działania**. Jeszcze w końcówce XX wieku poważną barierą we wprowadzaniu techniki mikroprocesorowej do regulacji przemysłowej była mała moc obliczeniowa mikroprocesorów w połączeniu z dużym poborem energii. Procesory o nieco większej mocy wymagały wymuszonego chłodzenia, co w warunkach przemysłowych często jest niemożliwe, ze względu na wymaganą niezawodność (awaryjność układów elektronicznych gwałtownie rośnie ze wzrostem ilości energii, którą trzeba rozproszyć). Zbyt mała moc obliczeniowa stanowiła barierę do stosowania zaawansowanych algorytmów obliczeniowych (np. wymagających rozwiązywania *on line* kwadratowego zadania optymalizacji) do szybkich procesów. Takie algorytmy praktycznie stosowane były wyłącznie do bardzo wolnych procesów chemicznych. W ostatnich latach nastąpił gwałtowny rozwój mikrokontrolerów o mocach wystarczających do realizacji zaawansowanych algorytmów regulacji większości procesów przemysłowych z tak małym poborem energii, że możliwe jest budowanie niezawodnych układów.
- **Niski koszt**. Ceny układów cyfrowych, także mikrokomputerów są obecnie tak niskie, że są one powszechnie stosowane nawet w najtańszych i najprostszych regulatorach.



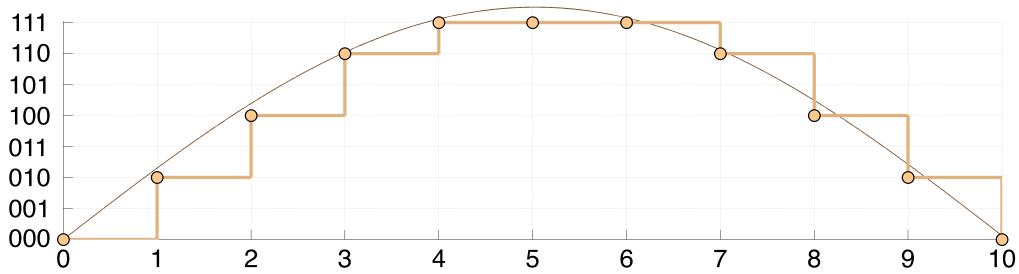
Rys. 1: Układ regulacji cyfrowej ze sprzężeniem zwrotnym

Podstawowy układ regulacji

Podstawowy układ regulacji cyfrowej ze sprzężeniem zwrotnym pokazany jest na rysunku 1. **Obiekt regulacji** (*plant*) jest to dowolny proces fizyczny, który do zadowalającej pracy wymaga sygnałów sterujących. Przez "zadowalającą pracę" rozumiemy to, że sygnał wyjściowy $y(t)$ różni się od sygnału odniesienia (wartości zadanej) $y^{zad}(t)$ dostatecznie mało, pomimo oddziaływania zakłóceń $w(t)$ na obiekt i pomimo szumu pomiarowego $v(t)$. Jeżeli sygnał $y^{zad}(t)$ jest stały w długich okresach czasu, mówimy o zadaniu regulacji stałowartościowej. Jeżeli $y^{zad}(t)$ jest zmienne to mamy regulację nadążną. Sygnały ciągłe będziemy oznaczać jako $x(t)$, gdzie t jest czasem ciągłym. Sygnały cyfrowe oznaczymy $x(k)$, gdzie k oznacza kolejne (dyskretne, czyli skwantowane) chwile czasowe. W układzie regulacji cyfrowej (rysunek 1) występują oba rodzaje sygnałów jednocześnie. Pomiędzy nimi są przetworniki: sygnał ciągły jest zamieniany na cyfrowy (dyskretny) w przetworniku analogowo–cyfrowym (*Analog Digital Converter*), oznaczonym na rysunku 1 jako *ADC*.



Rys. 2: Przetwornik ADC - dyskretyzacja sygnału analogowego

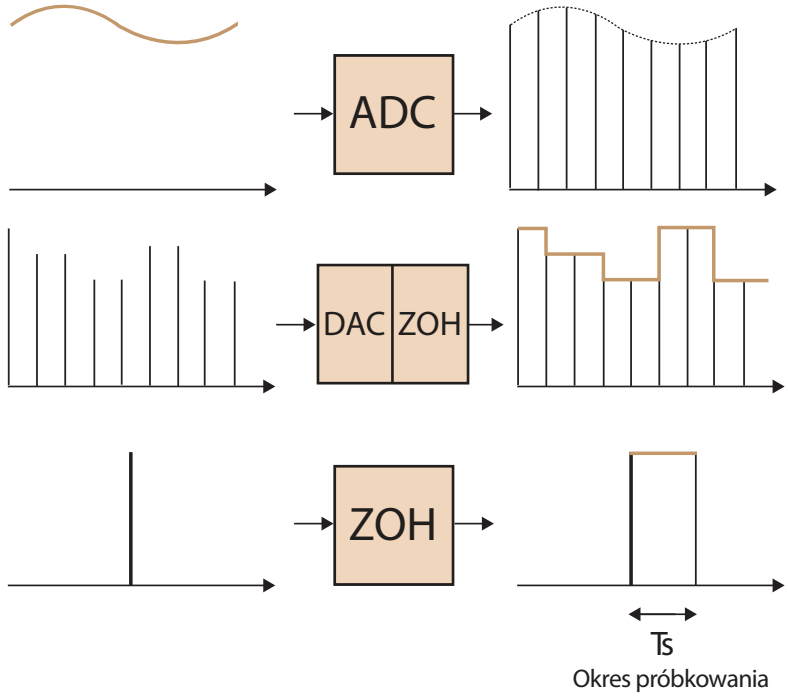


Rys. 3: Zero Order Hold – sygnał analogowy odtworzony z cyfrowego

Przetwarzanie w drugą stronę realizuje przetwornik cyfrowo–analogowy *DAC* z układem *ZOH* podtrzymującym napięcie do czasu następnego przetworzenia.

Różnica pomiędzy wartością zadaną, a wyjściem obiektu $e(k) = y^{zad}(k) - y(k)$ to **uchyb regulacji**. Kolejnym warunkiem, jaki musi spełnić układ regulacji jest wymaganie, aby uchyb regulacji pozostawał dostatecznie mały także przy niewielkich zmianach dynamiki obiektu. Proces utrzymywania małego uchybu regulacji to **proces regulacji**. O procesie regulacji, który zapewnia mały uchyb w warunkch występowania zakłóceń obiektu i szumów pomiarowych mówimy, że ma dobrą **kompensację zakłóceń** (*disturbance rejection*). O systemie, który zapewnia dobrą regulację pomimo zmian parametrów dynamiki obiektu, mówimy, że ma małą wrażliwość na zmiany tych parametrów. System, który łączy obie te cechy, to znaczy że zapewnia dobrą kompensację zakłóceń przy jednoczesnej małej wrażliwości na zmiany dynamiki obiektu, to **system odporny** (*robust*).

Regulację odporną można uzyskać w zamkniętym układzie regulacji: wyjście obiektu $y(t)$, po przetworzeniu na $y(k)$ jest porównywane z wartością zadaną $y^{zad}(k)$. Zgodnie z zastosowanym prawem regulacji, na wyjściu regulatora jest generowany sygnał sterujący $u(k)$, który po przetworzeniu na $u(t)$, stanowi wymuszenie obiektu. Idea sprzężenia zwrotnego jest realizowana od początków automatyki. Regulację cyfrową wyróżnia stosowanie cyfrowego przetwarzania sygnałów pomiędzy wyjściem obiektu, a jego wejściem. To oznacza, że sygnał sterujący powstaje w wyniku realizacji odpowiedniego programu w procesorze (mikrokontrolerze). Regulowany proces jest ciągły, to znaczy opisują go sygnały analogowe, czyli ciągłe w czasie i ciągłe w zakresie swoich wartości. Procesory działają w dziedzinie czasu dyskretnego i na dyskretnych wartościach. Musi więc nastąpić dyskretyzacja sygnału wyjściowego $y(t)$ zarówno w czasie jak i dyskretyzacja jego wartości (rysunek 2). Wykonuje to **przetwornik analogowo - cyfrowy** ADC (*analog-to-digital converter*), który zamienia sygnał analogowy w ciąg liczb.



Rys. 4: Przetwarzanie ADC i DAC

Typowy regulator cyfrowy ma stały **okres próbkowania** T_s (*sample period*). Jest to najczęściej realizowane za pomocą przerwań systemowych. W procesorach Cortex-M można do tego celu wykorzystać wewnętrzny licznik SysTick. W równych odstępach czasu T_s wykonywane jest przetwarzanie analogowo - cyfrowe, czyli mierzona wartość wyjścia obiektu $y(t)$ jest przetwarzana na liczbę $y(k)$, gdzie k symbolizuje bieżącą chwilę w dziedzinie czasu dyskretnego. Rysunek 2 pokazuje przykład działania przetwornika ADC 3-bitowego. Możliwe wartości liczbowe na wyjściu tego przetwornika to (binarnie) `000, 001, 010, 011, 100, 101, 110, 111`. Przetwornik w chwilach czasowych oznaczonych numerami od $k = 0$ do $k = 10$ zamienia wartość chwilową sygnału na najbliższą liczbę. Mikrokontroler otrzymuje liczbę $y(k)$, może wykonać odejmowanie $e(k) = y^{zad}(k) - y(k)$ i dalsze przetwarzanie danych. W wyniku działania algorytmu regulacji wyliczana jest wartość $u(k)$ która, przez przetwornik cyfrowo - analogowy, zostaje zamieniona na odpowiadające jej napięcie $u(t)$. To napięcie będzie podtrzymywane na stałej wartości przez układ ZOH *Zero Order Hold*, przez cały następny okres próbkowania T_s, czyli aż do wyliczenia nowej wartości sterowania $u(k+1)$ w następnym kroku działania regulatora. Zastosowanie ZOH powoduje, że **układ regulacji z rysunku 1 pracuje w otwartej pętli regulacji w czasie pomiędzy kolejnymi próbkami, gdyż sygnał sterujący $u(t)$ pozostaje stały bez względu na wartość wyjścia obiektu $y(t)$.**

Rysunek 4 przedstawia działanie przetwornika cyfrowo - analogowego i połączonego z nim układu ZOH. Jeżeli założymy, że $u(k) = y(k)$, to możemy porównać oryginalny sygnał analogowy z rysunku 2 z sygnałem, który powstał jako wynik przetwarzania analogowo - cyfrowego, a następnie cyfrowo - analogowego z podtrzymaniem ZOH. Dużą rozbieżność pomiędzy sygnałami można zmniejszyć stosując przetwornik o większej ilości bitów, na przykład Cortex-M3 jest wyposażony w przetwornik 12-bitowy, który zapewnia podział zakresu przetwarzania na 4096 części (3-bitowy przetwornik z przykładu

dzieli zakres na 8 części). Poprawę odwzorowania dynamiki sygnału uzyskamy zwiększając częstotliwość próbkowania, czyli zmniejszając okres T_s. O tym jak często trzeba próbkować określony sygnał analogowy mówi Twierdzenie o Próbkowaniu (twierdzenie Whittakera-Nyquista-Kotielnikova-Shannona).

Zagadnienia próbkowania sygnałów analogowych, przetwarzania cyfrowych i zamiany wyniku na sygnały analogowe (rysunek 4) są przedmiotem dziedziny wiedzy zwanej DSP *Digital Signal Processing*, czyli cyfrowego przetwarzania sygnałów.

Płyty rozwojowe używane w książce

Projekty opisane w książce wykorzystują zestawy uruchomieniowe (*evaluation boards*):

- ZL27ARM, procesor STM32F103VB, Cortex-M3
- STM3210E-EVAL, procesor STM32F103ZE, Cortex-M3

Przykładowe programy dołączone do książki podzielone są na katalogi: każda płyta ma swój katalog. Wykaz projektów podany jest w tabeli 2

PROGRAMY DOŁĄCZONE DO KSIĄŻKI PRZEZNACZONE SĄ WYŁĄCZNIE DO CELÓW SZKOLENIOWYCH.
AUTOR NIE PONOSI ODPOWIEDZIALNOŚCI ZA ŻADNE EWENTUALNE, BEZPOŚREDNIE I POŚREDNIE SZKODY WYNIKŁE Z ICH WYKORZYSTANIA.

THE SOFTWARE INCLUDED IS FOR GUIDANCE ONLY.
AUTHOR SHALL NOT BE HELD LIABLE FOR ANY DIRECT, INDIRECT OR CONSEQUENTIAL DAMAGES WITH RESPECT TO ANY CLAIMS ARISING FROM USE OF THIS SOFTWARE.

Programy są pisane w taki sposób, aby podkreślić omawiane zagadnienia. Nie mają wstawionych zabezpieczeń i nie realizują wymaganych algorytmów. Na przykład programy transmisji szeregowej USART nie są zabezpieczone na wypadek zerwania transmisji w trakcie wymiany danych – program nie wykryje takiej sytuacji i będzie czekał na kolejne bajty. Drugi przykład: nie ma pełnej obsługi protokołu MODBUS: rozpoznawania funkcji, numerów rejestrów, itd. Program jest tak napisany, żeby doszło do wymiany danych w jednym, określonym przypadku.

Ta uwaga dotyczy **wszystkich** programów dołączonych do książki.

Tabela 2: Wykaz projektów dołączonych do książki

ZL27ARM	**strona**	**STM3210E-EVAL**	**strona**
Projekt01_LED	69	Projekt01_LED	189
Projekt02_LCD	122		
Projekt03_asm	138	Projekt03_asm	
Projekt04_SysTick	149		
Projekt05_keypad	155		
Projekt06_scanf	159		
Projekt07_RTC	162		
Projekt08_TIM4-LED	166		
Projekt09_TIM2switch	172		
Projekt10_RadioControl	175		
Projekt11_6WheelDrive	180		
Projekt12_DistanceSensor	184		
Projekt13_PDC			
		Projekt14_TFTLCD	189
Projekt15_StepperMotor	201	Projekt15_StepperMotor	201
Projekt16_StepperMotor	211	Projekt16_StepperMotor	211
		Projekt17_FixedPoint_sin	230
		Projekt18_cube3D	232
		Projekt19_cube3D_FixedPoint	238
Projekt20_DMA_M2M	244	Projekt20_DMA_M2M	244
Projekt21_ADC1	253	Projekt21_ADC1	253
Projekt22_Pt100	258	Projekt22_Pt100	258
		Projekt23_DAC_sin	268
		Projekt24_1_Modbus_Master	278
		Projekt24_2_Modbus_Master	284
Projekt25_Modbus_Slave	282	Projekt25_Modbus_Slave	282
		Projekt26_I2C	290
Projekt27_SPI	292		
		Projekt28_obiekt_xxxx	296
		Projekt29_PI	303
		Projekt30_PID	307
		Projekt31_1_pole_placement	325
		Projekt31_2_pole_placement	333
		Projekt31_3_pole_placement	334
Projekt32_MultiTasking	339	Projekt32_MultiTasking	339

Nazewnictwo (terminologia)

W latach siedemdziesiątych i osiemdziesiątych ubiegłego wieku, czyli w czasach gwałtownego rozwoju techniki mikroprocesorowej, w Polsce wydawano stosunkowo dużo książek o układach logicznych (TTL i ECL) oraz o mikroprocesorach. Wielu autorów starało się tłumaczyć na język polski takie terminy angielskie jak *NAND gate*, *NOR gate*, czy *XOR gate*. Były to najczęściej "brama NIEI", "brama NIELUB", "brama ALBO". Jednocześnie młody inżynier (autor) korzystał z oryginalnych amerykańskich katalogów. Próby tłumaczenia wszystkiego na język polski sprawiały, że często książki polskich autorów stawały się nieczytelne. Po latach ustaliło się polskie nazewnictwo, mówimy teraz bramka NAND, bramka NOR i XOR. Nawet prosta zamiana oznaczenia zmiennej zespolonej w transmitancji operatorowej członu oscylacyjnego z s na p

$$G_o(p) = \frac{\omega_n^2}{p^2 + 2\zeta\omega_n p + \omega_n^2}$$

powoduje, że tekst jest trudny w odbiorze. A zamiana z s na z prowadzi wręcz do nieporozumień (z to zmienna transformaty dyskretnej $\mathcal{Z}$).

Dla czytelności przekazu, ważne jest stosowanie zwyczajowych oznaczeń i powszechnie przyjętego nazewnictwa. Materiały źródłowe producentów mikrokontrolerów są wydawane w języku angielskim. Czytelnik tej książki będzie musiał z nich korzystać. Dlatego, dla jasności przekazu warto jest zachować oryginalną (czyli angielską) terminologię. Ponadto większość nazw angielskich jest powszechnie używana w polskim świecie techniki wbudowanej (*embedded*). Dlatego, na przykład (szczególnie na rysunkach) zamiast oznaczenia **PoDS** ("Pamięć o Dostępie Swobodnym"), będzie stosowany opis *Random-Access Memory* lub samo RAM. Nazwy angielskie są wyróżnione *kursywą* (ang. *italic*).

Kolejna konwencja dotyczy adresowania pamięci na rysunkach: z dołu do góry, czy odwrotnie. Przyjęte jest, że mapa pamięci jest rysowana z adresami rosnącymi do góry, czyli najmniejsze są na dole, jak na rysunku 3.8. Jednak listing programu będziemy drukować odwrotnie, to znaczy od góry na dół, jak w tablicy 3.1. Pozwala to czytać program naturalnie, jak tekst w książce.

Ze względu na standard języka C i asemblera, w celu zachowania jednolitości oznaczeń, w całej książce będziemy stosować kropkę dziesiętną i kropkę binarną zamiast przecinka. Na przykład liczbę Π zapiszemy

$$\Pi = 3.14159265$$

Przedrostki dwójkowe i przedrostki SI

W całej dziedzinie komputerowej występuje przemieszanie przedrostków dwójkowych (G = 2^{30}) i dziesiętnych stosowanych w systemie SI (G = 10^9). Tak więc producenci dysków twardych podają pojemność w wartościach dziesiętnych, a systemy operacyjne w dwójkowych, przez co dysk 10GB w systemie operacyjnym pokazuje jedynie 9.3GB.

W związku z problemami związanymi z wymieszaniem przedrostków dwójkowych i tych z układu SI, w roku 1998 Międzynarodowa Komisja Elektrotechniczna (IEC) zaproponowała używanie alternatywnego nazewnictwa (tabela 3):

Tabela 3: Przedrostki dwójkowe IEC

nazwa	prefix	rozmiar
kibi	Ki	$2^{10} = 1'024$
mebi	Mi	$2^{20} = 1'048'576$
gibi	Gi	$2^{30} = 1'073'741'824$
tebi	Ti	$2^{40} = 1'099'511'627'776$

Niestety ten format nazw nie przyjmuje się zbyt szybko, przez co występują problemy, gdy jedna strona stosuje przedrostki układu SI, a druga spodziewa się przedrostków dwójkowych. Jednym ze skutków jest fakt, że początkujący programista uważa, iż jeden kilobajt to tysiąc bajtów, a zaawansowany, że kilometr to 1024 metry.

W książce, podobnie jak to jest w systemie Windows, będziemy stosować system binarny, ale oznaczać, jakby to był SI, (tabela 4):

Tabela 4: Przedrostki stosowane w książce

nazwa	prefix	rozmiar
kilo	k	$2^{10} = 1'024$
Mega	M	$2^{20} = 1'048'576$
Giga	G	$2^{30} = 1'073'741'824$
Tera	T	$2^{40} = 1'099'511'627'776$

Rozdział 1

Historia architektury ARM

1.1. Acorn - początki

Jest rok 1979. Atari stał się liderem rynku gier komputerowych. Powstaje język programowania ADA. Powstają firmy 3Com, Oracle i Seagate. TI wchodzi na rynek komputerów. Dennis C. Hayes opracował swój pierwszy modem, który stał się standardem światowym. Rozpoczęto produkcję procesorów Motorola 68k i Intel 8088. Ponadto Hermann Hauser i Chris Curry, z pomocą grupy studentów i pracowników uniwersytetu Cambridge zakładają firmę Acorn Computers. Celem firmy jest produkowanie komputerów osobistych w Cambridge, UK.

Pierwszym produktem jest komputer Atom z procesorem 1MHz i pamięcią ROM+RAM 12kB. Firma Acorn rozpoczyna współpracę z British Broadcasting Corporation (BBC) w zakresie produkcji i sprzedaży komputera domowego BBCmicro. Ale świat komputerów domowych nie stoi w miejscu. Apple rozpoczyna sprzedaż komputera Lisa, który łączy pierwszy system okien (*windows*) z procesorem 16-bitowym. Dla ludzi z Acorn staje się jasne, że muszą stworzyć procesor o znacznie większej wydajności w stosunku do powszechnych w tamtym czasie procesorów 8-bitowych. Powstaje sekcja Acorn's Advanced Research and Development, w której jest realizowany projekt procesora z ograniczoną listą instrukcji *Reduced Instruction Set Computing (RISC)*. W tamtych czasach, przewagi dużych komputerów stacjonarnych, była to idea rewolucyjna.

1.2. Narodziny procesorów ARM

Wynikiem badań był pierwszy procesor ARM, czyli Acorn RISC Machine. Nazwa ta została później zmieniona na **Advanced RISC Machine**. Pierwsze próbki pojawiły się w 1985 roku. ARM1 szybko został zastąpiony przez ARM2 - jego wersję produkcyjną. Był to prawdopodobnie najprostszy procesor RISC tamtych czasów - tylko 30'000 tranzystorów. Zawierał 32-bitową magistralę danych, 26-bitową magistralę adresową i 16 rejestrów wewnętrznych. W połowie 1987 roku Acorn zaprezentowało swój komputer klasy PC (*Personal Computer*) Archimedes, jednak nie osiągnął on sukcesu rynkowego. Wtedy rozwijał się już standard IBM PC i nowy komputer z własnym systemem operacyjnym, bez oprogramowania użytkowego nie był atrakcyjny dla potencjalnych klientów. Jednak Acorn kontynuowało prace rozwojowe i w roku 1990 pojawił się ARM3 z pamięcią *Cache* 4kB na pokładzie i zegarem 25MHz.

Konstruktorzy ARM zaprojektowali wersję statyczną procesora, dzięki temu osiągnęli znaczną redukcję mocy po zatrzymaniu zegara. Umożliwiło to konstruowanie urządzeń zasilanych z baterii *personal digital assistant (PDA)*. Rynkiem PDA była zainteresowana firma Apple. Postanowiono połączyć siły, ale Apple postawiło warunek, że z firmy Acorn zostanie wydzielony dział rozwojowy R&D. Zostało wynegocjowane porozumienie pomiędzy firmami Apple, VLSI Technology i Acorn.

1.3. Advanced RISC Machines Ltd.

W roku 1990, 27 listopada, powstała firma Advanced RISC Machines Ltd. (ARM). Jej dyrektorem generalnym (*Chief Executive Officer, CEO*) został Sir Robin Keith Saxby. Celem nowej firmy stał się rosnący rynek procesorów 32-bitowych RISC tanich, niskiej mocy, ale o wysokiej wydajności. ARM powstał jako partner w tworzeniu wszystkich nowych produktów Apple'a. Procesory ARM montowane są w prawie wszystkich współczesnych telefonach komórkowych, palmtopach i innych urządzeniach mobilnych takich producentów jak: Nokia, Sony Ericsson, Samsung, Apple, Nintendo oraz wielu innych. W roku 2005 ARM kupił firmę KEIL Software.

ARM nie produkuje mikrokontrolerów. ARM projektuje procesory oraz komponenty, które są potrzebne projektantom krzemu i sprzedaje licencje różnym firmom produkującym mikrokontrolery. ARM nazywa te projekty własnością intelektualną (*Intellectualy Property IP*), a model biznesu to sprzedaż licencji (*licensing IP*).

W typowym mikrokontrolerze, sam procesor zajmuje małą część powierzchni krzemu. Pozostała część to pamięci, układy taktujące (np. PLL), magistrale systemowe i peryferiale (układy peryferyjne), takie jak układy wejścia/wyjścia (GPIO, UART), liczniki (*timers*), przetworniki ADC, DAC i inne.

Wielu producentów mikrokontrolerów używa procesorów **ARM Cortex-M** w swoich projektach (*cortex* to kora, np. mózgowa, ale też na drzewie). Natomiast bloki pamięci i układy peryferyjne są dodawane indywidualnie. To pozwala każdemu producentowi mikrokontrolerów tak zaprojektować swój układ, aby wyróżnić się na rynku.

1.4. ARM i sprzedawcy mikrokontrolerów

Obecnie na rynku jest siedemnastu producentów mikrokontrolerów używających mikroprocesory Cortex-M3, Cortex-M4 lub Cortex-M7 w swoich produktach. Są to: Analog Devices, Atmel, Cypres, EnergyMicro, Freescale, Fujitsu, Holtek, Infineon, Microsemi, Milandr, NXP, Samsung, Silicon Laboratories, ST Microelectronics, Texas Instrument i Toshiba.

Gdy firma wykupi licencję IP na projekt Cortex-M, ARM dostarcza kody źródłowe procesora w języku Verilog-HDL (*Hardware Description Language*). Inżynierowie firmy-klienta dodają swoje własne bloki pamięci, magistrale, peryferiale i tworzą ostateczną postać mikrokontrolera.

ARM dostarcza także zintegrowane środowisko programistyczne IDE (*Integrated Development Environment*) Keil **Microcontroller Development Kit (MDK-ARM)**. Obecna wersja to μVision 5. Ponieważ wszystkie mikrokontrolery Cortex-M mają ten sam procesor, projektanci urządzeń mogą używać jednego środowiska IDE do mikrokontrolerów produkowanych przez różnych dostawców z zachowaniem możliwości łatwego przenoszena programów.

1.5. Zalety procesorów Cortex-M

1.5.1. Niski pobór energii

W porównaniu do innych procesorów 32-bitowych, Cortex-M jest mały i zaprojektowany na niski pobór energii (*low power*). Większość procesorów Cortex-M ma pobór rzędu

200μA/MHz, a niektóre poniżej 100μA/MHz. Ponadto są dostępne tryby pracy oszczędzające energię, takie jak uśpienie (*sleep mode*) i głębokie uśpienie (*deep sleep mode*), zatrzymanie (*stop mode*) oraz czuwanie (*standby mode*).

1.5.2. Moc obliczeniowa

Procesory Cortex-M3 i Cortex-M4 mają ponad 3 CoreMark/MHz i 1.25 DMIPS/MHz. Cortex-M7 to 5 CoreMark/MHz i 3.23 DMIPS/MHz. Dzięki temu są w stanie obsłużyć wiele skomplikowanych i wymagających aplikacji. W przypadku mniejszych wymagań można obniżyć częstotliwość taktowania w celu dalszej redukcji poboru mocy. Szczegółowe informacje o sposobie określania mocy obliczeniowej zawiera nota aplikacyjna 350 "CoreMark Benchmarking for ARM Cortex Processors" dostępna na stronie firmy ARM http://infocenter.arm.com/

CoreMark jest programem komputerowym opracowanym w 2009 roku przez *Embedded Microprocessor Benchmark Consortium*. Zastępuje dawny test Dhrystone z 1984 roku. CoreMark zawiera kilka często używanych fragmentów kodu, takich jak operacje na macierzach, wykorzystanie wskaźników, rozejścia programowe (skoki warunkowe) i liczenie CRC *Cyclic Redundancy Check* (jest to system sum kontrolnych, zapisywany na końcu bloku danych, który umożliwia kontrolę poprawności transmisji i zapisu/odczytu bloku danych). Test jest dostępny na stronie http://www.coremark.org/ i zgodny z ANSI C.

DMIPS to *Dhrystone Million Instructions Per Second*. Czyli 1.25 DMIPS/MHz oznacza, że procesor wykonuje średnio 1.25 instrukcji na takt zegara (zgodnie z testem *Dhrystone*), co jest możliwe dzięki przetwarzaniu potokowemu. Dla porównania, procesor Intel 386Ex potrzebuje do wykonania jednej instrukcji od 2 do 98 taktów zegara. Instrukcja przełączania zadań *TSS* to 309 taktów. Dawniej w celu określenia wydajności procesora stosowano metrykę MIPS, czyli *Million Instructions Per Second*. Niestety, jako że nigdzie nie zdefiniowano jakie instrukcje mają być wykorzystane w tym wyliczeniu, często zdarzało się że producenci podawali wyniki uwzględniając jedynie instrukcje najszybsze (NOP, czyli instrukcja nie robiąca absolutnie nic, była popularnym kandydatem). Doprowadziło to do powstania alternatywnego rozwinięcia skrótu MIPS: *Meaningless Information Provided by Salesmen*. Dopiero wprowadzenie standardu *Dhrystone* umożliwiło porównywanie różnych procesorów. Metryka DMIPS dotyczy operacji na liczbach całkowitych. Liczba operacji zmiennoprzecinkowych na sekundę to **FLOPS** *FLoating point Operations Per Second*.

1.5.3. Efektywność energetyczna

Połączenie niskiego poboru energii i dużej mocy obliczeniowej daje wysoką efektywność energetyczną. To oznacza, że można szybko wykonać obliczenia i przejść do trybu *sleep mode* na dłuższy okres czasu. W układach zasilanych bateryjnie uzyskamy w ten sposób

impulsowy pobór energii. Typowe baterie pracują lepiej z impulowym poborem energii i długim czasem przerwy, gdyż następuje wtedy częściowa“regeneracja” baterii.

1.5.4. Gęstość upakowania kodu

Zestaw instrukcji Thumb2 ISA *Instruction Set Architecture* zapewnia bardzo dobrą gęstość kodu (*code density*), dzięki ich wysokiej efektywności. To oznacza, że do wykonania tego samego zadania potrzebujemy mniej kodu maszynowego, czego efektem jest krótszy czas wykonania.

1.5.5. Przerwania

Procesory Cortex-M są zintegrowane z układem obsługi przerwań NVIC *Nested Vectored Interrupt Controller*. Procesor potrzebuje tylko 12 taktów zegara na pobranie wektora przerwań i wywołanie odpowiedniej obsługi przerwania. Dzięki temu procesory Cortex-M bardzo dobrze sprawdzają się w regulacji i sterowaniu procesów czasu rzeczywistego.

1.5.6. Łatwe w użyciu, zorientowane na język C

Procesory Cortex-M są bardzo łatwe w użyciu, często łatwiejsze niż niektóre kontrolery 8-bitowe. Jest tak, ponieważ Cortex-M ma liniową przestrzeń pamięci z wydzielonymi obszarami instukcji, danych i rejestrów sterujących peryferialami. Można zaprogramować niemal wszystko w C, łącznie z obsługą przerwań. Wiele instrukcji maszynowych jest prostym odpowiednikiem instrukcji języka C.

1.5.7. Skalowalność

Rodzina Cortex-M zawiera układy od najmniejszych mikrokontrolerów w cenie poniżej jednego dolara do największych, pracujących z zegarem 216 MHz. Są też układy wielordzeniowe. Dzięki wspólnej architekturze oraz jednakowemu sposobowi konfiguracji i dostępu do peryferiali uzyskujemy możliwość łatwego przenoszenia kodu pomiędzy systemami różnej wielkości.

1.5.8. Uruchamianie programów

Zgodnie z tradycją informatyki, błąd w programie to *bug*, czyli pluskwa. Narzędzia do wyszukiwania błędów to debugery. Procesory Cortex-M są wyposażone w moduł *Embedded Trace Macrocel* ETM, który jest bardzo efektywnym narzędziem do śledzenia wykonywania programów. Wykorzystamy to w rozdziale 13.4.1 do analizy kodu.

1.5.9. RTOS

Procesory Cortex-M zostały zaprojektowane z myślą o stosowaniu w nich systemu operacyjnego czasu rzeczywistego RTOS *Real Time Operating System*. Są dostępne dwa tryby pracy: *Privileged* w którym pracuje RTOS i *Unprivileged* z ograniczonym dostępem do zasobów, dla aplikacji pracujących pod nadzorem RTOS. Każdy z tych trybów pracy ma własny wskaźnik stosu. Obecnie dostępnych jest około 30 systemów RTOS przeznaczonych dla Cortex-M.

1.5.10. CMSIS

Jedną z inicjatyw firmy ARM jest CMSIS (*Cortex Microcontroller Software Interface Standard*). Jest to standardowa biblioteka plików nagłówkowych i źródłowych C do obsługi wszystkich zasobów rdzenia Cortex-M wraz ze zintegrowanymi układami peryferyjnymi.

Rozdział 2

Reprezentacja liczb całkowitych

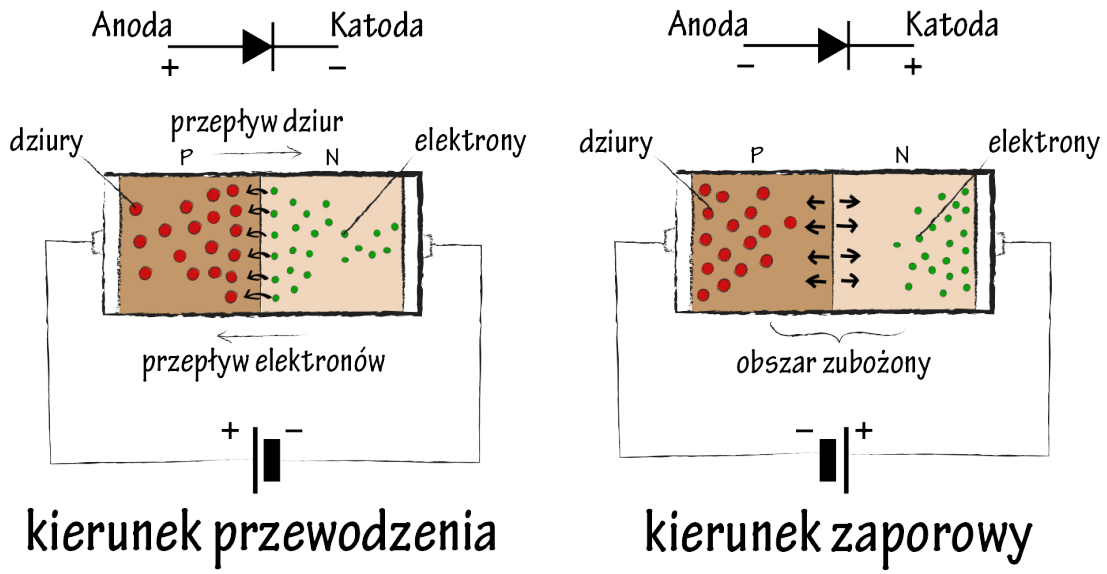
Zaczniemy od kilku podstawowych informacji z zakresu elektroniki.

2.1. Półprzewodniki

Mikrokontrolery są wytwarzane w krysztale krzemu, najczęściej w postaci płaskiej płytki, którą nazywamy *chip*em. Zawartość krzemu w zewnętrznych warstwach Ziemi to 27% (wagowo). Jest drugim, po tlenie najbardziej rozpowszechnionym pierwiastkiem. Krzem SI należy do grupy półmetali. Ma liczbę atomową 14 = 2+8+4, to znaczy że licząc od środka, na pierwszej powłoce ma 2 elektrony, na następnej 8 i na zewnętrznej 4. Te 4, to są elektrony walencyjne, czyli peryferyjne. Uczestniczą one w tworzeniu wiązań chemicznych. Półprzewodnik o określonych własnościach uzyskujemy przez domieszkowanie, czyli część atomów krzemu w sieci krystalicznej zastępujemy innymi. Jeżeli są to atomy fosforu P (gr. *phosphoros* - niosący światło), pierwiastek, niemetal, zapełnienie powłok 15 = 2+8+5, to jego 4 elektrony walencyjne utworzą wiązania z czterma elektronami sąsiednich atomów krzemu, a jeden pozostanie wolny. Wprowadzenie domieszki dającej nadmiar elektronów powoduje powstanie **półprzewodnika typu n** (*negative electric charge*), a domieszka jest nazywana donorową ("oddaje elektron").

Następnie do czystego kryształu kwarcu dodamy bor - pierwiastek z grupy półmetali o liczbie atomowej 5 = 2+3. Gdy trzy elektrony walencyjne boru zwiążą się z sąsiednimi elektronami krzemu, zabraknie jednego elektronu. Powstanie **półprzewodnik typu p** (*positive electric charge*), a domieszka jest nazywana akceptorową ("przyjmuje elektron"). Takie wolne miejsce po brakującym elektronie to dziura (*hole*). Zachowuje się ona jak swobodna cząstka o ładunku dodatnim i jest zdolna do przewodzenia prądu. Ładunek dodatni wynika z braku zbalansowania ilości protonów i elektronów.

Jeżeli połączymy oba typy półprzewodników, to uzyskamy złącze *p*-*n*, czyli diodę. Gdy dioda jest podłączona w kierunku przewodzenia (Anoda + i Katoda -), to elektrony (o ładunku ujemnym) są przyciągane przez plus, a dziury przez minus baterii (rysunek 2.1). Elektrony i dziury pokonują barierę złącza i prąd płynie, czyli dioda przewodzi prąd. Gdy dioda jest spolaryzowana odwrotnie (A- i K+), to elektrony i dziury odsuwają się od złącza, powstaje obszar zubożony, czyli bez swobodnych nośników ładunku (ten obszar staje się izolatorem) i prąd nie płynie. Wtedy mówimy, że dioda jest spolaryzowana zaporowo. Dioda działa więc jak zawór jednokierunkowy. Żeby spowodować przepływ prądu, należy przyłożyć w kierunku przewodzenia pewne minimalne napięcie, które pomoże ładunkom pokonać barierę złącza. To napięcie nada elektronom energię na tyle dużą, że będą mogły pokonać siły atomowe trzymające je po stronie *n*. Dla diod krzemowych jest to od 0.6 do 0.7 Volta. Napięcie to zależy głównie od temperatury złącza.



Rys. 2.1: Polaryzacje diody półprzewodnikowej

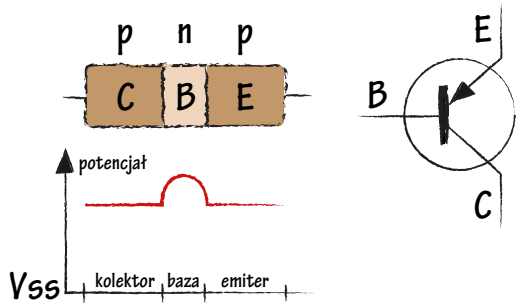
Light - Emitting Diode, czyli diody świecące LED. Zasada działania opiera się na zjawisku rekombinacji nośników ładunku (rekombinacja to połączenie dziury z elektronem, wtedy oba ładunki "znikają"). Zjawisko to zachodzi w półprzewodnikach wówczas, gdy elektrony przechodząc z wyższego poziomu energetycznego na niższy zachowują swój pęd. Podczas tego przejścia energia elektronu zostaje zamieniona na kwant promieniowania elektromagnetycznego, czyli światła. LED spolaryzowany w kierunku przewodzenia świeci, a w kierunku zaporowym nie. Natężenie światła (mierzone w kandelach) zależy prawie proporcjonalnie od prądu i w przypadku typowych, małych LEDów prąd przewodzenia nie powinien przekraczać 20 mA. Musimy pamiętać, że **dioda jest elementem prądowym**, a więc układ w którym pracuje dioda musi zapewniać przepływ prądu o określonej wartości. Napięcie przewodzenia zależy od materiału z którego dioda LED została wykonana i temperatury złącza. Parametry kilku typowych diod podane są w tabeli 2.1

Tabela 2.1: Parametry diod świecących LED

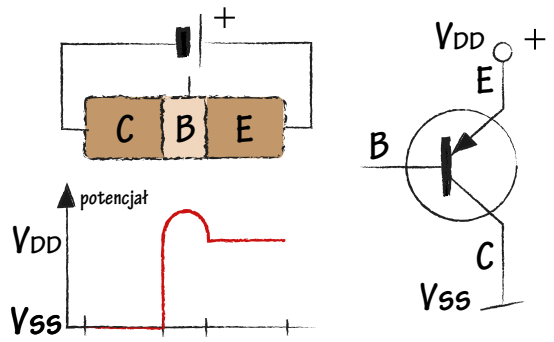
długość światła [nm]	**kolor**	**napięcie V_{LED} [V]**	**materiał**
940	Infrared	1.5	GaAlAs/GaAs
635	High Efficiency Red	2.0	GaAsP/GaP
633	Super Red	2.2	InGaAIP
612	Super Orange	2.2	InGaAIP
605	Orange	2.1	GaAsP/GaP
595	Super Yellow	2.2	InGaAIP
585	Yellow	2.1	GaAsP/GaP
565	High Efficiency Green	2.1	GaP/GaP
525	Aqua Green	3.5	SIC/GaN
505	Blue Green	3.5	SIC/GaN
470	Super Blue	3.6	SIC/GaN
430	Ultra Blue	3.8	SIC/GaN
	Pale White	3.6	SIC/GaN
	Cool White	3.6	SIC/GaN

Tranzystor bipolarny to połączenie dwóch złącz *p-n*. Mamy dwie możliwości *p-n-p* oraz *n-p-n*. Zjawisko tranzystorowe opiszemy na przykładzie tranzystora *p-n-p*.

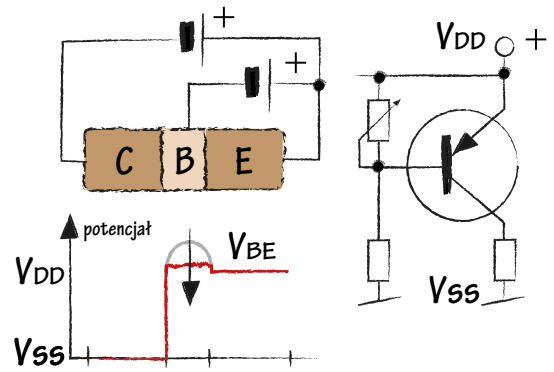
W stanie bez polaryzacji zewnętrznej dziury z emitera nie przenikają do bazy ze względu na barierę złącza. Ta bariera nie jest napięciem które samoistnie odkłada się na bazie. Rysujemy ją symbolicznie jako potencjał wewnętrzny, który trzeba pokonać polaryzując złącze z zewnątrz, żeby spowodować przepływ prądu.



Po przyłożeniu zewnętrznej różnicy potencjałów (czyli napięcia) pomiędzy emiter E i kolektor C (baza B pozostaje z niczym niepołączona, czyli "wisi w powietrzu") również nie będzie przepływu prądu, gdyż złącze baza-kolektor jest spolaryzowane zaporowo.



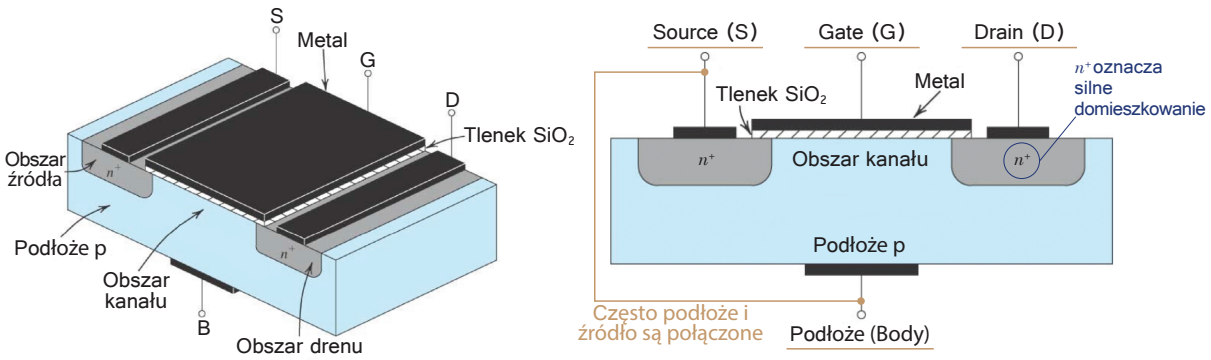
Jeżeli między bazę i emiter zostanie przyłożone napięcie V_{BE} zmniejszające barierę złącza, to dziury z emitera dostaną się do bazy, a następnie, o ile nie zrekombinują w niej, przedyfundują do kolektora, tworząc prąd I_C. Regulując napięcie V_{BE} regulujemy wysokość bariery potencjału na tym złączu, co przekłada się na ilość dziur dostających się do bazy i dalej do kolektora. Dzięki temu za pomocą sygnału elektrycznego dostarczanego do bazy kontrolujemy oporność między emiterem i kolektorem. Tak działa



TRANSfereable rez**ISTOR**.

Aby wystąpił efekt tranzystorowy (by dziury nie zrekombinowały w bazie), baza musi być odpowiednio cienka. **Czas rekombinacji dziur w bazie musi być znacznie dłuższy niż czas ich dyfuzji przez bazę**.

Działanie tranzystora *n-p-n* jest analogiczne. Kierunki napięć i prądów są odwrotne niż w przypadku *p-n-p*. Nośnikami prądu kolektora są elektrony. Pamiętajmy, że kierunek prądu jest odwrotny do kierunku ruchu elektronów i zgodny z kierunkiem ruchu dziur. Tranzystor, podobnie jak dioda jest elementem prądowym, to znaczy że sterujemy nim wstrzykując prąd w bazę, a nie zmieniając napięcie baza-emiter które jest w przybliżeniu stałe (zależy głównie od temperatury). Parametrem charakteryzującym tranzystor jest jego wzmocnienie prądowe $\beta = I_C/I_B$, które określa ile razy prąd kolektora jest większy od prądu bazy. Typowe tranzystory mają wzmocnienie od $\beta = 40$ do $\beta = 800$.

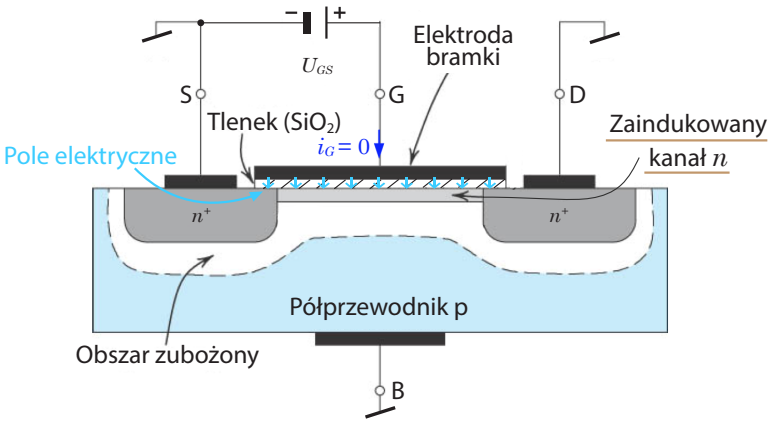


Rys. 2.2: Tranzystor unipolarny (polowy) MOSFET

Tranzystor polowy MOSFET *Metal-Oxide-Semiconductor, Field Effect Transistor*. Przekrój takiego tranzystora jest pokazany na rysunku 2.2. W podłożu – płytce słabo domieszkowanego półprzewodnika typu p albo n tworzone są dwa małe obszary o przeciwnym typie przewodnictwa – odpowiednio $n+$ lub $p+$ (plus oznacza silne domieszkowanie tych obszarów). Te silnie domieszkowane obszary tworzą dren *Drain D* oraz źródło *Source S* do których doprowadzane są kontakty. Powierzchnia półprzewodnika pomiędzy drenem i źródłem jest pokryta cienką warstwą izolatora, którym jest dwutlenek krzemu SiO_2 *oxide*. Grubość tej warstwy jest rzędu kilkunastu nanometrów. Na izolator napylana jest warstwa materiału przewodzącego (metalu) tworząca bramkę *Gate*. Ze względu na niewielką grubość warstwy izolacyjnej istnieje realne niebezpieczeństwo jej trwałego uszkodzenia (przepalenia) na skutek doprowadzenia z zewnątrz ładunku elektrostatycznego. Dlatego układy elektroniczne zawierające tranzystory MOS (na przykład mikrokontrolery) są przechowywane w foliach przewodzących. W fabrykach produkcja i serwis układów z mikrokontrolerami odbywa się w specjalnych strefach ochrony od ESD *Electrostatic Discharge*. Obok pokazany jest znak ochrony od ESD. Wykonując wszystkie ćwiczenia opisane w książce należy przestrzegać zasad ochrony od ESD. Ładunek elektrostatyczny, jaki gromadzi na sobie człowiek jest rzędu 40'000 Volt. Rozróżnia się dwa typy tranzystorów MOS:

- z kanałem zubożanym (z kanałem wbudowanym) – normalnie włączone, to jest takie, w których istnieje kanał przy zerowym napięciu bramka-źródło,
- z kanałem wzbogacanym (z kanałem indukowanym) – normalnie wyłączone, kanał tworzy się dopiero, gdy napięcie bramka-źródło przekroczy napięcie progowe U_T.

Rysunek 2.2 przedstawia tranzystor MOSFET z kanałem typu n (będziemy go oznaczać N-MOS) wzbogacanym. Bez napięcia polaryzującego bramkę G, pomiędzy źródłem S i drenem D nie ma przepływu prądu, ponieważ na drodze S↔D są złącza n-p-n i zawsze jedno z nich jest spolaryzowane zaporowo. Oporność S↔D jest rzędu $10^{12}\Omega$. Podobnie jak w przypadku tranzystorów bipolarnych, do wywołania przepływu prądu MOSFET potrzebuje pewnego minimalnego napięcia progowego U_T. Napięcie przykładamy do bramki, względem źródła U_{GS}, rysunek 2.3. Z powodu warstwy izolacyjnej pomiędzy bramką a



Rys. 2.3: Tranzystor MOSFET w stanie przewodzenia

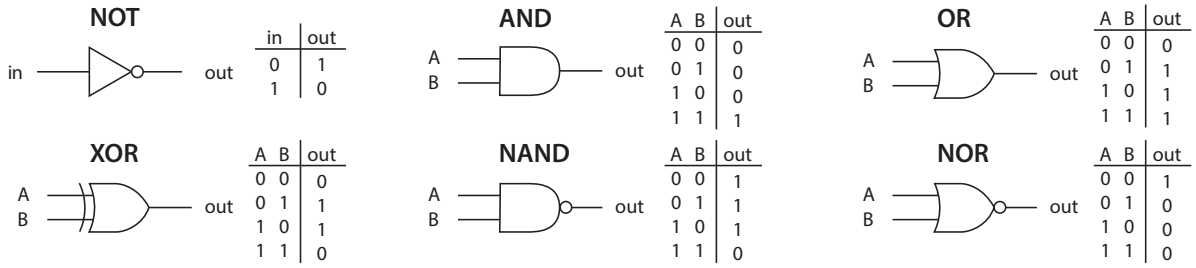
tranzystorem, prąd bramki jest zerowy. To oznacza, że **MOSFET jest elementem sterowanym napięciowo**. Bramka i region kanału tworzą kondensator. Napięcie $U_{GS} > U_T$ powoduje dwa zjawiska

- swobodne dziury półprzewodnika p są wypychane z obszaru pod bramką, ten proces otwiera obszar pod bramką dla ładunków ujemnych (elektronów),
- elektrony z silnie domieszkowanych obszarów $n+$ (drenu i źródła) są przyciągane pod bramkę.

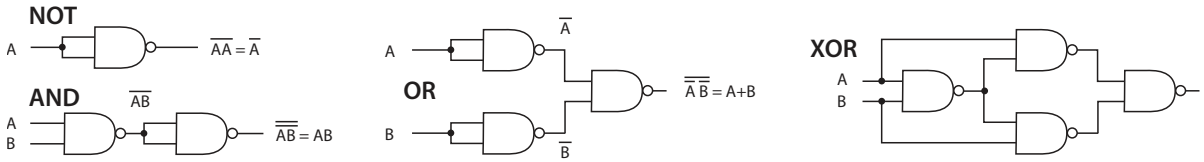
Oba te zjawiska prowadzą do utworzenia kanału typu n. Znikają złącza p-n i jest możliwy przepływ prądu pomiędzy źródłem i drenem. Napięcie progowe U_T (*threshold*) jest potrzebne do zgromadzenia dostatecznej ilości swobodnych elektronów, aby utworzyć kanał n: nie ma $U_{GS} \geqslant U_T$, nie ma kanału. Dla tranzystorów MOSFET z kanałem n $U_T \approx 1 \ldots 3\ V$. Po utworzeniu kanału, tranzystor MOSFET może być traktowany jako rezystor o oporności malejącej ze wzrostem napięcia na bramce. Tranzystory MOS są elementami bardzo szybkimi w porównaniu z tranzystorami bipolarnymi, gdyż zachodzące w nich zjawiska są czysto elektrostatyczne. Głównym czynnikiem zwiększającym czas przełączania jest obecność pojemności bramki, którą trzeba przeładować przy przełączaniu.

CMOS *Complementary Metal-Oxide Semiconductor*. Jest to technologia wytwarzania układów scalonych, głównie cyfrowych, składających się z tranzystorów MOS o przeciwnym typie przewodnictwa i połączonych w taki sposób, że w ustalonym stanie logicznym przewodzi tylko jeden z nich. Dzięki temu układ statycznie nie pobiera mocy, a prąd ze źródła zasilania płynie tylko w momencie przełączania. Tracona w układach CMOS moc wzrasta nieliniowo wraz z częstotliwością przełączania, co wiąże się z przeładowywaniem wszystkich pojemności, szczególnie pojemności obciążających wyjścia. Układy CMOS są relatywnie proste i tanie w produkcji umożliwiając uzyskanie bardzo dużych gęstości upakowania tranzystorów na jednostce powierzchni płytki krzemu.

Schemat wyjścia zrealizowanego w technologii CMOS pokazany jest na rysunku 4.20 (strona 65). Górny tranzystor z kanałem typu p załączany jest napięciem ujemnym, a dolny z kanałem typu n, napięciem dodatnim. Bit GPIO sterujący tym wyjściem równy 0 spowoduje podanie napięcia dodatniego, wyłączenie górnego tranzystora i włączenie dolnego, wyjście zostanie połączone z masą, czyli będzie wymuszony stan niski.



Rys. 2.4: Działania logiczne



Rys. 2.5: NAND: przykłady działań logicznych

2.2. Bit czyli najmniejsza jednostka informacji

Bit może przyjmować jedną z dwóch wartości

- 0 albo 1 - jest to cyfra binarna
- *False* albo *True* - wartość logiczna
- niski stan napięcia albo wysoki stan napięcia - sygnał binarny

Przyjmujemy konwencję, że równoważne sobie wzajemnie są

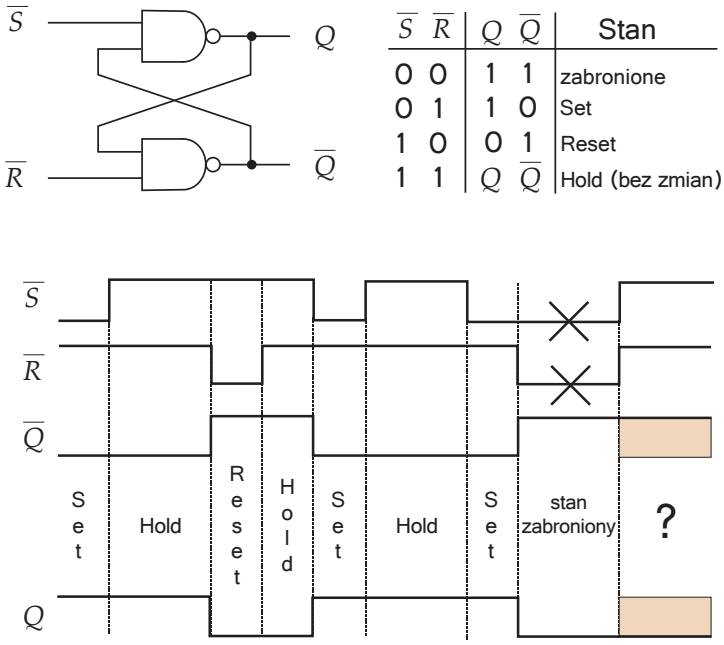
- $0 \iff$ *False* $\iff$ niski stan napięcia
- $1 \iff$ *True* $\iff$ wysoki stan napięcia

Dlatego będziemy mówić, że sygnał ma wartość logiczną 0, choć w rzeczywistości jest to na przykład napięcie 0.125 V. Zdefiniujemy też zmienne logiczne A i B, z których każda może przyjmować wartość 0 (*False*) albo 1 (*True*).

2.2.1. Działania logiczne

Zdefiniujemy działania logiczne na zmiennych A i B, takie jak negacja NOT, iloczyn AND, alternatywa *inclusive* OR (LUB) i alternatywa *exclusive* (czyli wykluczająca) XOR (ALBO). Negację zmiennej logicznej będziemy oznaczać kreską poziomą nad symbolem zmiennej $\bar{A}$, $\bar{B}$. Zdefiniujemy jeszcze dwa szczególne działania NAND = NOT(AND), czyli wykonujemy AND i negujemy wynik, oraz analogicznie NOR = NOT(OR). W systemach mikroprocesorowych działania logiczne na sygnałach wykonywane są za pomocą bramek logicznych *Logic Gates*. Definicje działań podane są na rysunku 2.4 NAND jest szczególne, gdyż mając tylko bramkę NAND możemy zrealizować wszystkie działania logiczne, przykłady na rysunku 2.5. Tą samą cechę ma NOR.

Wykorzystując dwie bramki NAND możemy zbudować przerzutnik, czyli element pamięciowy, rysunek 2.6. Przerzutnik ma dwa wejścia: ustaw *Set* i kasuj *Reset*. Oba sygnały są **aktywne niskie**, to znaczy że wymuszenie zerem spowoduje właściwe działanie. Sygnały aktywne niskie oznaczamy poziomą kreską nad symbolem. Zero na $\bar{S} = 0$ powoduje zapamiętanie jedynki $Q = 1$, zero na $\bar{R} = 0$ powoduje zapamiętanie zera $Q = 0$. Dwie



Rys. 2.6: Przerzutnik RS

jedynki $\bar{S} = 1$ i $\bar{R} = 1$, to stan *Hold*, czyli przerzutnik ("komórka pamięci") pamięta ostatnio zapisaną wartość. Dwa zera na wejściu są zabronione. Na rysunku 2.6 pokazane są przebiegi czasowe wejść i wyjść przerzutnika.

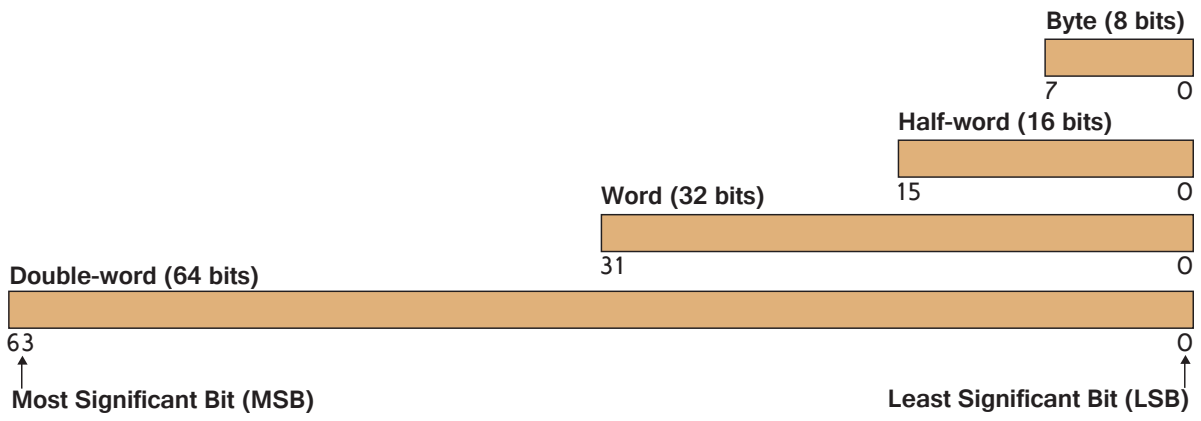
2.3. Reprezentacja Danych

Komputer przechowuje dane w postaci ciągu bitów. Te ciągi mają ustalone długości. Jak pokazano na rysunku 2.7 bajt *byte* jest grupą ośmiu bitów, połowa słowa *halfword* to 16 bitów (czyli 2 bajty), słowo *word* ma 32-bity (4 bajty) i podwójne słowo *double-word* 64 bity (8 bajtów). Najbardziej znaczący bit MSB *Most Significant Bit* to skrajny lewy (7, 15, 31, 63). Najmniej znaczący bit LSB *Least Significant Bit* to skrajny prawy (bit numer 0). Kompilator języka C **armcc** środowiska μVision używa następujących rozmiarów typów danych, tabela 2.2

Tabela 2.2: Typy danych kompilatora armcc μVision

typ języka C	**ilość bajtów w pamięci**
char	1 bajt
short	1 halfword (2 bajty)
signed /unsigned int	1 word (4 bajty)
signed /unsigned long	1 word (4 bajty)
signed /unsigned long long	1 double word (8 bajtów)
float	4 bajty
double	8 bajtów

Bajt jest najmniejszą porcją informacji jaka może być zapisana (odczytana) do (z) pamięci gdyż układy scalone pamięci mają 8 linii danych. Każdy bajt pamięci ma swój indywidualny adres. Wskaźnik *pointer* w programach C jest specjalną zmienną wskazującą



Rys. 2.7: Ilość bitów w bajcie, pół-słowie, słowie i podwójnym słowie Cortex-M

na adres bajtu w pamięci. W procesorach 32-bitowych rejestry mają 32 bity i przestrzeń adresowa też ma 32 bity, czyli rejestr może przechowywać adres pamięci (wskaźnik), a także każdą zmienną wymienioną w tabeli 2.2 oprócz obu typu *double*, które wymagają dwóch rejestrów.

Każdy adres pamięci to 8 bitów (bajt), czyli *halfword*, *word* i *double-word* zajmują (odpowiednio) 2, 4 i 8 kolejnych adresów. Zgodnie z konwencją przyjętą we wszystkich komputerach, jeżeli zmienna zajmuje kilka bajtów w pamięci, to procesor używa najmniejszego adresu do adresowania całej zmiennej. Na przykład, jeżeli zmienna typu `int` jest zapisana w czterech bajtach od adresu `0x20000000` do `0x20000003`, to adresem całej zmiennej jest `0x20000000`. Kolejność bajtów w pamięci jest nazywana *Big Endian* oraz *Little Endian* i jest opisana w rozdziale 7.2.

2.4. Liczby binarne, oktalne, dziesiętne i hexadecymalne

W życiu codziennym liczby całkowite przedstawiamy w zapisie dziesiętnym. Na przykład rok tysiąc czterysta dziesiąty zapiszemy

$$1 * 10^3 + 4 * 10^2 + 1 * 10^1 + 0 * 10^0 = 1410$$

Każda pozycja w zapisie ma wagę która jest potęgą podstawy zapisu. W powyższym przypadku podstawą jest 10. Do zapisu potrzebujemy tylu cyfr jaka jest wartość podstawy, czyli w zapisie dziesiętnym używamy dziesięciu cyfr $(0, 1, 2, \ldots, 9)$. W komputerze mamy tylko dwie cyfry $(0, 1)$, spróbujmy zapisać liczbę z podstawą równą ilości cyfr, czyli z podstawą 2. Wagi pozycji są więc potęgami dwójki

Tabela 2.3: Wagi pozycji reprezentacji binarnej

MSB											LSB
...	2^{10}	2^9	2^8	2^7	2^6	2^5	2^4	2^3	2^2	2^1	2^0
...	1024	512	256	128	64	32	16	8	4	2	1

Bitwa pod Grunwaldem odbyła się więc w roku

$$10110000010_2 \triangleq 2^{10} + 2^8 + 2^7 + 2^1 = 1024 + 256 + 128 + 2 = 1410$$

Liczby całkowite w komputerze zapisywane są binarnie. Jednak nie jest to zapis ani wygodny, ani czytelny, szczególnie gdy liczba jest duża. Dlatego, dla naszej wygody pogrupujemy (od końca) cyfry po trzy i każdej trójce przyporządkujemy cyfrę od 0 do 7. Cyfr jest osiem, otrzymamy więc zapis ósemkowy, czyli oktalny

$$10'110'000'010_2 \triangleq 2602_8 \triangleq 2*8^3+6*8^2+2*2^0 = 2*512+6*64+2 = 1024+384+2 = 1410$$

Zapis oktalny też nie spełnia naszych oczekiwań: podział po trzy cyfry nie pokrywa się z podziałem na bajty, więc nie jest zbyt czytelny. Dlatego pogrupujemy po cztery, zaczynając od końca, czyli od LSB. Każda czwórka to szesnaście możliwości: od `0000` do `1111`. Potrzebujemy więc szesnaście cyfr. Te cyfry to $0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, F$. Cyfry dziesiętne uzupełniliśmy więc literami.

$$101'1000'0010_2 \triangleq 582_{16} \triangleq 5*16^2+8*16^1+2*16^0 = 5*256+8*16+2 = 1280+128+2 = 1410$$

Zapisy powyższe są mylące, gdyż używamy w jednym równaniu różnych podstaw. Dlatego będziemy posługiwać się standardem zapisu literałów języka C

- **Literały dziesiętne** mogą zawierać cyfry od 0 do 9 i *nie mogą* zaczynać się od zera
- **Literały ósemkowe** mogą zawierać cyfry od 0 do 7 i *muszą* zaczynać się od zera
- **Literały szesnastkowe** mogą zawierać cyfry z przedziału od 0 do 9 i litery, czyli cyfry od A do F i *zawsze* zaczynają się od `0x`.

Teraz możemy prawidłowo zapisać rok bitwy pod Grunwaldem

```
1410 = 02602 = 0x582
```

Termin "heksadecymalny" jest długi, dlatego będziemy mówić o liczbach Hexa. Zapis **Hexa** jest wygodny, gdyż bajt to liczba dwucyfrowa, adres (wskaźnik) i słowo to 8 cyfr, a *double* to 16 cyfr Hexa. Zgodnie z tym, adres początku pamięci S-RAM `0x20000000` ma osiem cyfr Hexa.

Tabela 2.4: Konwersja liczb

Decimal	**Binary**	**Octal**	**Hexa**decimal
0	0000	0	0x0
1	0001	01	0x1
2	0010	02	0x2
3	0011	03	0x3
4	0100	04	0x4
5	0101	05	0x5
6	0110	06	0x6
7	0111	07	0x7
8	1000	010	0x8
9	1001	011	0x9
10	1010	012	0xA
11	1011	013	0xB
12	1100	014	0xC
13	1101	015	0xD
14	1110	016	0xE
15	1111	017	0xF

Wyobraźmy sobie rejestr mikrokontrolera złożony z trzydziestu dwóch przerzutników RS, takich jak na rysunku 2.6. Każdy przerzutnik ma możliwe dwa stany 0 i 1. Rejestr może więc zapamiętać 32-bitowy ciąg zero-jedynkowy. Tak samo są pamiętane dane w pamięci Flash i S-RAM. Liczby w komputerze zapisywane są binarnie. Zapis Hexa możemy stosować w źródłach C i asemlera, jest on tylko dla naszej wygody. **Hexa nie jest reprezentacją liczb w komputerze**. Mając zapis binarny możemy pogrupować cyfry po cztery i każdą czwórkę bitów zastąpic cyfrą Hexa, korzystając z tablicy 2.4.

2.5. Liczby całkowite bez znaku

Ile liczb możemy zapisać za pomocą n bitów? Ponieważ każdy bit to 0 lub 1, razem mamy 2^n możliwości. Są to liczby od 0 do $(2^n - 1)$. Na przykład 4 bity to liczby od `0000` do `1111` $\triangleq 2^4 - 1 = 15$.

Zamiana liczby dziesiętnej na binarną. Wykonujemy kolejne dzielenia całkowitoliczbowe przez dwa. Reszty z tych dzieleń, zapisane w kolejności, są binarną reprezentacją liczby. Przykłady

1410	0	LSB	1365	1	LSB
705	1		682	0	
352	0		341	1	
176	0		170	0	
88	0		85	1	
44	0		42	0	
22	0		21	1	
11	1		10	0	
5	1		5	1	
2	0		2	0	
1	1	MSB	1	1	MSB

2.6. Liczby całkowite ze znakiem

Jest kilka sposobów kodowania liczb całkowitych ze znakiem: znak-liczba, kod uzupełnień do jedności i uzupełnień do dwóch (w polskiej literaturze oznaczany **U2**). Wszystkie mają wspólną cechę: najstarszy bit (MSB) jest bitem znaku: zero = plus, jeden = minus. Tabela 2.5 pokazuje te różne sposoby kodowania.

W komputerach używany jest jedynie kod uzupełnień do dwóch. Powód jest prosty: gdy używamy U2, to ten sam układ logiczny (*hardware*) procesora wykonuje wszystkie operacje arytmetyczne na liczbach całkowitych (ze znakiem i bez): dodawanie, odejmowanie i mnożenie. Ponadto tylko U2 ma jedną reprezentację zera: w kodzie znak-liczba i U1 są dwa zera +0 i -0. Komplikuje to układ wykrywania zera i ustawiania flagi *Zero*.

2.6.1. Kod uzupełnień do jedności U1

Liczbę dodatnią, np. `a=0101'1010` zamienimy na ujemną `-a` o tej samej wartości bezwzględnej (czyli 90 na -90). Wystarczy zanegować każdy bit indywidualnie `-a=10100101`. Uzyskaliśmy dwie liczby: tam gdzie w pierwszej są jedynki, to w drujej zera i odwrotnie. Dodajmy te liczby

```
 01011010
 10100101
 --------
 11111111
```

uzyskaliśmy same jedynki, czyli minus zero. Możemy napisać ogólną formułę

$$a + (-a) = (2^n - 1)$$

gdzie n to ilość bitów reprezentacji liczby.

Logiczną negację każdego bitu indywidualnie wykonuje instrukcja asemblerowa NOT. W języku C także istnieje operator negacji logicznej

```
y = ~ y;    //negacja liczby całkowitej w kodzie U1
```

2.6.2. Kod uzupełnień do dwóch U2

Teraz nieco zmienimy algorytm: po zanegowaniu każdego bitu, do wyniku dodamy jedynkę. Suma liczby i jej negacji będzie teraz równa

$$a + (-a) = (2^n - 1) + 1 = 2^n$$

Zamienimy 90 na -90, a następnie -90 na 90, kodowanie U2

```
 01011010   +90                 10100110   -90
 10100101                       01011001
       +1                             +1
 --------------------------------------------
 10100110   -90                 01011010   +90
```

Jak widać działa w obie strony. Teraz dodamy 90+(-90)

```
     01011010   +90
     10100110   -90
 ----------------------
 1←00000000      0
```

Uzyskaliśmy zero i bit przeniesienia *Carry* równy jeden. Nazwa kodowania U2 wywodzi się z faktu, że suma liczb przeciwnych jest równa 2^n. W powyższym przykładzie liczby są zapisywane w jednym bajcie, wynik też jest jednobajtowy (zero). Suma liczb jest równa 2^8, co zapisane w jednym bajcie daje zero i *Carry*=1.

Fakt, że liczby U2 sumują się do potęgi dwójki 2^n prowadzi do następującej interpretacji wag poszczególnych pozycji. W liczbie zapisanej na n bitach, najbardziej znaczący bit MSB ma wagę ujemną -2^{n-1}, pozostałe bity mają wagi dodatnie tak jak dotychczas, czyli $2^i, i \in [n-2, n-3, \ldots, 0]$. Cyfrę na pozycji n liczby oznaczmy jako b_i. Każdą liczbę ze znakiem w kodzie U2 (dodatnią, ujemną i zero) zapisujemy zgodnie z formułą

$$a = -b_{n-1} * 2^{n-1} + \sum_{i=0}^{n-2} b_i * 2^i \tag{2.1}$$

Na przykład liczbę zapisaną na 4 bitach `1011` interpretujemy zgodnie ze wzorem (2.1) jako (n=4)

$$-1 * 2^3 + 0 * 2^2 + 1 * 2^1 + 1 * 2^0 = -8 + 2 + 1 = -5$$

Algorytm z negacją bit po bicie i dodaniem jedynki daje ten sam efekt

```
 1011
 0100
   +1
 ----
 0101
```

czyli `1011` to jest -5 (Tablica 2.5, kolumna U2). Jeżeli chcemy zwiększyć ilość bitów kodowania liczby, to dopisujemy zera z lewej strony, gdy MSB jest zerem i jedynki, gdy MSB jest jedynką (powielamy najstarszy bit w lewo do końca), na przykład liczbę pięć zapisaną na 4 bitach `0101` rozszerzymy do ośmiu bitów dopisując zera `00000101`. Liczbę ujemną -5 rozszerzymy dopisując jedynki `11111011`. Sprawdzimy to zamieniając -5 na 5

```
 11111011   -5
 00000100
       +1
 --------
 00000101   +5
```

Następujący fragment kodu C neguje liczbę kodowaną U2 (komentarze angielskie).

```
y = ~ y;    // toggle every bit
y += 1;     // obtain two's complement
```

Tabela 2.5: Kodowanie liczb ze znakiem

Kod binarny	Znak-Liczba	uzupełnień U1	uzupełnień U2
0000	+0	+0	0
0001	1	1	1
0010	2	2	2
0011	3	3	3
0100	4	4	4
0101	5	5	5
0110	6	6	6
0111	7	7	7
1000	-0	-7	-8
1001	-1	-6	-7
1010	-2	-5	-6
1011	-3	-4	-5
1100	-4	-3	-4
1101	-5	-2	-3
1110	-6	-1	-2
1111	-7	-0	-1

2.7. Dodawanie i odejmowanie liczb bez znaku, flaga *Carry*

Rejestr statusów *Program Status Register* xPSR (rysunek 3.9) zawiera flagi, czyli bity statusu. Teraz najważniejsze dla nas są flagi *Carry* C = przeniesienie, *Overflow* V = przepełnienie, *Zero* Z = zero, zapalana (Z=1) gdy wynik jest zerowy i gaszona (Z=0) gdy

niezerowy oraz *Negative* N = wynik ujemny, zapalana gdy MSB wyniku jest jedynką i gaszona gdy zerem.

Odejmowanie liczb bez znaku służy do ich porównywania. Procesor "nie wie" czy liczba jest ze znakiem, czy bez znaku. To jest nasza interpretacja, jak traktujemy ciąg bitów przechowywany np. w rejestrze procesora. **Procesor działania arytmetyczne wykonuje zawsze tak samo**. ADD dodaje dwie liczby (dwa rejestry) i ustawia flagi w zależności od wyniku. Odejmowanie SUB zamienia odjemnik na liczbę ujemną U2 i wykonuje dodawanie ustawiając stosownie flagi. Do programisty należy właściwa interpretacja flag.

Teraz prześledzimy jak należy interpretować flagę *Carry* gdy liczby traktujemy jako całkowite bez znaku. Wykonamy cztery przykładowe działania. Dla uproszczenia założymy, że mamy liczby czterobitowe $n = 4$. Zaczniemy od dodawania 3+6=9 i 6+12=18. Nad liczbami zaznaczono przeniesienia z poszczególnych pozycji (kolorem niebieskim).

```
Carry 0 ← 1 1 0                    Carry 1 ← 1 0 0
          0 0 1 1   +3                       0 1 1 0   +6
          0 1 1 0   +6                       1 1 0 0  +12
  C = 0 ← 1 0 0 1   +9             C = 1 ← 0 0 1 0   +2
```

W drugim przypadku wynik nie zmieścił się w czterech bitach, dlatego została zapalona flaga C *Carry*. Wykonując odejmowanie SUB, zmienimy znak odjemnika i dodamy do odjemnej (tak wykonuje to procesor). Przykładowe działania to 3 - 2 = 1 i 2 - 3 = -1.

```
Carry 1 ← 1 1 0                    Carry 0 ← 0 0 0
          0 0 1 1   +3                       0 0 1 0   +2
          1 1 1 0   -2                       1 1 0 1   -3
  C = 1 ← 0 0 0 1   +1             C = 0 ← 1 1 1 1   15
```

Teraz jest odwrotnie, flaga C jest zapalana, gdy jest OK i gaszona, gdy wynik nie mieści się w formacie liczb (-1 nie jest liczbą bez znaku). W języku angielskim przeniesienie wsteczne ("pożyczka") to *Borrow*. Procesor Cortex-M nie ma oddzielnej flagi *Borrow*, fizycznie zrealizowana jest tylko flaga *Carry*. Możemy powiedzieć, że w odejmowaniu liczb całkowitych bez znaku *Borrow* definiujemy jako negację *Carry*.

2.8. Dodawanie i odejmowanie liczb ze znakiem, flaga *Overflow*

Flagę *Overflow* V = przepełnienie, zdefiniujemy jako XOR (*exclusive OR*) dwóch bitów przeniesienia: do bitu MSB (bit znaku) i z bitu MSB, czyli z pozycji przedostatniej na ostatnią i z ostatniej do flagi *Carry*. Działanie XOR będziemy oznaczać $\oplus$, a jego definicja podana jest na rysunku 2.4. Wróćmy do dodawania 90 + (-90)

```
        Carry 1 ← 1 1 1 1 1 1 0
                  0 1 0 1 1 0 1 0   +90
                  1 0 1 0 0 1 1 0   -90
V = 1⊕1 = 0       0 0 0 0 0 0 0 0     0
```

Mamy dwa przeniesienia $b_7 \leftarrow b_6$ i $C \leftarrow b_7$. Ich XOR jest zerem, czyli po wykonaniu dodawania stan flag jest następujący V = 0 (brak przepełnienia) i C = 1 (przeniesienie). W

przypadku liczb ze znakiem interesuje nas tylko flaga przepełnienia V *Overflow*. Ponieważ nie jest zapalona, to znaczy że wynik jest poprawny. Sprawdzimy to na przykładach ($n = 4$) z rozdziału 2.7.

```
      Carry 0 ← 1 1 0                          Carry 1 ← 1 0 0
                0 0 1 1   +3                               0 1 1 0   +6
                0 1 1 0   +6                               1 1 0 0   -4
V = 0⊕1 = 1     1 0 0 1   -7          V = 1⊕1 = 0          0 0 1 0   +2
```

Pierwsze działanie (z lewej strony). Jeżeli liczby traktujemy jako całkowite ze znakiem, to `0011` oznacza +3, a `0110` oznacza +6. Ich suma 3+6=9 nie mieści się na czterech bitach. Zakres liczb 4-bitowych ze znakiem to [-8, 7] - patrz ostatnia kolumna tabeli 2.5. Dlatego zapalona jest flaga przepełnienia V = 1. Jeżeli te same liczby traktujemy jako całkowite bez znaku, to mamy takie samo działanie 3 + 6 = 9, ale teraz wynik jest poprawny, na co wskazuje zgaszone flaga C = 0, gdyż mieści się w zakresie liczb bez znaku [0, 15].

Drugie działanie (z prawej strony). Jeżeli liczby traktujemy jako całkowite ze znakiem, to `0110` oznacza +6, a `1100` oznacza -4. Ich suma 6-4=2 mieści się na czterech bitach, dlatego zgaszona jest flaga przepełnienia V = 0. Jeżeli te same liczby traktujemy jako całkowite bez znaku, to `0110` oznacza +6, a `1100` oznacza +12. Ich suma 6+12=18 nie mieści się w zakresie liczb bez znaku [0, 15] i flaga C = 1 jest zapalona.

Widzimy, że **flaga *Carry* dotyczy działań bez znaku, flaga *Overflow* dotyczy działań ze znakiem**. Procesor wykonując operacje arytmetyczne ustawia (odpowiednio do wyniku) wszystkie flagi. Do programisty należy prawidłowa interpretacja stanów flag.

Wyjaśnimy jeszcze dlaczego procesor stosuje XOR z dwóch przeniesień. Wynik nie mieści się w formacie (*Overflow ON*), gdy dodajemy dwie liczby, duże co do wartości bezwzględnej i obie są albo dodatnie, albo ujemne. Działanie XOR oznacza, że flaga *Overflow* będzie zapalona, gdy przeniesienie do bitu znaku różni się od przeniesienia do *Carry*. Liczby dodatnie:

```
      Carry 0 ← 0 1 0                          Carry 0 ← 1 1 1
                0 0 1 1   +3                               0 0 1 1   +3
                0 0 1 0   +2                               0 1 0 1   +5
V = 0⊕0 = 0     0 1 0 1   +5          V = 0⊕1 = 1          1 0 0 0   -8
```

Pierwsze działanie (z lewej strony). Wynik dodatni mieści się na trzech bitach, nie było przeniesień, wynik prawidłowy. Drugie działanie (z prawej strony). Jest tylko jedno czerwone przeniesienie, które "spada" na pozycję bitu znaku. Bit znaku powinien być zerem, a jest jedynką, gdyż został nadpisany, bo wynik dodatni nie mieści się w trzech bitach. Liczby ujemne: dwa przeniesienia: OK, jedno: źle

```
      Carry 1 ← 1 0 1                          Carry 1 ← 0 0 0
                1 1 0 1   -3                               1 1 0 1   -3
                1 1 0 1   -3                               1 0 1 0   -6
V = 1⊕1 = 0     1 0 1 0   -6          V = 1⊕0 = 1          0 1 1 1   +7
```

Pierwsze działanie (z lewej strony). Dodanie dwóch liczb ujemnych daje wynik ujemny. Żeby znak pozostał jedynką, **musi** być przeniesienie do bitu znaku. Wtedy `1 + 1 + 1 = 11 =` *Carry ON* `+ 1`, czyli muszą być dwa przeniesienia czerwone. Drugie działanie (z prawej strony), zabrakło przeniesienia do bitu znaku, więc w wyniku bit znaku został

nadpisany zerem. Jednocześnie dodanie dwóch znaków (jedynek) zapaliło flagę *Carry*. Razem zapala to *Overflow*.

2.9. Jak kompilator tłumaczy instrukcję `if (a < b)`

W kompilatorze C wszystkie zmienne muszą (przed użyciem) być zadeklarowane co do swego typu. Podobnie w definicji funkcji deklarujemy typ jej parametrów (danych wejściowych). Teraz przekonamy się jak to jest ważne. Zadeklarujemy zmienne bez znaku i napiszemy fragment programu

```
unsigned int a = 2, b = 3, c = 2;
if (a <= b);
    c = 1;
```

Kompilator przetłumaczy to na instrukcje asemblerowe

```
          CMP r6,r7    ; r6 = 2, r7 = 3
          BHI label    ; skocz gdy a > b
          MOV r8,#0x01 ; w r8 jest c
label     ...
```

Została zastosowana instrukcja skoku warunkowego `BHI`, *Branch if unsigned HIgher* która wykonuje skok gdy C = 1 & Z = 0. W naszym przykładzie instrukcja `CMP r6,r7` wykona odejmowanie `r6-r7 = 2 - 3 = -1`, ustawi flagi N=1, Z=0, C=0, V=0 czyli skok nie będzie wykonany. Teraz zadeklarujemy zmienne ze znakiem

```
int a = 2, b = 3, c = 2;
if (a <= b);
    c = 1;
```

Kompilator przetłumaczy to na instrukcje asemblerowe

```
          CMP r6,r7    ; r6 = 2, r7 = 3
          BGT label    ; skocz gdy a > b
          MOV r8,#0x01 ; w r8 jest c
label     ...
```

Została zastosowana instrukcja skoku warunkowego `BGT`, *Branch if signed Greater Than* która wykonuje skok gdy Z = 0 & N = V. W naszym przykładzie instrukcja `CMP r6,r7` wykona odejmowanie `r6-r7 = 2 - 3 = -1`, ustawi flagi N=1, Z=0, C=0, V=0, czyli skok nie będzie wykonany, gdyż N≠V. Instrukcje skoków warunkowych wymienione są w tablicy 8.5.

2.10. Stringi (ciągi znaków) w pamięci

Na temat przechowywania tekstu w postaci cyfrowej można by napisać cały oddzielny rozdział. Poruszane zagadnienia odnoszą się jedynie do prostych przypadków z którymi Czytelnik będzie miał do czynienia podczas wykonywania ćwiczeń zawartych w tej książce.

The American Standard Code for Information Interchange (ASCII) jest standardem kodowania znaków na siedmiu bitach, co daje 128 kodów. W tym są 33 znaki sterujące (`0x00-0x19` plus `0x7f`) oraz 95 znaków pisarskich. Znak spacji (`0x20`) jest znakiem pisarskim. Pierwszy standard ASCII został opublikowany w roku 1963. W tym czasie powszechnie używano mechanicznych maszyn do pisania. Na przykład `0x0D,CR` to powrót karetki (*Carriage return*). Karetka, to była ruchoma część maszyny do pisania w którą

wkręcało się papier. Znak `0x0A,LF` to wysuw papieru (*Line Feed*). W systemach operacyjnych Windows, te dwa znaki łącznie `0x0D0A` oznaczają przejście na początek (CR) nowej linii (LF). W systemach wzorowanych na UNIX (MacOS X, Linux) stosuje się jedynie znak LF.

Należy odróżniać znak sterujący `0x00 NUL` od znaku pisarskiego zero `0x30 0`. W późniejszym okresie pojawiły się różne rozszerzenia do ośmiu bitów kodu, ale pierwotny 7-bitowy standard, oznaczany często jako ASCII7 jest stosowany powszechnie do dziś w niezmienionej postaci. Tabela B.1 w dodatku B zawiera podstawowe, 7-bitowe kody znaków ASCII.

String, który jest ciągiem (wektorem) znaków ASCII kończy się bajtem `0x00 NUL`, który jest zarezerwowany jako znacznik końca stringu. Gdy w programie C deklarujemy string, kompilator automatycznie dodaje znak `NUL` na jego końcu. W asemblerze sami musimy to zrobić. Należy bezwzględnie pamiętać, że każdy **string musi kończyć się bajtem zerowym**, czyli znakiem NUL. W przeciwnym razie przykładowy program z tabeli 3.1 może przepisać całą pamięć, niszcząc wszystko po drodze.

Adresem całego stringu w pamięci jest adres jego początku, czyli bajtu o najmniejszym adresie, w którym jest pamiętany pierwszy znak stringu. W kolejnych bajtach są kolejne znaki i na końcu jest NUL. To oznacza, że długość stringu jest o jeden większa od ilości znaków. Przykład rozmieszczenia stringu o etykiecie `strSRC` w pamięci (patrz tabela 3.1)

adres	**kod**	**znak**
0x20000004	0x20	␣
0x20000005	0x53	S
0x20000006	0x6F	o
0x20000007	0x75	u
0x20000008	0x72	r
0x20000009	0x63	c
0x2000000A	0x65	e
0x2000000B	0x20	␣
0x2000000C	0x53	S
0x2000000D	0x74	t
0x2000000E	0x72	r
0x2000000F	0x69	i
0x20000010	0x6E	n
0x20000011	0x67	g
0x20000012	0x20	␣
0x20000013	0x20	␣
0x20000014	0x00	NUL

Adres całego stringu to `0x20000004`. Znak ␣ symbolizuje spację.

String w pamięci jest wektorem (tablicą jednowymiarową liczb). Dzięki temu możemy je ze sobą porównywać, co umożliwia porządkowanie stringów (na przykład nazwisk) w porządku alfabetycznym. Algorytm takiego porządkowania:

- Jeżeli dwa stringi mają różną długość, to zakłada się, że krótszy jest uzupełniony zerami `0x00`, czyli znakami NUL.
- Teraz stringi są porównywane znak po znaku począwszy od pierwszego.
- Jeżeli porównywany bieżący znak jest mniejszy od odpowiadającego w drugim stringu, to cały string uważa się za mniejszy i nie porównuje sią następnych.

Rozdział 3

Jak działa procesor

Asembler nie jest kolejnym językiem programowania. Jest tylko wygodnym narzędziem do programowania bezpośrednio w kodach maszynowych procesora. Pozwala na lepsze zrozumienie jak procesor wykonuje program. Programowanie asemblerowe jest nieodzownym elementem wiedzy o kompilatorach, systemach operacyjnych i architekturze komputera. Ponadto programy asemblerowe mogą potencjalnie być bardziej efektywne od napisanych w językach wysokiego poziomu, jak C. Kompilatory czasami nie są w stanie wykorzystać szczególnych cech danej architektury. Dlatego często stosujemy wstawki asemblerowe do tych fragmentów kodu, których czas wykonania jest szczególnie krytyczny. Asembler jest często wykorzystywany w rdzeniu systemu operacyjnego i funkcjach które są "blisko sprzętu", np. sterownikach urządzeń peryferyjnych. Dla nas najważniejsze jest to, że znajomość sposobu w jaki kompilator tłumaczy program C na asembler pomaga programiście pisać dobre programy w C. Zestaw instrukcji procesorów Cortex-M jest stworzony z myślą o C (na przykład asemblerowa instrukcja `TBB [pc,Rn]` realizuje `switch` języka C – rozdział 8.7). W książce będziemy często porównywać fragmenty C z odpowiadającym im kodem asemblerowym.

Z drugiej strony, współczesne kompilatory C bardzo dobrze optymalizują kod wynikowy i są wyposażone w rozbudowane opcje optymalizacji. Dlatego w prostych aplikacjach nie ma potrzeby stosowania asemblera.

Ten rozdział stanowi tylko wprowadzenie do generacji i wykonania kodu maszynowego procesora. Jeżeli Czytelnik nie zrozumie niektórych szczegółów, proszę się nie zrażać, będą one wyjaśnione w kolejnych rozdziałach.

3.1. Tłumaczenie kodu źródłowego C na kod maszynowy

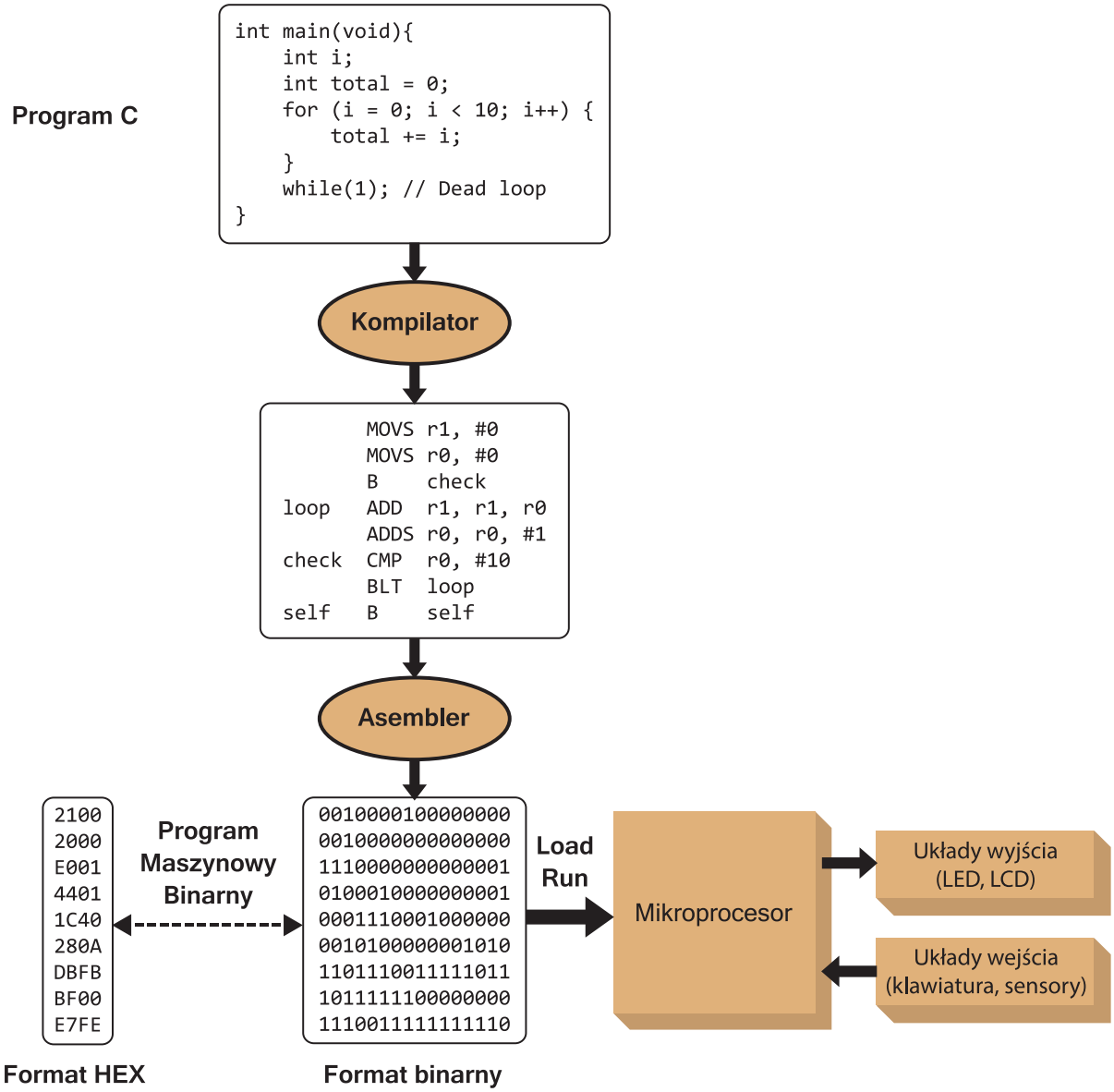
Kompilator tłumaczy program C, napisany w postaci źródłowej zrozumiałej dla nas (*source code*), na kod pośredni *intermediate representation (IR)*. Kody tworzone przez kompilator całkowicie zależą od architektury i listy rozkazów konkretnego procesora. Jeżeli chcemy przenieść nasz program napisany w kodzie źródłowym C na inną architekturę, musimy powtórzyć kompilację na zgodną z listą instrukcji tej nowej architektury.

Program asemblerowy składa się z pięciu zasadniczych elementów

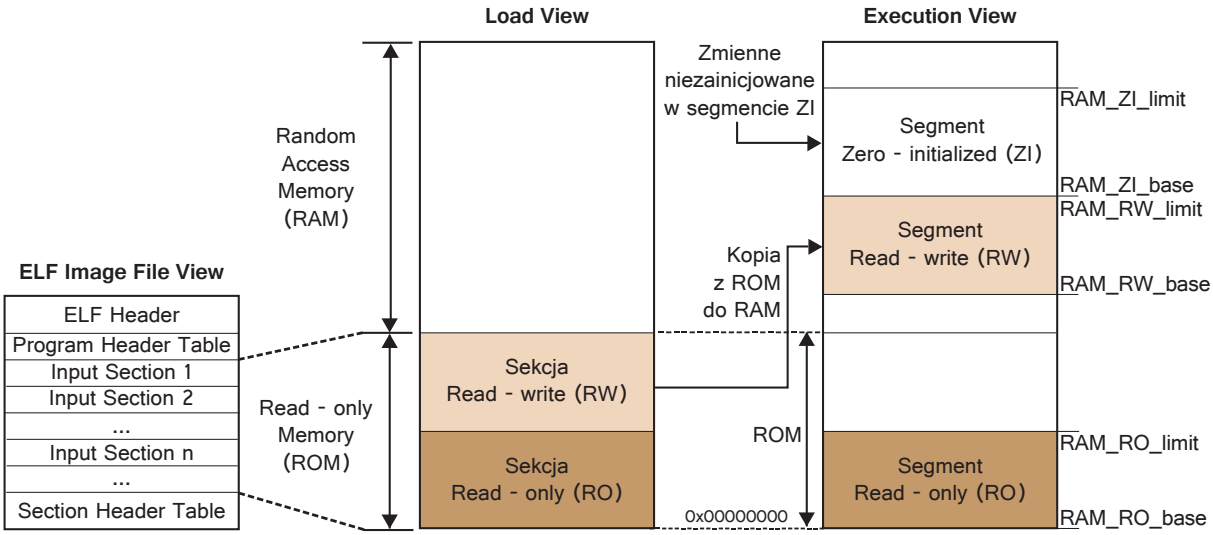
1. **Etykieta** (*label*) reprezentuje adres instrukcji lub danej (zmiennej lub stałej) w pamięci. Musi zaczynać się równo z początkiem linii (bez odstępów).
2. **Mnemonik** (*instruction mnemonic*) jest symbolem instrukcji, którą ma wykonać procesor, np. `ADD` to dodaj liczby całkowite.
3. **Operandy** (*operands*) wskazują na jakich danych ma być wykonana operacja. Najczęściej to nazwy rejestrów lub liczby stałe.
4. **Komentarz** (*program comment*) komentarz zaczyna się od średnika i kończy z końcem linii.

5. **Dyrektywy** asemblacji (*assembly directives*) nie są instrukcjami procesora. Sterują przebiegiem asemblacji i pozwalają łączyć uzyskany kod z pozostałą częścią programu.

Na proces kompilacji składają się następujące etapy: wykonanie poleceń preprocesora, analiza leksykalna, składniowa i semantyczna, optymalizacja kodu wynikowego oraz generacja kodu. Postać IR jest bardzo podobna do asemblera, jak pokazano na rysunku 3.1. Wynikiem działania kompilatora jest plik w formacie *executable and linkable format* (ELF). Firma Unix System Laboratories opracowała i opublikowała ELF jako standard formatu kodów wykonywalnych, kodów obiektowych (*object codes*), bibliotek współdzielonych oraz zrzutów pamięci. Plik ELF składa się z jednego nagłówka, listy segmentów programu, listy nagłówków sekcji oraz danych podzielonych na segmenty i sekcje. Kody obiektowe są programami pośrednimi (*intermediate code*) generowanymi przez kompilator.



Rys. 3.1: Kompilacja programu C do postaci wykonywalnej



Rys. 3.2: Rozmieszczenie segmentów pamięci w fazie ładownia i fazie wykonania programu

Kompilator łączy ze sobą kody obiektowe w fazie konsolidacji, czyli linkowania (*linking stage*), to znaczy tworzenia kodu maszynowego (*machine program*), czyli binarnego wykonywalnego (*binary executable*).

ELF zawiera informacje o łączeniu z innymi fragmentami kodu potrzebne w fazie konsolidacji (*linkable interface*) oraz informacje gdzie należy załadować program i dane przed wykonaniem programu (*executable interface*). W pliku ELF obszary pamięci o tym samym charakterze są łączone w większe sekcje. Są dwa sposoby lokowania danych zawartych w pliku ELF: faza ładowania do pamięci stałej (*Load View*) i faza rozmieszczenia segmentów przed wykonaniem programu (*Execution View*).

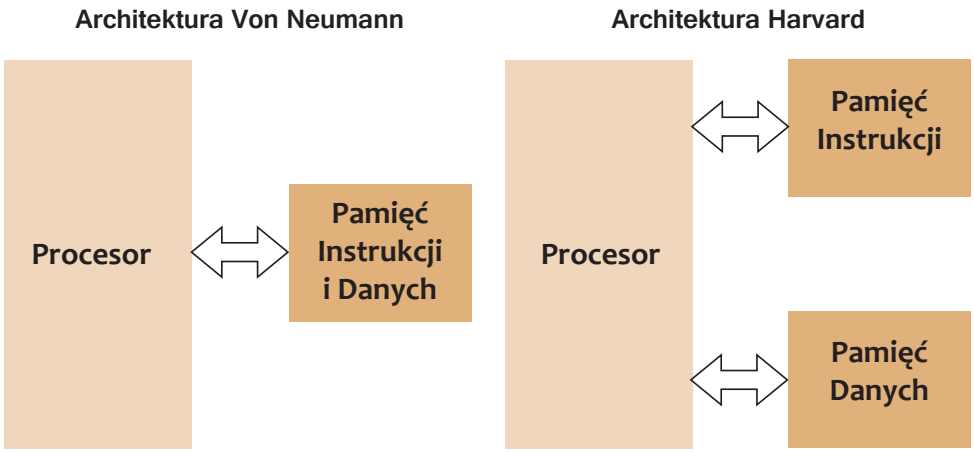
• W fazie ładowania do pamięci stałej dane są podzielone na dwa regiony: pamięci stałej ROM (*Read-only Memory*) i pamięci o dostępie swobodnym (*Random Access Memory*). Plik ELF zawiera także adresy bazowe obu regionów zlokalizowane w pamięci stałej typu Flash mikrokontrolera (rysunek 3.2).

• W fazie wykonania (*execution*) dostępne są adresy czterech krytycznych obszarów pamięci:

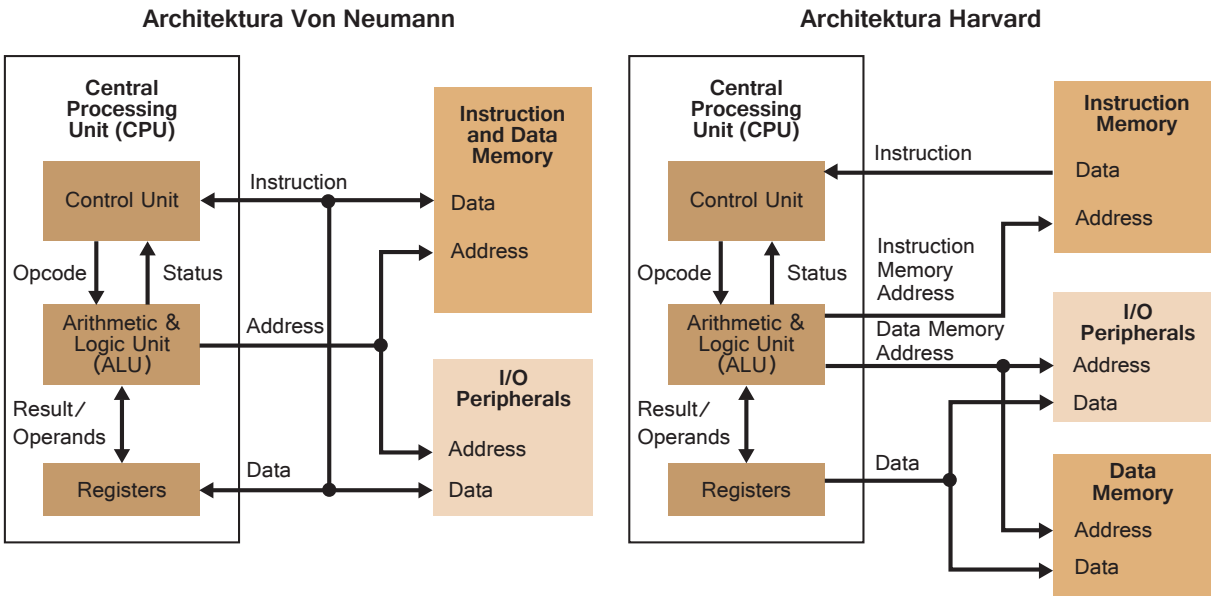
1. Segment **Code** zawiera binarną postać kodu programu.
2. Segment **RO-data**: dane stałe, przechowywane w pamięci ROM.
3. Segment **RW-data**: zmienne zainicjowane w RAM.
4. Segment **ZI-data**, wyzerowany RAM: wszystkie zmienne zadeklarowane w programie niezainicjowane i zainicjowane jako zerowe.

3.2. Ładowanie programu maszynowego do pamięci

Programy wykonywalne są przechowywane w pamięci nieulotnej (*non-volatile memory*), która zachowuje zapisane w niej informacje po wyłączeniu zasilania. Dzięki temu program nie jest tracony i system może wystartować po załączeniu zasilania. Procesor musi załadować zmienne zainicjowane do pamięci o dostępie swobodnym RAM przed rozpoczęciem wykonywania właściwego programu. Pamięci ulotne wykonywane są jako dynamiczny DRAM (*Dynamic Random Access Memory*), które wymagają ciągłego odświeżania poprzez okresowe wymuszanie sygnałów wyboru RAS (*Row Access Strobe*) lub CAS (*Column Access Strobe*), albo statyczny S-RAM (*Static Random Access Memory*), w



Rys. 3.3: Dwie architektury komputerów



Rys. 3.4: Architektura Von Neumann i architektura Harvard

którym każdy bit jest przerzutnikiem. W stosunku do DRAM, gęstość danych jest cztery razy mniejsza w S-RAM, ale za to czas dostępu do danych w S-RAM jest siedem razy krótszy.

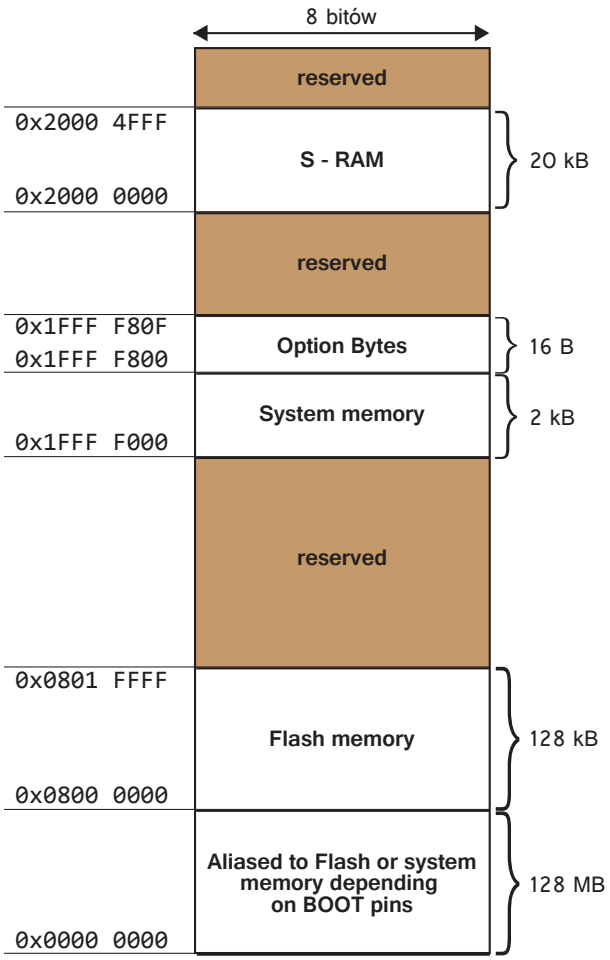
3.2.1. Architektura Harvard versus Von Neumann

Są dwie podstawowe architektury komputerów: *Von Neumann* i *Harvard architecture*, rysunki 3.3 i 3.4. W architekturze *Von Neumann* dane i instrukcje współdzielą ten sam obszar pamięci połączony z procesorem jedną, wspólną magistralą adresową. Magistrala (*memory address bus*) jest wewnętrznym elementem komunikacyjnym systemu komputerowego, który umożliwia wymianę danych pomiędzy poszczególnymi składnikami systemu, takimi jak jednostka centralna CPU (*Central Processor Unit*), pamięć i urządzenia peryferyjne. W architekturze *Harvard* pamięci danych i programu są rozdzielone i połączone z procesorem za pomocą oddzielnych magistral. Umożliwia to pobranie kodu instrukcji i operandu (danej) w tym samym cyklu maszynowym. W architekturze *Von Neumann*

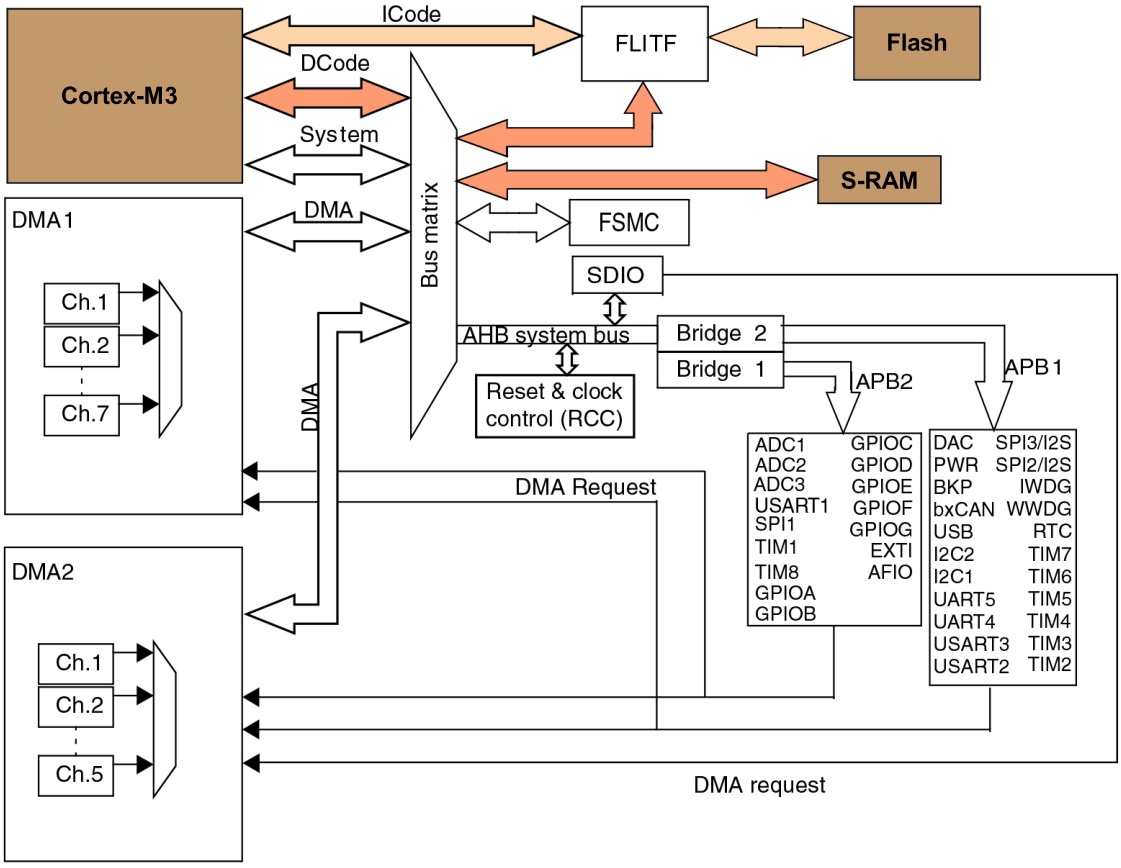
potrzebne są co najmniej dwa cykle: jeden na pobranie kodu instrukcji, drugi na pobranie danej, po tej samej magistrali.

Procesory 32-bitowe mają 32 linie adresowe. Współczesne komputery są adresowane bajtowo, to znaczy, że każdy adres wskazuje na jeden bajt (8 bitów) pamięci. Gdy pamięć ma 32 linie (sygnały) adresu, to znaczy, że cała przestrzeń pamięci zawiera $2^{32} = 4'294'967'296$ bajtów, czyli cztery gigabajty $4GB$ (stosowany w książce sposób oznaczania przedrostków był omówiony we wstępie, strona XXI).

Ilość pamięci, jaką może zaadresować procesor nazywamy jego pamięcią wirtualną lub całkowitą przestrzenią adresową. W tej przestrzeni możemy umieszczać fizyczne układy pamięciowe. Na przykład mikrokontroler **STM32F103VB** ma wbudowaną pamięć nieulotną typu Flash 128kB i statyczny RAM 20kB. Pamięć Flash 128kB to $128 * 1024 = 131'072 = 2^7 * 2^{10} = 2^{17}$ bajtów, czyli Flash ma 17 linii adresowych. Zgodnie z architekturą Cortex-M3 jest umieszczony od adresu 0x0800'0000 do adresu 0x0801'FFFF. Statyczny RAM 20kB to $20 * 1'024 = 20'480 = 16'384 + 4'096 = 2^{14} + 2^{12}$ =0x5000. Jest umieszczony od adresu 0x2000'0000 do adresu 0x2000'4FFF. Rysunek 3.5 pokazuje rozmieszczenie fizycznych bloków pamięci w obszarze wirtualnym Cortex-M3. Mikrokontroler ma dwie "nóżki" (piny, wyprowadzenia) oznaczone jako BOOT0 i BOOT1, które określają tryb startu procesora, np. gdy BOOT0=0, pamięć Flash jest powielana (*aliased*) od adresu zerowego. Procesor startując pobiera zawartość wskaźnika stosu z adresu `0x0000 0000`, a zawartość licznika programu *Program Counter* PC z adresu `0x0000 0004`. Powielenie



Rys. 3.5: Mapa pamięci procesora STM32F103VB



Rys. 3.6: Architektura mikrokontrolera STM32F103Vx

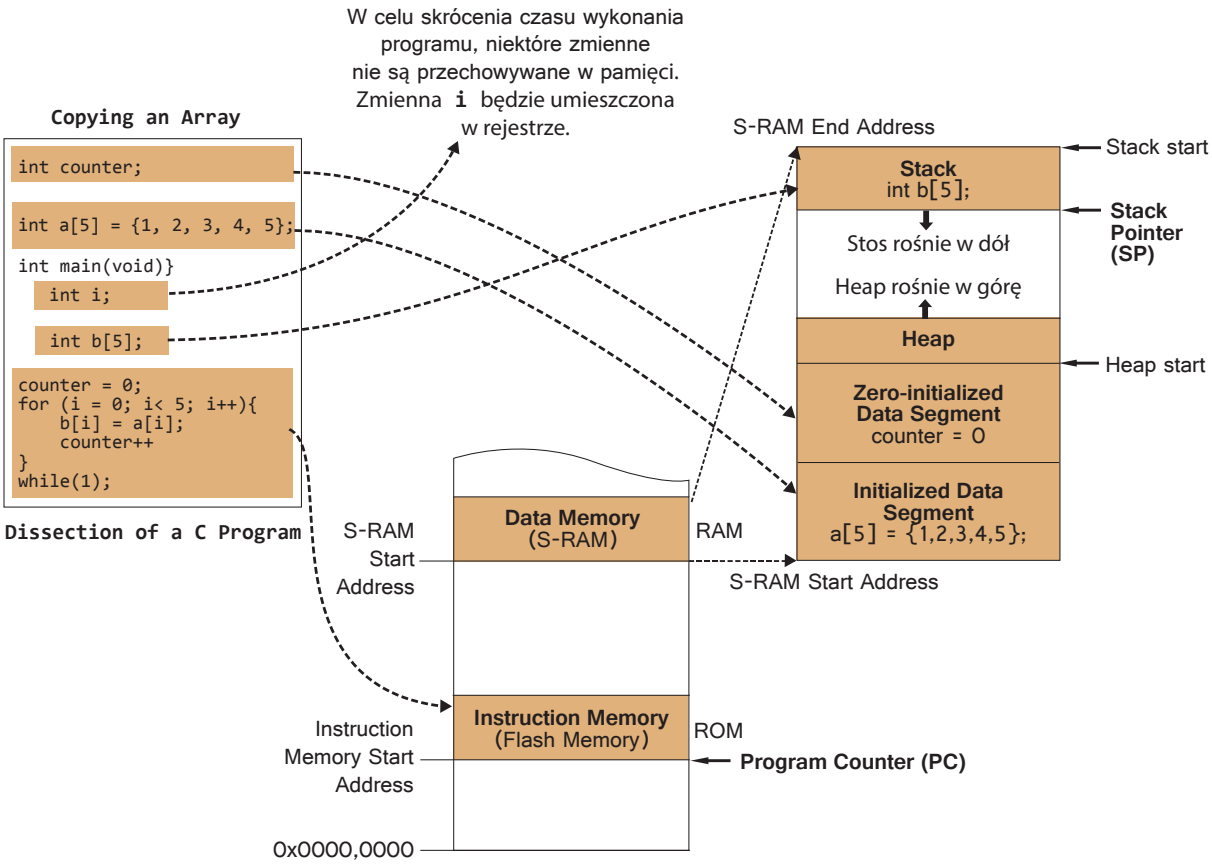
obszarów powoduje, że wartości te są pobierane fizycznie z Flash, ale procesor "myśli", że pobiera je z adresów zerowych.

- Architektura *Von Neumann* jest prosta i tania
- Architektura *Harvard* pozwala procesorowi na jednoczesne pobranie kodu rozkazu z pamięci instrukcji i operandu (danej) z pamięci danych. Architektura *Von Neumann* wymaga, aby oba odczyty odbyły się w kolejnych taktach zegara. Dlatego wykonywanie programu jest wolniejsze, przy tej samej częstotliwości taktowania.
- Architektura *Harvard* ma lepszą efektywność enegetyczną. Przy tych samych wymaganiach na moc obliczeniową, procesor *Harvard* może być wolniej taktowany, co powoduje mniejsze zużycie energii.
- W architekturze *Harvard* magistrale danych i instrukcji mogą mieć różne szerokości (ilości linii do przesyłania danych). Na przykład, często w procesorach sygnałowych DSP magistrala instrukcji jest szersza, żeby zmniejszyć ilość taktów zegara potrzebną do wykonania jednej instrukcji.

3.2.2. Rozmieszczenie obszarów pamięci

Procesory ARM Cortex-M3/M4/M7 mają architekturę typu *Harvard*. Pamięć instrukcji Flash i pamięć danych S-RAM są zintegrowane z procesorem w tym samym *chip*ie (kawałku krzemu). Rysunek 3.6 przedstawia architekturę mikrokontrolera STM32F103Vx. Kolorami zaznaczono mikroprocesor i obie pamięci oraz magistralę instrukcji programu **ICode** i magistralę danych **DCode**. Architektura przewiduje pobieranie danych z S-RAM

i instrukcji z Flash. Blok FLITF wspomaga potokowe przetwarzanie instrukcji, patrz rysunek 3.12. Możliwe jest też pobieranie danych z Flasha - są to operandy stałe, lub inne stałe programu (np. liczba Π). Pamięć instrukcji stosuje pamięć wolną, ale nieulotną Flash. Pamięć danych: szybką, ale ulotną S-RAM.



Rys. 3.7: Rozmieszczenie obszarów pamięci w architekturze *Harvard*

Rysunek 3.7 pokazuje rozmieszczenie obszarów pamięci w architekturze *Harvard*. Gdy procesor ładuje program, wszystkie zainicjowane **zmienne globalne**, jak wektor `a[5]`, są kopiowane z pamięci programu Flash do zainicjowanego segmentu danych *Initialized Data Segment*. Niezainicjowane zmienne globalne, jak zmienna `counter` mają przydzielone lokacje w wyzerowanym obszarze pamięci *Zero-initialized Data Segment*. **Zmienne lokalne** funkcji, jak tablica `b[5]`, są lokowane na stosie w chwili wywołania funkcji. Ta lokacja jest zwalniana (likwidowana) w chwili powrotu z funkcji. Stos jest umieszczony od najwyższego adresu pamięci S-RAM i "rośnie w dół", wskaźnik stosu SP (*Stack Pointer*) jest dekrementowany **przed** umieszczeniem danych na stosie. Ważnym obszarem pamięci jest sterta *Heap* zarezerwowana dla dynamicznego przydziału pamięci. Jest umieszczana nad obszarem zerowanym i rośnie w górę, czyli w kierunku większych adresów.

Przed rozpoczęciem wykonywania programu pamięć danych jest więc podzielona na cztery segmenty. Dane zainicjowane, niezainicjowane, stos i sterta. Dwa pierwsze są przydzielone na stałe i ich rozmiar nie zmienia się w czasie całego wykonywania programu, dwa ostatnie (stos i sterta) zmieniają swoje rozmiary.

- **Initialized Data Segment** zawiera zmienne globalne i statyczne `static`, którym program nadaje wartości początkowe. Na przykład jeżeli deklaracja w kodzie źródłowym C `int capacity = 100;` występuje poza ciałem funkcji (to znaczy,

że jest zmienną globalną), to procesor umieszcza tą zmienną w segmencie danych zainicjowanych z jej wartością początkową. Robi to tak zwana "rozbiegówka", czyli program systemowy wykonywany po RESET, a przed wywołaniem funkcji `main`.

- **Zero-Initialized Data Segment** zwiera wszystkie zmienne globalne i `static`, niezainicjowane, albo które zostały zainicjowane na wartość zerową. Na przykład globalnie zadeklarowany wektor `char name[20];` otrzyma lokację w tym segmencie.
- Sterta **Heap** jest przeznaczona na wszystkie przydziały dynamiczne, wykonywane za pomocą funkcji bibliotecznych `malloc()` lub `calloc()`. Funkcja `free()` likwiduje przydział i zwalnia odpowiedni obszar pamięci sterty. Sterta jest umieszczana bezpośrednio powyżej obszaru *Zero-Initialized Data Segment*.
- Stos **Stack** przechowuje wszystkie zmienne lokalne podprogramów, łącznie ze zmiennymi funkcji `main`. Jest kolejką typu **LIFO** (*Last-In, First-Out*), czyli pierwszy wychodzi ten, który wszedł ostatni. Mówimy, że odkładamy na wierzch stosu PUSH i zdejmujemy z wierzchu stosu POP. Jeżeli mamy stos talerzy, to z wierzchu stosu zdejmiemy POP talerz, który położyliśmy PUSH jako ostatni. Jeżeli nie było przepychanek, to kolejka do kasy jest typu **FIFO** (*First-In, First-Out*), czyli pierwszy płaci ten, który pierwszy zajął miejsce w kolejce. Na stosie przechowywane są także rejestry, które chcemy chronić w momencie wywołania funkcji (podprogramu) oraz adresy powrotu wywołań zagnieżdżonych (*nested*). Adres powrotu funkcji aktualnie wykonywanej jest przechowywany w specjalnym rejestrze *Link register (LR)*. Wskaźnik stosu `SP` wskazuje na element ostatnio odłożony na stos. Gdy procesor ma odłożyć (PUSH) nowy element na stos, najpierw dekrementuje wskaźnik stosu (`SP = SP - 4`), a następnie zapisuje element do nowej lokacji.

Procesor umieszcza stertę i stos w pamięci RAM, na przeciwnych końcach obszaru pozostałego po wcześniejszych przydziałach. W czasie wykonywania programu zmienne są dynamicznie odkładane na stos i z niego zdejmowane (wtedy zwalniają przydzielony wcześniej fragment pamięci), dynamicznie są także tworzone i zwalniane lokacje w stercie *heap*. Stos rośnie/maleje w kierunku sterty, a sterta w kierunku stosu, jak pokazano na rysunku 3.7. Gdy oba segmenty najdą na siebie, dochodzi do nadpisania (zniszczenia) adresu powrotu, co zawsze powoduje totalną katastrofę.

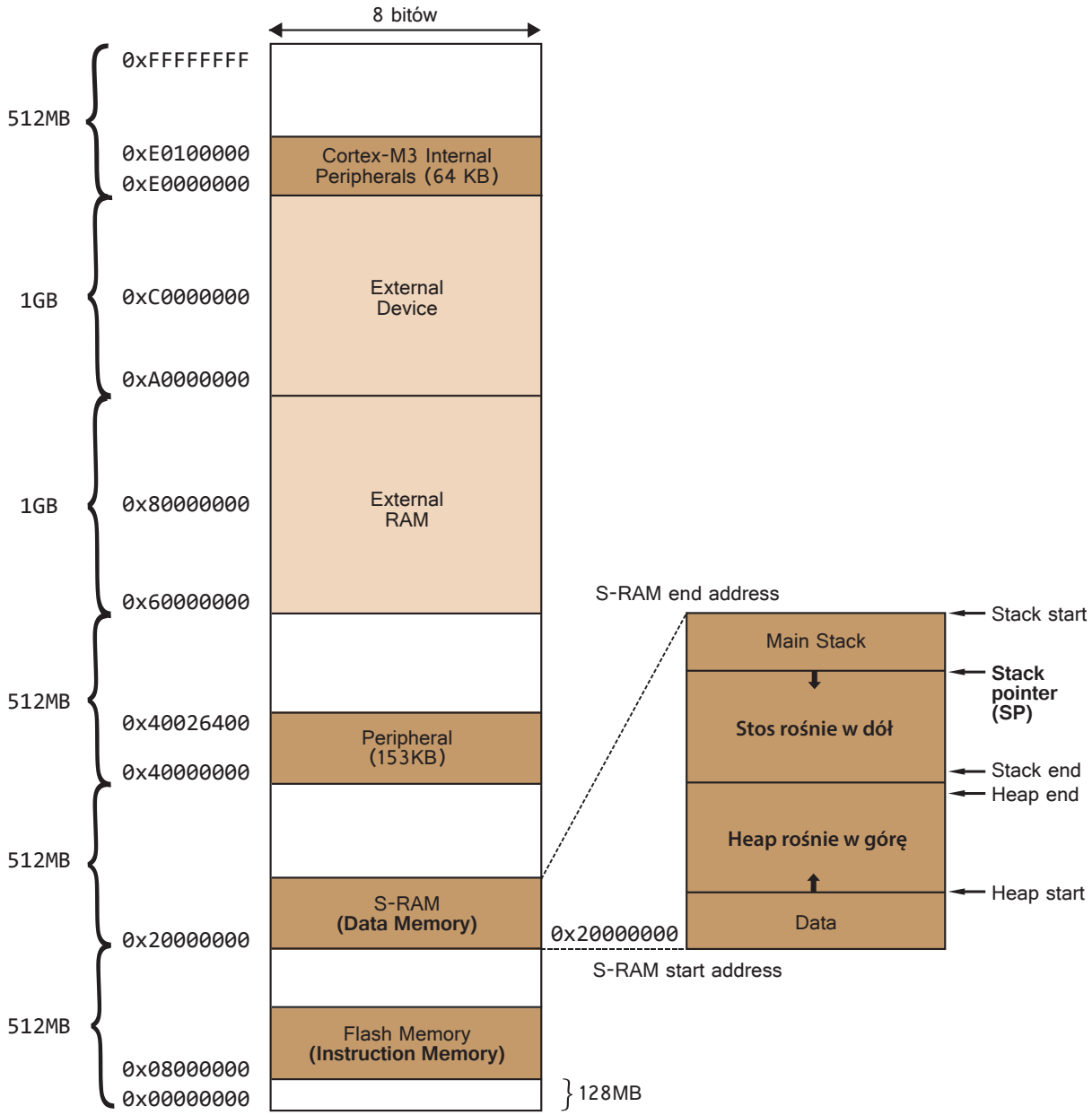
Układy peryferyjne mają z reguły rejestry sterujące ich pracą oraz rejestry służące do wymiany danych z rdzeniem. W architekturze ARM, wszystkie rejestry układów zintegrowanych z rdzeniem są umieszczone w tej samej przestrzeni adresowej co Flash i S-RAM i są dostępne tak, jak komórki pamięci. Ta metoda jest nazywana *memory mapped I/O*. Rysunek 3.8 przedstawia mapę pamięci mikrokontrolera z rdzeniem Cortex-M. Mapa pamięci jest ustalana przez producenta zgodnie z architekturą ARMv7-M i programista nie może jej zmienić. W przestrzeni liniowej 4GB zakresy adresów pamięci programu, danych, peryferiali i pamięci zewnętrznej są rozłączne:

- Zintegrowany (*on chip*) Flash ma adres startowy `0x0800'0000`
- Zintegrowany (*on chip*) S-RAM ma adres startowy `0x2000'0000`
- Rejestry peryferiali zintegrowanych z rdzeniem (*on chip*) zaczynają się od adresu `0x4000'0000`
- Urządzenia zewnętrzne mają rejestry od adresu `0xA000'0000`, na przykład blok FSMC opisany w dokumencie RM0008 (Tabela 4.2)
- Od adresu `0xE000'0000` są umieszczone rejestry rdzenia *core peripherals*, szczegóły w dokumencie PM0056 (Tabela 4.2)

3.3. Rejestry wewnętrzne procesora

Zanim zaczniemy opisywać jak jest wykonywany program, musimy poznać bardzo ważny element każdego procesora, czyli jego rejestry wewnętrzne. Rejestry są najszybszym miejscem przechowywania danych w całym systemie komputerowym. Wszystkie rejestry mają taką samą długość: w procesorze 8-bitowym to 8 bitów, 16-bitowym 16 i tak dalej. Cortex-M jest 32-bitowy więc każdy jego rejestr ma 32 bity. Procesor czyta /zapisuje wszystkie bity rejestru jednocześnie, choć są możliwe operacje logiczne na pojedynczych bitach. Rejestr może przechowywać wartość operandu instrukcji logicznej lub arytmetycznej. Może to być także adres pamięci (wskaźnik) lub przesunięcie adresu (*offset*) w instrukcjach odczytu /zapisu pamięci.

Rdzeń procesora *core* ma dwa rodzaje rejestrów: ogólnego przeznaczenia (akumulator) *general-purpose registers* i specjalne *special-purpose registers*. Rejestry ogólne przechowują



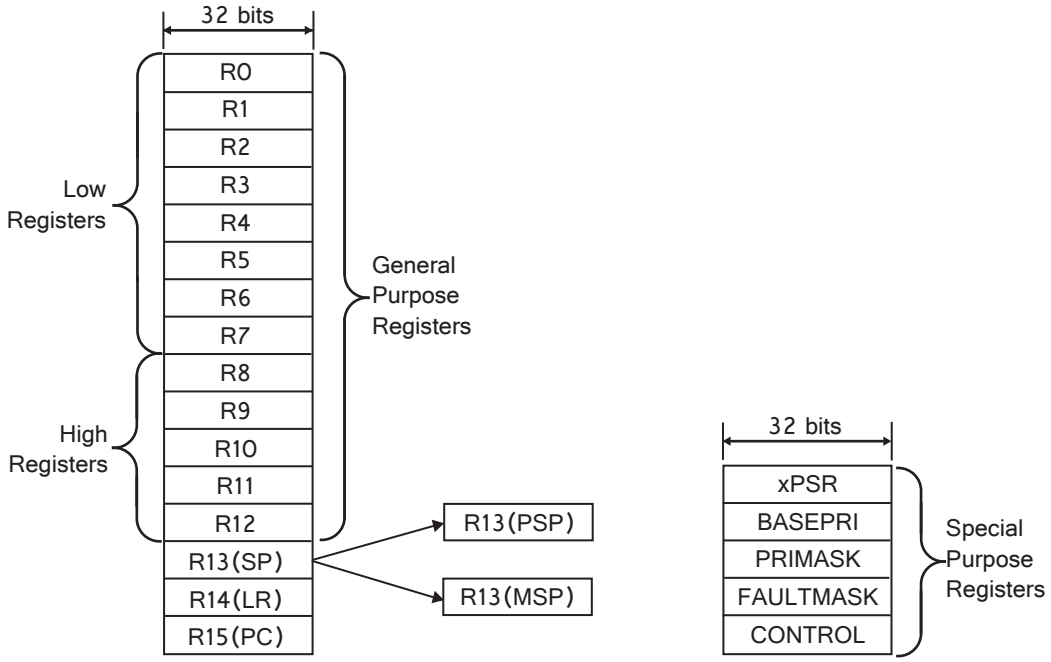
Rys. 3.8: Mapa pamięci mikrokontrolera z rdzeniem Cortex-M

operandy i wyniki działań w czasie wykonywania programu, rejestry specjalnego przeznaczenia mają ściśle określone funkcje jak status procesora, licznik programu PC, czy wskaźnik stosu SP.

Dostęp do danych w rejestrze jest znacznie szybszy niż do danych w pamięci. Jeżeli najczęściej wykorzystywane dane (np. licznik pętli) przechowujemy w rejestrach, może to znacznie przyśpieszyć wykonywanie programu. W większości programów częstotliwość odwoływania się do różnych zmiennych jest bardzo zróżnicowana, do pewnych zmiennych program odwołuje się szczególnie często. To zjawisko jest wszechobecne w aplikacjach i jest nazywane *temporal locality*, czyli lokalność w czasie. Określa się też *spatial locality*, czyli lokalność w przestrzeni. Pojęcia te opisują dwie różne charakterystyki dostępu programu do danych. Sekwencja odwołań ma cechę *spatial locality*, jeżeli dane do których program się odwołuje w krótkim odstępie czasu są blisko siebie w pamięci (np. kolejne adresy pamięci, ten sam sektor na dysku). Sekwencja ma cechę *temporal locality*, gdy odwołania do określonych danych następują często w krótkim odcinku czasu. Skrócenie czasu wykonania programu można uzyskać jeżeli dane o częstym dostępie umieścimy w rejestrach, a dane *spatial locality* w szybkiej pamięci podręcznej *cache*.

Ilość rejestrów wewnętrznych procesora jest mała, najczęściej od 4 do 16, z dwóch ważnych powodów:

- Po pierwsze, pomiary pokazały, że obszar rejestrów w strukturze procesora nagrzewa się znacznie mocniej niż pozostałe obszary krzemu. Zmniejszenie ilości rejestrów pomaga rozwiązać problem odprowadzania ciepła
- Po drugie, jeżeli jest mało rejestrów, to potrzebujemy mniej bitów w kodzie instrukcji do ich adresowania (wskazywania), dzięki temu kodowanie jest krótsze. Załóżmy, że procesor ma 16 rejestrów. Zakodowanie instrukcji która ma trzy operandy, np. `ADD r3,r1,r2` (r3 = r1 + r2) wymaga 4 bitów na wskazanie każdego rejestru, czyli razem 12 bitów w kodzie instrukcji jest poświęcone tylko na adresowanie rejestrów. Gdy procesor ma 8 rejestrów, liczba bitów maleje do 9.



Rys. 3.9: Rejestry wewnętrzne procesora Cortex-M

Decyzja, które zmienne umieścić w rejestrach nazywana jest *register allocation* i jest ważnym składnikiem optymalizacji kodu. Później przekonamy się że kompilator C środowiska Keil μVision maksymalizuje wykorzystanie rejestrów. Optymalizacja ta nie jest łatwa, gdyż niektóre instrukcje procesora Cortex-M wykorzystują tylko osiem pierwszych rejestrów (*low registers*). Decyzja o alokacji zmiennych do rejestrów jest podejmowana na podstawie czasu życia danej zmiennej. Jeżeli zakończył się czas życia określonej zmiennej w naszym programie, możemy zwolnić zajmowany przez nią rejestr dla innej zmiennej.

Rejestry wewnętrzne procesora Cortex-M pokazane są na rysunku 3.9.

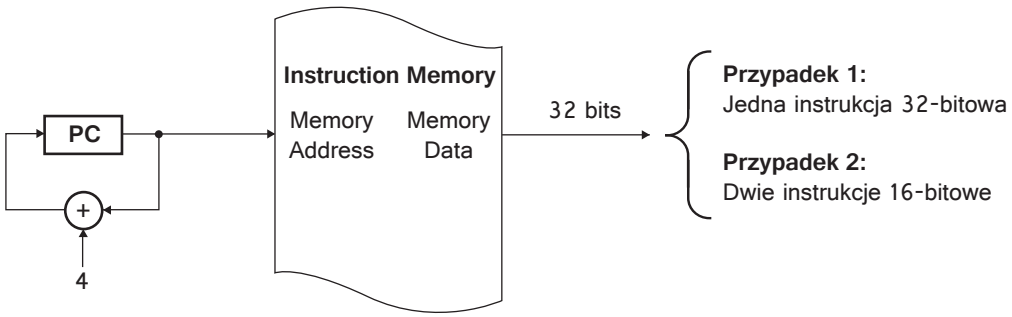
- Jest 13 rejestrów ogólnego przeznaczenia *General Purpose Registers* (`r0 - r12`). Pierwszych 8 jest nazywanych *Low Registers*, kolejnych pięć (`r8 - r12`) to *High Registers*. Niektóre instrukcje 16-bitowe mają dostęp tylko do *Low Registers*.
- ***Stack Pointer*** SP `r13`, czyli wskaźnik stosu, przechowuje adres wierzchołka stosu (adres danej ostatnio zapisanej na stos). Cortex-M ma dwa wskaźniki stosu: podstawowy *Main Stack Pointer* i procesowy *Process Stack Pointer*. To oznacza, że możliwe jest używanie dwóch różnych stosów. Procesor używa PSP gdy wykonuje program użytkownika, a MSP w trybie pracy uprzywilejowanej i w czasie obsługi przerwań. Po starcie, aktywnym wskaźnikiem w SP jest MSP.
- ***Link Register*** LR `r14` przechowuje adres powrotu z aktualnie wykonywanego podprogramu, czyli instrukcji następnej po CALL w programie głównym. Podczas obsługi przerwania LR przechowuje specjalną wartość wskazującą, czy używane jest MSP, czy PSP.
- ***Program Counter*** PC `r15`, licznik programu. Wskazuje (adresuje) instrukcję następną po ostatnio pobranej (*Fetch*) do wykonania.
- ***Special-Purpose program status register*** xPSR zawiera flagi statusów, takie jak: *negative, zero, carry, overflow*.
- ***Base priority mask register*** BASEPRI zawiera próg priorytetu obsługi przerwań. Priorytet jest tym wyższy, im mniejsza jest liczba go określająca. Procesor blokuje obsługę przerwań, których priorytety są liczbowo większe od progu zapisanego w BASEPRI, czyli wszystkich o niższych priorytetach.
- ***Priority mask register*** PRIMASK blokuje wszystkie przerwania z wyjątkiem NMI oraz *hard faults*, czyli twardych protestów (wyjątków).
- ***Fault mask register*** FAULTMASK blokuje wszystkie przerwania z wyjątkiem NMI.
- ***Control register*** CONTROL określa który wskaźnik stosu jest aktualnie w SP, PSP czy MSP oraz tryb pracy procesora *privileged/unprivileged*.

Licznik programu PC przechowuje adres pamięci programu, z którego procesor pobierze następną instrukcję do wykonania. Instrukcje są wykonywane sekwencyjnie, jeżeli nie ma rozejść programowych, czyli skoków (warunkowych /bezwarunkowych) i wywołań podprogramów (obsługi przerwań i funkcji). Instrukcje w Cortex-M są 16 lub 32 bitowe. Typowo procesor zwiększa zawartość PC o cztery po każdej operacji *Fetch* PC = PC + 4. Procesor pobiera cztery bajty (32 bity) z pamięci programu w jednym cyklu zegara, jak pokazano na rysunku 3.10

Procesory Cortex-M0/M3/M4 każdą instrukcję wykonują w trzech etapach, jak pokazano na rysunku 3.11.

- ***Fetch*** - w pierwszym etapie procesor pobiera instrukcję z pamięci programu i zwiększa licznik programu automatycznie, PC wskazuje na następną instrukcję do pobrania.

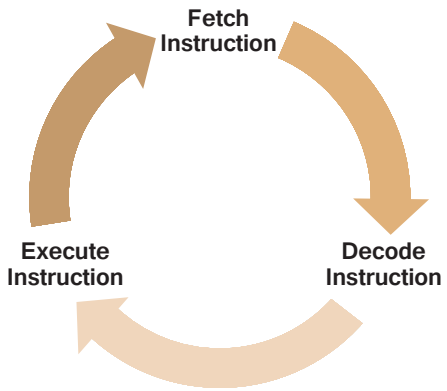
- ***Decode*** - procesor dekoduje instrukcję i już "wie co ma zrobić"
- ***Execute*** - etap wykonania: pobiera operandy z rejestrów i pamięci, wykonuje operację logiczną lub arytmetyczną, zapisuje wynik do rejestru lub pamięci.



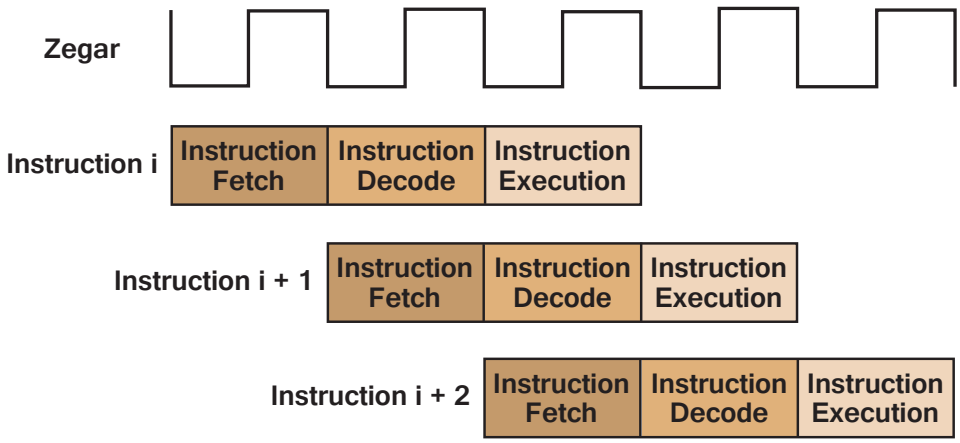
Rys. 3.10: Licznik programu jest zwiększany o 4 po każdym *Fetch*. Gdy program czyta PC, to jego wartość jest równa: (adres bieżącej instrukcji) + 4

Te trzy etapy są powtarzane aż do wyłączenia zasilania. Część procesora odpowiedzialna za wykonywanie programu jest podzielona na trzy specjalizowane elementy w ten sposób, że każdy wykonuje swoją część pracy. Jest to **potokowość** *pipelining*. Elementy te są połączone sekwencyjnie w potok (*pipe*) i wykonują pracę równocześnie, pobierając dane od poprzedniego elementu w sekwencji. W każdym z elementów rozkaz jest w innym stadium wykonania. Zgodnie z rysunkiem 3.12 przejście przez wszystkie etapy potoku (wykonanie jednej instrukcji) zajmuje co najmniej 3 takty zegara. Jednak dzięki pracy wszystkich elementów potoku, jednocześnie wykonywane są trzy instrukcje, każda w innym stadium wykonania. Średnio mamy więc jeden takt na jedną instrukcję. Zakłóceniem tego idealnego obrazu są rozkazy skoków i wywołań, które mogą powodować konieczność usunięcia z potoku już pobranych następnych rozkazów. Wiele procesorów stosuje próby przewidzenia, czy skok warunkowy będzie wykonany czy nie, aby pobrać właściwe następne instrukcje.

Procesory Cortex-M0, Cortex-M3 i Cortex-M4 mają trzy etapy potoku. Procesor Cortex-M0+ tylko dwa: *Fetch* i *Execution*, ponadto Cortex-M0+ jest zbudowany w architekturze *Von Neumann*. Natomiast najsilniejszy Cortex-M7 ma wiele potoków, takich jak potok *load/store*, dwa potoki ALU i jeden FPU. Dzięki temu wiele instrukcji może być wykonywanych jednocześnie. Każdy potok ma aż do sześciu etapów: *instruction fetch, decode, instruction issue, execute sub-stage1, execute sub-stage2* i *write back*.



Rys. 3.11: Cykl życia instrukcji



Rys. 3.12: Potokowy sposób wykonywania programu

3.4. Wykonywanie programu maszynowego

W tym rozdziale prześledzimy jak procesor Cortex-M wykonuje program. Napiszemy funkcję, która kopiuje string z miejsca źródłowego *source* `strSrc` do miejsca przeznaczenia *destination* `strDst` i zwraca liczbę przepisanych znaków. W standardzie C, każdy string kończy się znakiem NUL, czyli `0x00` (tabela znaków ASCII7 jest w dodatku B), więc przepisywanie trwa aż do napotkania bajtu zerowego.

```
int copyStr(char *dst, char *src){
   int counter = 0;
   while(*src){
     *dst++ = *src++;
     counter++;
   }
   return counter;
}
```

Wywołanie funkcji w programie:

```
n = copyStr(strDst, "Source String: 0123456789");
```

Obraz pamięci mikrokontrolera pokazany jest w tabeli 3.1. Pierwsza kolumna to adres, druga to kod instrukcji. Jak widać, kod programu i string źródłowy zostały umieszczone we Flashu.

Wywołanie funkcji w kodzie źródłowym C `n = copyStr(strDst, strSrc);` spowoduje (zgodnie z zasadami przekazywania parametrów do i z podprogramów), że program załaduje pierwszy parametr wywołania (adres miejsca przeznaczenia) do rejestru `r0`, a drugi parametr wywołania (adres źródła) do rejestru `r1` i wywoła podprogram obsługi funkcji. Ilość przepisanych znaków zostanie zwrócona w rejestrze `r0`.

Wywołanie oznacza, że do rejestru *Link Register* LR zostanie wpisany adres instrukcji będącej bezpośrednio za instrukcją wywołania (jest to adres powrotu), a do licznika programu PC adres wywoływanej funkcji, w naszym przykładzie PC ←`0x08000286`.

Tabela 3.1: Adresy w pamięci, kod maszynowy i program po deasemblacji

adres	kod	program asemblerowy
0x08000286	4602	MOV r2,r0 ; r2 = adres destination
0x08000288	2000	MOVS r0,#0x00 ; r0 = counter = 0
0x0800028A	E004	B loop2 ; skok do etykiety loop2
0x0800028C	F8113B01	loop1 LDRB r3,[r1],#0x01 ; pobierz z 'source'
0x08000290	F8023B01	STRB r3,[r2],#0x01 ; zapisz do 'destination'
0x08000294	1C40	ADDS r0,r0,#1 ; counter++
0x08000296	780B	loop2 LDRB r3,[r1,#0x00] ; pobierz z 'source'
0x08000298	2B00	CMP r3,#0x00 ; czy jest już znak NUL ?
0x0800029A	D1F8	BNE loop1 ; skocz, gdy nie koniec
0x0800029C	4770	BX lr ; powrót z 'counter' w r0
0x200002AC	6F 53 72 75	"Source String: 0123456789",0
0x200002B0	65 63 53 20	
0x200002B4	72 74 6E 69	
0x200002B8	3A 67 30 20	
0x200002BC	32 31 34 33	
0x200002BE	34 33 36 35	
0x200002C2	38 37 00 39	
0x200002C6		

Następną wykonaną instrukcją będzie więc:

0x08000286: `MOV r2,r0`, czyli *move*, przesuń. Instrukcja kopiuje drugi operand do pierwszego `r2` ← `r0`. W `r2` znajdzie się pierwszy parametr wywołania, czyli adres *destination*.

0x08000288: `MOVS r0,#0x00`, drugi operand (bezpośredni `#0x00`) zostanie wpisany do `r0`. Jest to inicjalizacja zerem licznika `counter`. Przyrostek `S` w mnemoniku instrukcji oznacza, że flagi statusu zostaną ustawione stosownie do wartości wpisanej do rejestru `r0`. Nie wykorzystamy tego, ale zastosowanie przyrostka skraca kod, gdyż instrukcja `MOVS r0,#0x00` jest 16-bitowa, a `MOV r0,#0x00`, 32-bitowa. Zaoszczędziliśmy 16 bitów kodu.

0x0800028A: `B loop2`, czyli *branch*, rozejście, rozgałęzienie, ale bez warunku, czyli jest to skok bezwarunkowy do etykiety. Adresowanie jest relatywne, czyli w kodzie instrukcji zawarte jest przesunięcie względem aktualnej wartości licznika programu. Dzięki takiemu adresowaniu uzyskujemy kod relokowalny: bez względu w które miejsce pamięci zostanie załadowany, to może być wykonywany bez żadnych zmian i nie trzeba go linkować (linker wpisuje aktualne adresy w te miejsca kodu, gdzie są one podane bezpośrednio, gdy zna już adres początku sektora) . Ponadto uzyskujemy skrócenie kodu: generujemy adres 32-bitowy na podstawie 12-bitowego przesunięcia. Postać binarna kodu instrukcji: bity najstarsze zawierają `11100`, a 11 najmłodszych to przesunięcie w kodzie uzupełnień do dwóch. Ponieważ kod programu jest wyrównywany do adresów parzystych, więc najmłodszy bit adresu jest zawsze zerowy i nie musimy go pamiętać. Bity przesunięcia należy

przesunąć o jeden w lewo i uzupełnić na najmłodszej pozycji zerem, co jest równoważne pomnożeniu przez dwa. Pamiętajmy, że licznik programu jest zwiększony o cztery. Ostatecznie formuła wyliczenia adresu do którego należy skoczyć jest następująca:

`PC` ← (adres bieżącej instrukcji) + 4 + 2 * (przesunięcie z kodu instrukcji)

W naszym przykładzie kod instrukcji to `0xE004`. Mamy więc przesunięcie o cztery, czyli adres, który zostanie wpisany do licznika programu PC jest równy `0x0800028A + 4 + 2 * 4 = 0x08000296` i jest to rzeczywiście adres etykiety `loop2`.

0x0800028C: `LDRB r3,[r1],#0x01`, czyli *load*, ładuj z pamięci do rejestru. Przyrostek `B` oznacza bajt bez znaku (`unsigned char`). Bajt z miejsca pamięci wskazywanego przez zawartość rejestru `r1` zostanie załadowany do `r3`, po czym do adresu w `r1` zostanie dodany trzeci operand, czyli `0x01`. Adres inkrementujemy o jeden, gdyż czytamy pojedyncze znaki. W `r1` jest drugi parametr wywołania, czyli adres źródła. Jest to instrukcja 32-bitowa. Procesor dekodując instrukcję 32-bitową "zasysa" z potoku brakujące 16 bajtów kodu i zwiększa PC o kolejne 4, czyli następną instrukcją jest:

0x08000290 `STRB r3,[r2],#0x01`, *store*, składuj. Instrukcja bliźniacza do `LDRB`, działa "w drugą stronę". Bajt z `r3` zapisz do miejsca pamięci wskazywanego przez `r2`, inkrementuj adres w `r2`.

0x08000294 `ADDS r0,r0,#1`, *add*, dodaj. Prefiks `S` oznacza ustaw flagi statusu odpowiednio do wyniku dodawania. Nie wykorzystujemy tych statusów, ale ta postać instrukcji jest 16-bitowa, natomiast `ADD` jest 32-bitowa. Skróciliśmy kod o kolejne 16-bitów.

0x08000296: `LDRB r3,[r1,#0x00]`, czyli *load*, ładuj z pamięci do rejestru. Bajt z miejsca pamięci wskazywanego przez zawartość rejestru `r1` zostanie załadowany do `r3`, adres w `r1` nie będzie zmieniony.

0x08000298 `CMP r3,#0x00`, *compare*, porównaj. Instrukcja odejmuje `r3 - 0x00`, operand bezpośredni od zawartości rejestru (drugi od pierwszego). Nie zapisuje wyniku, tylko ustawia flagi statusu. Jeżeli zawartość rejestru `r3` będzie zerowa, flaga *Zero* zostanie ustawiona na jedynkę, czyli *True*, w przeciwnym wypadku będzie wyzerowana, czyli *False*.

0x0800029A `BNE loop1`, *Branch if Not Equal*, skocz gdy nierówne. Zostanie wykonany skok do adresu relatywnego, gdy stan flagi *Zero* jest *False*, w przeciwnym przypadku idź dalej (do następnej instrukcji). W naszym przykładzie będzie skakał dopóki nie natrafi na zerowy bajt, czyli znak NUL. Adres docelowy skoku wylicza się według tej samej reguły, co poprzednio. W naszym przykładzie przesunięcie (tylko 8 najmłodszych bitów kodu instrukcji) to `0xF7`, czyli -9 w kodzie U2 (Tabela 2.4), razy dwa to -18, plus cztery to -14. Od adresu bieżącej instrukcji musimy odjąć 14, czyli otrzymujemy adres etykiety `loop1`.

Po przepisaniu całego stringu, z pamięci pobrany zostanie znak NUL, i skok `BNE` nie zostanie wykonany. Wtedy następną instrukcją będzie:

0x0800029C `BX lr`. Adres powrotu z rejestru `LR` zostanie przepisany do licznika programu `PC`. W ten sposób kolejną instrukcją do wykonania jest następna za wywołaniem

i wykonanie wróci do programu wywołującego. W momencie powrotu rejestr `r0` zawiera wartość zwracaną, czyli ilość znaków stringu, które zostały przepisane.

Rozdział 4

Zestaw uruchomieniowy z procesorem STM32F103VB

4.1. Wprowadzenie

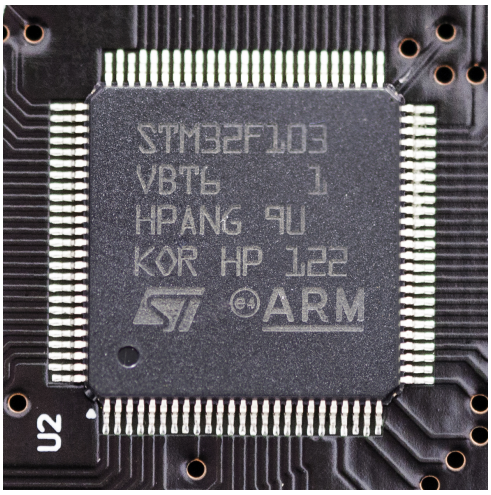
Nauka programowania mikrokontrolerów wymaga ćwiczeń. Najłatwiejszą drogą jest zakup prostego zestawu uruchomieniowego. My zaczniemy od **ZL27ARM** z dodatkowym wyświetlaczem alfanumerycznym **LCD1602** (dwie linie po 16 znaków). Potrzebne jeszcze będą: programator, np. **j-link** firmy Segger oraz komputer z zainstalowanym środowiskiem programistycznym **Keil μVision 5**. ZL27ARM zbudowany jest w oparciu o procesor **STM32F103VB**. W dalszych rozdziałach książki będziemy stosowali inny zestaw uruchomieniowy: STM3210E-EVAL z procesorem STM32F103ZE.

4.2. Rodzina mikrokontrolerów STM32

Firma ST Microelectronics www.st.com/ produkuje rodzinę mikrokontrolerów **STM32**. Wszystkie zawierają wbudowaną pamięć Flash i rdzeń, czyli korę (*core*) ARM Cortex-M. Tabela 4.1 pokazuje podstawowy podział rodziny STM32 na linie i maksymalną częstotliwość taktowania procesora oraz maksymalną osiąganą moc obliczeniową w każdej linii.

Tabela 4.1: Rodzina STM32

	Cortex-M0	**Cortex-M3**	**Cortex-M4**	**Cortex-M7**
High-Performance		STM32 **F2** 398 CoreMark 120 MHz 150 DMIPS	STM32 **F4** 608 CoreMark 180 MHz 225 DMIPS	STM32 **F7** 1'082 CoreMark 216 MHz 462 DMIPS
Mainstream	STM32 **F0** 106 CoreMark 48 MHz 38 DMIPS	**STM32 F1** 177 CoreMark 72 MHz 61 DMIPS	STM32 **F3** 245 CoreMark 72 MHz 90 DMIPS	
Ultra - Low - Power	STM32 **L0** 75 CoreMark 32 MHz 26 DMIPS	STM32 **L1** 93 CoreMark 32 MHz 33 DMIPS	STM32 **L4** 273 CoreMark 80 MHz 100 DMIPS	



Seria STM32F1 dzieli się na pięć linii

- STM32F100 - 24 MHz CPU ze sterowaniem silników i funkcjami CEC
- STM32F101 - 36 MHz CPU z Flash do 1MB
- STM32F102 - 48 MHz CPU z USB FS
- **STM32F103** - 72 MHz CPU z Flash do 1MB, sterowanie silników, USB i CAN
- STM32F105/107 - 72 MHz CPU z Ethernet MAC, CAN i USB 2.0 OTG

Mikrokontrolery linii STM32F103VB są produkowane w obudowach czterech rozmiarów: **T**x - 36 wyprowadzeń ("nóżek"), **C**x - 48, **R**x - 64, **V**x - 100 i **Z**x - 144 wyprowadzenia. Ostatni znak określa wielkość pamięci stałej Flash: **8** = 64kB, **B** = 128kB, **C** = 256kB, **D** = 384kB, **E** = 512kB, **F** = 768kB i **G** = 1MB. Ponadto, w zależności od wielkości Flasha firma STM dzieli mikrokontrolery na cztery gęstości upakowania układu scalonego

- **Low-density devices** - wielkość pamięci Flash od 16kB do 32kB
- **Medium-density devices** - Flash od 64kB do 128kB
- **High-density devices** - Flash od 256kB do 512kB
- **XL-density devices** - Flash od 768kB do 1MB

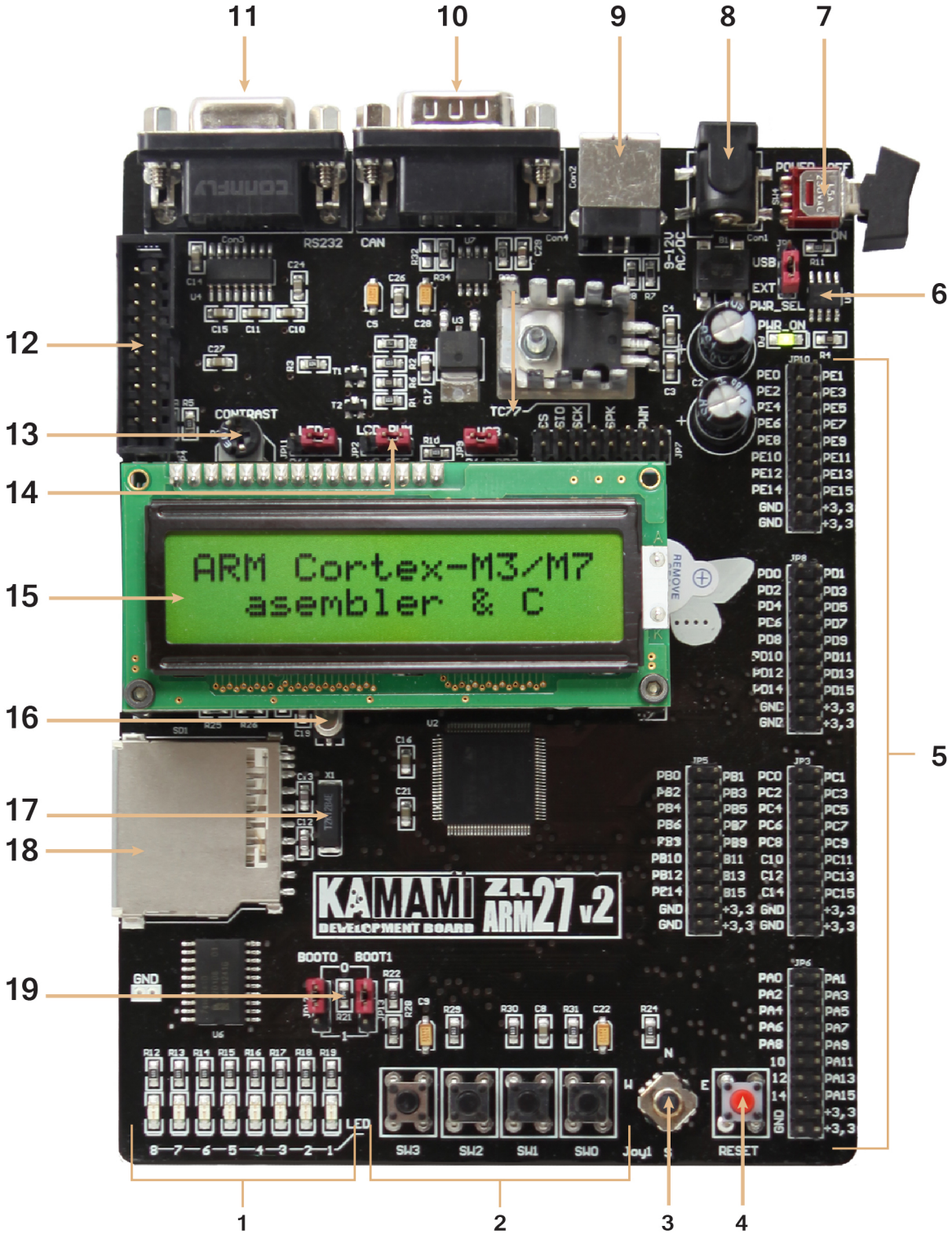
Mikrokontroler **STM32F103VB** ma rdzeń Cortex-M3, 72MHz CPU, 100 nóżek, 128kB Flash, 20kB RAM i jest klasyfikowany jako *medium-density*

4.3. Zestaw uruchomieniowy

Zestaw uruchomieniowy ZL27ARM wraz z modułem rozszerzającym LCD1602 można zamówić na stronie https://kamami.pl/. Zawiera on mikrokontroler STM32F103VB oraz dodatkowe elementy pokazane na rysunku 4.1. Schemat zamieszczono w Dodatku A.

1. Osiem diod LED sterowanych z portu GPIO
2. Cztery przyciski monostabilne, połączone równolegle z joystickiem
3. Joystick, jest równoważny pięciu przyciskom
4. Przycisk Reset
5. Porty równoległe GPIO: PA, PB, PC, PD, PE
6. Termometr cyfrowy Microchip TC77 z *interface* SPI
7. Wyłącznik zasilania
8. Gniazdo zasilania DC 9...12V
9. Gniazo USB, umożliwia zasilanie z komputera PC
10. Złącze DB9Male transmisji szeregowej CAN
11. Złącze DB9Female transmisji szeregowej RS232
12. Złącze J-TAG do podłączenia programatora
13. Potencjometr regulacji kontrastu wyświetlacza LCD
14. Wejście do układu regulacji jasności podświetlenia LCD sygnałem PWM

15. Moduł wyświetlacza LCD1602
16. Kwarc systemowy 8MHz
17. Kwarc zegarkowy 32kHz
18. Złącze kart pamięci
19. Zworki do ustalenia poziomów na wejściach BOOT0/1 mikrokontrolera



Rys. 4.1: Zestaw uruchomieniowy ZL27ARM

4.4. Dokumentacja Cortex-M3 i STM32F103

W dalszej pracy jest bardzo ważne, abyśmy dysponowali odpowiednią dokumentacją (czyli wiedzą). Szczególnie jak coś nie działa i nie wiemy dlaczego, odpowiedź można znaleźć w dokumentacji mikrokontrolera STM32 lub samego rdzenia ARM. Obie firmy: STMicroelectronics i ARM zamieszczają na swoich stronach pełną dokumentację: karty katalogowe, podręczniki i noty aplikacyjne.

Należy wejść na stronę www.st.com/ i wyszukać (*Search*) "STM32F103". Pojawi się tabela *Part Number Results (29)*, wybieramy STM32F103VB i wchodzimy na stronę poświęconą naszemu mikrokontrolerowi. W zakładce *Quick View* pobieramy *Datasheet*. Następnie w zakładce *Design Resources* w dziale *Reference manual* pobieramy dokument **RM0008**, a w dziale *Programming Manual* pobieramy **PM0056** oraz **PM0075**. Procesory też mają błędy i ograniczenia, dlatego od początku musimy uczyć się korzystania z wykazów wad publikowanych przez producentów. Dlatego z działu *Errata Sheet* pobieramy dokument **ES096**. Najważniejszym dokumentem na początkowym etapie programowania jest oczywiście podręcznik programowania PM0056. Zawiera on między innymi opis wszystkich instrukcji asemblerowych procesora Cortex-M3. Praktycznie, to pobrane dotychczas dokumenty są całkowicie wystarczające na początkowym etapie nauki. Bardzo szczegółowe dane dotyczące rdzenia Cortex-M3 znajdziemy w dokumentach, które należy pobrać ze strony infocenter.arm.com/. W zakładce *Contents* rozwijamy kliknięciem *Cortex-M series processors*, następnie rozwijamy *Cortex-M3* i najnowsze *Revision*. Klikamy na *ARM Cortex-M3 Processor Technical Reference Manual*, wchodzimy na tą stronę i na samym dole możemy pobrać wersję PDF. Następnie wchodzimy na stronę *Cortex-M3 Devices Generic User Guide* i też pobieramy PDF.

Wracamy do zakładki *Contents*. Klikamy na *ARM Architecture* i rozwijamy *Reference Manuals*. Pobieramy *ARMv7-M Reference Manual* (tylko wersja PDF) - musimy wcześniej podać swoje dane. Pobrane dokumenty zostały zestawione w tabeli 4.2

Tabela 4.2: Dokumentacja do pobrania

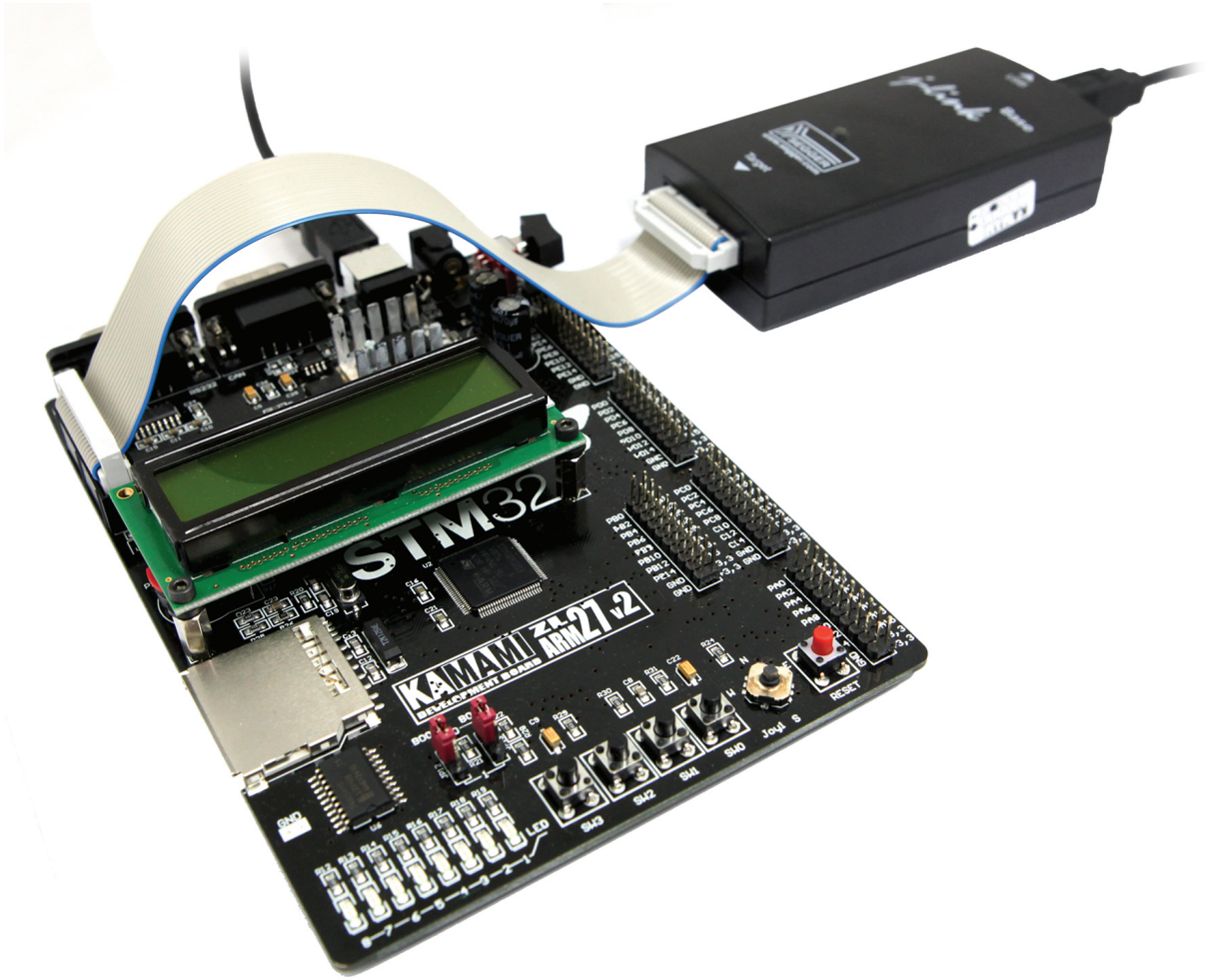
Oznaczenie	**Nazwa dokumentu STMicroelectronics**
PM0056	Cortex-M3 programming manual
PM0075	Flash memory programming manual
Datasheet	STM32F103x8/xB. Medium-density performance line ARM
RM0008	Reference manual
ES096	Errata sheet
Oznaczenie	**Nazwa dokumentu ARM**
ARM-Tech-Ref	ARM Cortex-M3 Processor Technical Reference Manual
ARM-User	Cortex-M3 Devices Generic User Guide
ARMv7-M	ARMv7-M Reference Manual

4.5. Programator

Po kompilacji kodu źródłowego C zapiszemy (za pomocą programatora) kod wykonywalny w wewnętrznej pamięci stałej Flash. Po Reset procesor rozpocznie wykonywanie

programu. Programator ułatwia także debugowanie (odpluskwianie) programu. Umożliwia pracę krokową (instrukcja po instrukcji), zakładanie pułapek, podgląd rejestrów wewnętrznych procesora i zawartości pamięci.

Wygodnym rozwiązaniem jest programator połączony z uruchamianą płytą za pomocą złącza JTAG. Standard JTAG (*Joint Test Action Group*) został zdefiniowany przez komitet IEEE (www.ieee.org) w roku 1990. Pozwala na testowanie zmontowanych płyt drukowanych i programowanie procesorów wlutowanych w te płyty (*In-System Programming, ISP*). Aby móc skorzystać w pełni z możliwości *interface*u, procesor musi mieć wbudowane odpowiednie bloki wspomagające debugowanie i śledzenie programu. Mikrokontrolery ARM mają bardzo rozbudowane możliwości debugowania z użyciem JTAG, daleko wykraczające poza standard. Mikrokontrolery Cortex-M4/M7 i niektóre M3, są ponadto wyposażone w układ *Embedded Trace Macrocell ETM*, który umożliwia (za pomocą złącza *ETM Trace debugging*) śledzenie programu i danych *on-line*, czyli bez zatrzymywania procesora, co może być szczególnie ważne w systemach czasu rzeczywistego. Będziemy z tego korzystać w dalszej części książki. W początkowym etapie nauki wystarczające jest zastosowanie najprostszego programatora firmy Segger www.segger.com/jlink-debug-probes.html *j-link BASE*, który występuje w wersji edukacyjnej *j-link EDU*. Wersja edukacyjna jest w bardzo niskiej, promocyjnej cenie i (w stosunku do *BASE*) nie ma ograniczenia na



Rys. 4.2: Zestaw uruchomieniowy z programatorem *j-link*

ilość pułapek *Flash Breakpoints*. Wszystkie programatory *j-link* wspierają między innymi procesory Cortex-M0/M0+/M1/M3/M4/M7, ponadto środowisko μVision5 współpracuje z *j-link*. Zestaw uruchomieniowy ZL27ARM z dołączonym programatorem *j-link BASE* pokazany jest na rysunku 4.2.

Zanim zaczniemy korzystać z programatora należy pobrać dokumentację oraz sterowniki ze strony producenta. Na stronie https://www.segger.com/jlink-debug-probes.html należy kliknąć w link Software and Dokumentation download. Zostanie otwarta strona "J-Link downloads". W dziale "Manuals" pobieramy dokument **UM08001** "J-Link manual". Następnie pobieramy programy odpowiednie do używanego systemu operacyjnego i instalujemy zgodnie zaleceniami podanymi w dokumencie UM08001 (Tabela 4.3).

Tabela 4.3: Dokumentacja programatora Segger *j-link*

Oznaczenie	**Nazwa dokumentu SEGGER Microcontroller GmbH**
UM08001	J-Link / J-Trace User Guide

4.6. Środowisko IDE Keil μVision

4.6.1. Zintegrowane środowisko programistyczne

Projekty opisane w książce zostały wykonane w zintegrowanym środowisku programistycznym *Integrated Development Environment IDE* firmy **Keil**. Wyboru dokonano z kilku powodów. Po pierwsze właścicielem firmy Keil jest ARM. Gwarantuje to pełną zgodność i obsługę rdzenia Cortex przez kompilator. Środowisko to jest w pełni profesjonalne i zawiera wszystkie potrzebne narzędzia programistyczne: edytor, kompilator C/C++, asembler, linker i rozbudowany debugger. Umożliwia symulację procesora, co pozwala prześledzić interesujące nas fragmenty kodu. Wprawdzie wersja bezpłatna pozwala pisać i uruchamiać programy o maksymalnym rozmiarze 32kB, ale na początek nam to wystarczy.

Zaczniemy od pobrania przewodnika po instalacji. Na stronie www.keil.com/support/ klikamy link View on-line product manuals, a następnie ARM Product Manuals. Z listy wybieramy MDK-ARM Getting Started i pobieramy dokument PDF.

4.6.2. Instalacja środowiska μVision

Teraz zainstalujemy zintegrowane środowisko programistyczne *Integrated Development Environment IDE*. Na stronie http://www.keil.com klikamy MDK Version 5 Microcontroller Development Kit. Tam klikamy na link Download & Instal, który przekierowuje nas na stronę "Getting Started". Klikamy Download MDK-Core. Musimy teraz podać swoje dane kontaktowe i możemy pobrać program instalacyjny. Wykorzystamy bezpłatną wersję MDK-Lite, która ma kod ograniczony do 32kB.

Do pracy w środowisku IDE potrzebujemy jeszcze "Device Family Pack" naszego procesora STM32F103VB. Jeżeli zmienimy procesor musimy doinstalować przypisany do niego "Device Family Pack". Służy do tego **Pack Installer** uruchamiany w czasie instalacji, ale można go wywołać w μVision *Project* > *Manage* > *Pack Installer*. Strona www.keil.com/dd2/ jest bazą mikrokontrolerów obsługiwanych przez Keil μVision. Rozwijamy listę STMicroelectronics, następnie podlistę STM32F1 Series i klikamy na

STM32F103VB. Ze strony STM32F103VB możemy pobrać "Device Family Pack" klikając Download.

4.7. Cortex Microcontroller Software Interface CMSIS

W projektach będziemy wykorzystywać standardową bibliotekę CMSIS dostarczaną przez producenta mikrokontrolera STMicroelectronics. Na stronie www.st.com/ do wyszukiwarki *Search* wpisujemy "STM32F10x Standard Peripheral Library". Z listy wybieramy STSW-STM32054 opisany jako *STM32F10x standard peripheral library*. Pobieramy przyciskiem *Download*. Na dysku tworzymy katalog ...*ARM* i w nim rozpakowujemy (.zip). Zostanie utworzony katalog *STM32F10x_StdPeriph_Lib_V3.5.0* ze strukturą plików jak na rysunku 4.3.

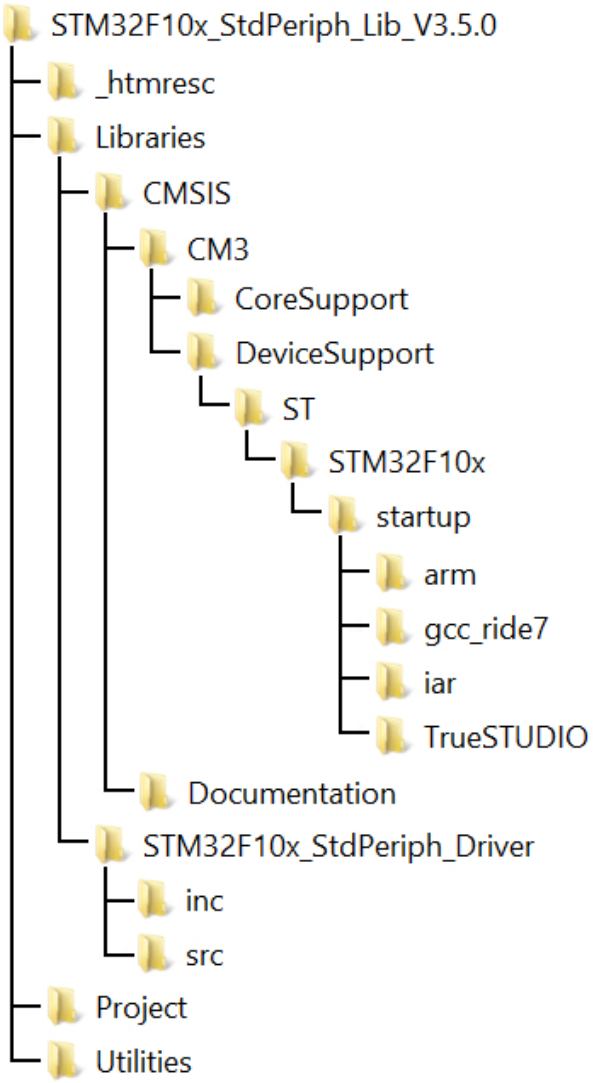
CMSIS jest standardową biblioteką procesora Cortex opracowaną przez firmę ARM. Producenci mikrokontrolerów Cortex także dostarczają biblioteki CMSIS zgodne ze standardem ARM i dostosowane do produkowanych przez siebie układów. Biblioteka CMSIS zawiera definicje i deklaracje stałych (adresy portów), typów zmiennych i struktur oraz funkcje obsługi rdzenia i wewnętrznych układów peryferyjnych mikrokontrolera.

Zalety stosowania biblioteki CMSIS

- Umożliwia przenoszenie programów i wykorzystywanie tego samego kodu w różnych projektach Cortex-M, redukując wysiłek związany z projektowaniem i weryfikacją.
- Zwiększa kompatybilność programów dzięki jednolitej strukturze API (*Application Programming Interface*) funkcji dostępu do rdzenia, metody inicjalizacji systemu, dzięki jednakowemu stylowi definiowania peryferiali. Programy z różnych źródeł mogą współpracować, z małym ryzykiem związanym z ich integracją.
- CMSIS umożliwia łatwy dostęp do zasobów rdzenia oraz wbudowanych peryferiali z poziomu języka C. Nie wymaga studiowania wszystkich szczegółów architektury procesora. Ten, który nauczył się programować określony procesor Cortex-M, łatwo zacznie pracę z innym, dzięki jednolitej strukturze definicji i funkcji biblioteki CMSIS. Wystarczy wymienić bibliotekę na właściwą dla nowego procesora i wcześniej napisane programy będą pracowały z nowym mikrokontrolerem, gdyż wszystkie wersje CMSIS tak samo definiują nazwy i funkcje.

W katalogu *_htmresc* są dwa obrazki. Katalog *Utilities* zawiera przykłady dla zestawów uruchomieniowych firmy STM. Katalog *Project* podzielony jest na podkatalogi *STM32F10x_StdPeriph_Examples* i *STM32F10x_StdPeriph_Template*. Pierwszy zawiera pouczające przykłady wykorzystania poszczególnych układów peryferyjnych mikrokontrolerów STM32F10x. W drugim są wzorcowe pliki projektów pięciu popularnych środowisk programowania. Nas najbardziej interesuje jednak katalog *Libraries*, który podzielony jest na dwa podkatalogi: *CMSIS* oraz *STM32F10x_StdPeriph_Driver*.

W katalogu *\Libraries\CMSIS\CM3\DeviceSupport\ST\STM32F10x\startup\arm* są pliki startowe dla środowiska Keil μVision. Procesor STM32F103VB jest klasyfikowany jako *medium-density* (patrz strona 40). Plikiem właściwym jest *startup_stm32f10x_md.s*. Końcówka *_md* w nazwie pliku oznacza *medium-density*. Plik startowy zawiera definicje rozmiarów stosu i sterty, definiuje wektor adresów obsług przerwań, a na końcu wykonuje skok do funkcji *main*.



Rys. 4.3: Struktura katalogu CMSIS

W katalogu *Libraries**CMSIS**CM3**CoreSupport* są pliki *core_cm3.c* i *core_cm3.h*. Natomiast w katalogu *Libraries**CMSIS**CM3**DeviceSupport**ST**STM32F10x* znajdziemy pliki *stm32f10x.h*, *system_stm32f10x.c* oraz *system_stm32f10x.h*. Pliki te zawierają definicje adresów wszystkich rejestrów procesora i układów peryferyjnych, definicje poszczególnych bitów w rejestrach sterujących oraz podstawowe funkcje obsługi rdzenia.

Drugi podkatalog *STM32F10x_StdPeriph_Driver* zawiera funkcje obsługi układów peryferyjnych. Podzielony jest na katalog plików nagłówkowych *inc* i katalog kodów źródłowych *src*. Końcówki nazw ułatwiają rozpoznanie pliku. Końcówka *_gpio* oznacza funkcje obsługi portów GPIO, końcówka *_rtc* to funkcje obsługi zegara czasu rzeczywistego RTC, i tak dalej.

4.8. Nasz pierwszy: Projekt00

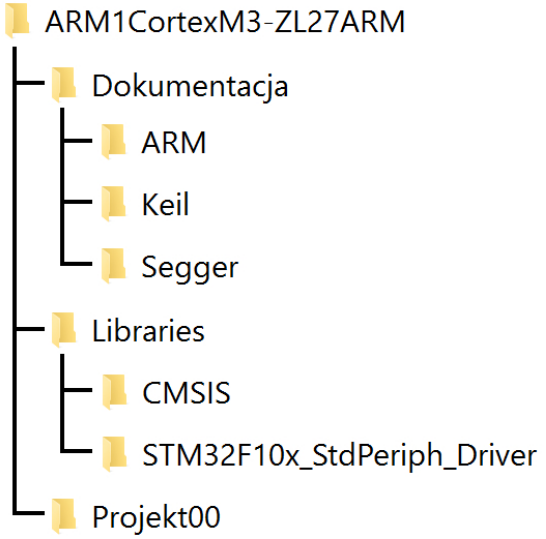
Przygotujemy teraz płytę ZL27ARM do pierwszego projektu. Płyta może być zasilana z portu USB komputera PC (złącze **9** na rysunku 4.1) lub z zewnętrznego źródła napięcia o wartości 9... 12V (złącze **8** na rysunku 4.1). W zależności od źródła zasilania musimy ustawić odpowiednio zworkę PWR_SEL: na pozycję USB, gdy zasilamy

z USB, albo na pozycję EXT gdy ze źródła zewnętrznego. Zestaw wyposażony jest w wyłącznik (**7** na rysunku 4.1), który umożliwia wyłączenie zasilania bez konieczności odłączania kabla zasilającego. Zielona dioda LED sygnalizuje załączenie zasilania. Obie zworki BOOT0 i BOOT1 (**19** na rysunku 4.1) umieszczamy w pozycji zero. Dzięki temu program będzie wykonywany z wewnętrznego Flasha. Wyłączamy komputer i do pierwszego portu USB podłączamy programator. Drugi port USB łączymy kablem ze złączem **9** (PWR_SEL na USB). Do złącza JTAG płyty **12** dołączamy programator jak na rysunku 4.2. Teraz możemy włączyć komputer. Nasz pierwszy *Projekt00* to będzie symulacja pracy procesora. Zaczniemy od utworzenia struktury katalogów. Tworzymy katalog:

ARM1CortexM3-ZL27ARM a w nim katalog *Dokumentacja* i kopiujemy zebrane dokumenty w podkatalogach jak na rysunku 4.4. W katalogu *Keil* umieszczamy PDF "MDK-ARM Getting Started". Następnie kopiujemy cały podkatalog *Libraries* z katalogu *STM32F10x_StdPeriph_Lib_V3.5.0*. Na końcu tworzymy pusty katalog *Projekt00*. Uzyskana struktura katalogów powinna wyglądać tak jak na rysunku 4.4. Do katalogu *Projekt00* kopiujemy pliki

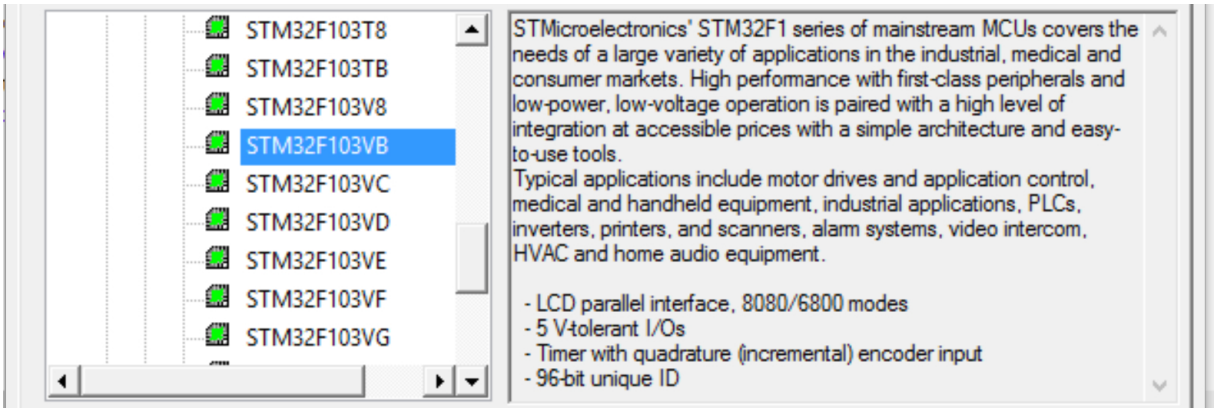
stm32f10x_conf.h
stm32f10x_it.c
stm32f10x_it.h

(są dołączone do książki).



Rys. 4.4: Struktura katalogów

Utworzymy teraz pierwszy projekt w środowisku programowania. Należy uruchomić Keil μVision5. Następnie należy wybrać *Project > New uVision Project...*. W oknie, które się otworzy wybieramy katalog *Projekt00* i podajemy nazwę pliku *projekt00*. Naciskamy *Zapisz* i pojawia się okno wyboru procesora. Wybieramy **STM32F103VB**. Następne okno *Manage Run-Time Environment* proponuje dołączenie bibliotek, ale nic nie wybieramy i kończymy naciskając *OK*. Okno wyboru procesora pokazane jest na rysunku 4.5. Jeżeli w oknie nie ma katalogu *STMicroelectronics*, to należy dołączyć właściwą bibliotekę. Instalacja opisana jest w dokumencie "MDK-ARM Getting Started".



Rys. 4.5: Wybór procesora

W oknie plików projektu (*Project*) rozwijamy katalog *Target 1*. Zaznaczamy utworzoną automatycznie grupę *Source Group 1*. Naciskamy klawisz F2 i zmieniamy nazwę na *UserCode*. Klikamy prawym klawiszem myszy na katalogu *Target 1*, wybieramy *Add Group...* i zmieniamy nazwę nowej grupy na *StdPeriphDrv*. W podobny sposób należy utworzyć grupy *CMSIS* oraz *RVMDK*.

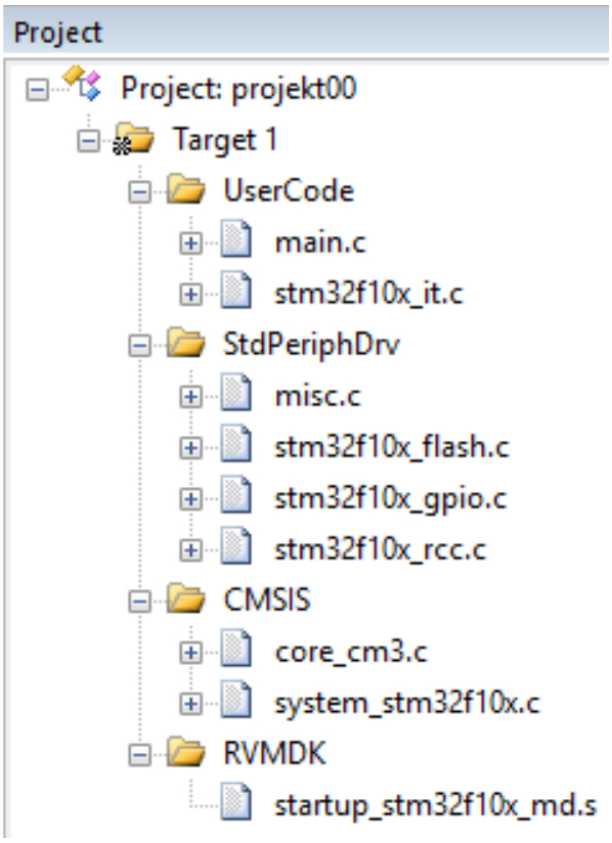
Po utworzeniu struktury projektu, z menu wybieramy polecenie *File > New...* i tworzymy plik *main*. W otwartym nowym pliku wpisujemy

```
#include "stm32f10x.h"
int main(void){
   return 0
}
```

Zapisujemy ten plik *File > Save As...* pod nazwą *main.c*. Kolejnym krokiem jest dołączenie pliku *main.c* do projektu. Jeżeli tego nie zrobimy, program w nim zawarty nie zostanie skompilowany i nie będzie dołączony do kodu wynikowego. Aby dodać plik do projektu, w oknie *Project*, klikamy prawym klawiszem na grupę *UserCode* i wybieramy polecenie *Add Existing Files to Group 'UserCode'...*. Z listy wybieramy *main* i dodajemy klawiszem *Add*. Plik *main.c* powinien pojawić się w grupie *UserCode*. W kolejnym kroku, z katalogu *Projekt00*, do grupy *UserCode* dodajemy

stm32f10x_it.c

Wszystkie kolejne pliki, które będziemy dołączać w następnych krokach muszą pochodzić z katalogu *Libraries*, który jest w naszym podstawowym *ARM1CortexM3-ZL27ARM*. Ułatwi to tworzenie nowych projektów przez kopiowanie starych. Kolejna grupa to *StdPeriphDrv*, dodamy następujące pliki z katalogu *Libraries\STM32F10x_StdPeriph_Driver*



Rys. 4.6: Struktura plików Projekt00

misc.c
stm32f10x_flash.c
stm32f10x_gpio.c
stm32f10x_rcc.c

Do grupy *CMSIS* dodamy pliki z dwóch katalogów. Najpierw plik z katalogu *Libraries\CMSIS\CM3\CoreSupport*

core_cm3.c,

oraz z katalogu *Libraries\CMSIS\CM3\DeviceSupport\ST\STM32F10x*

system_stm32f10x_conf.c

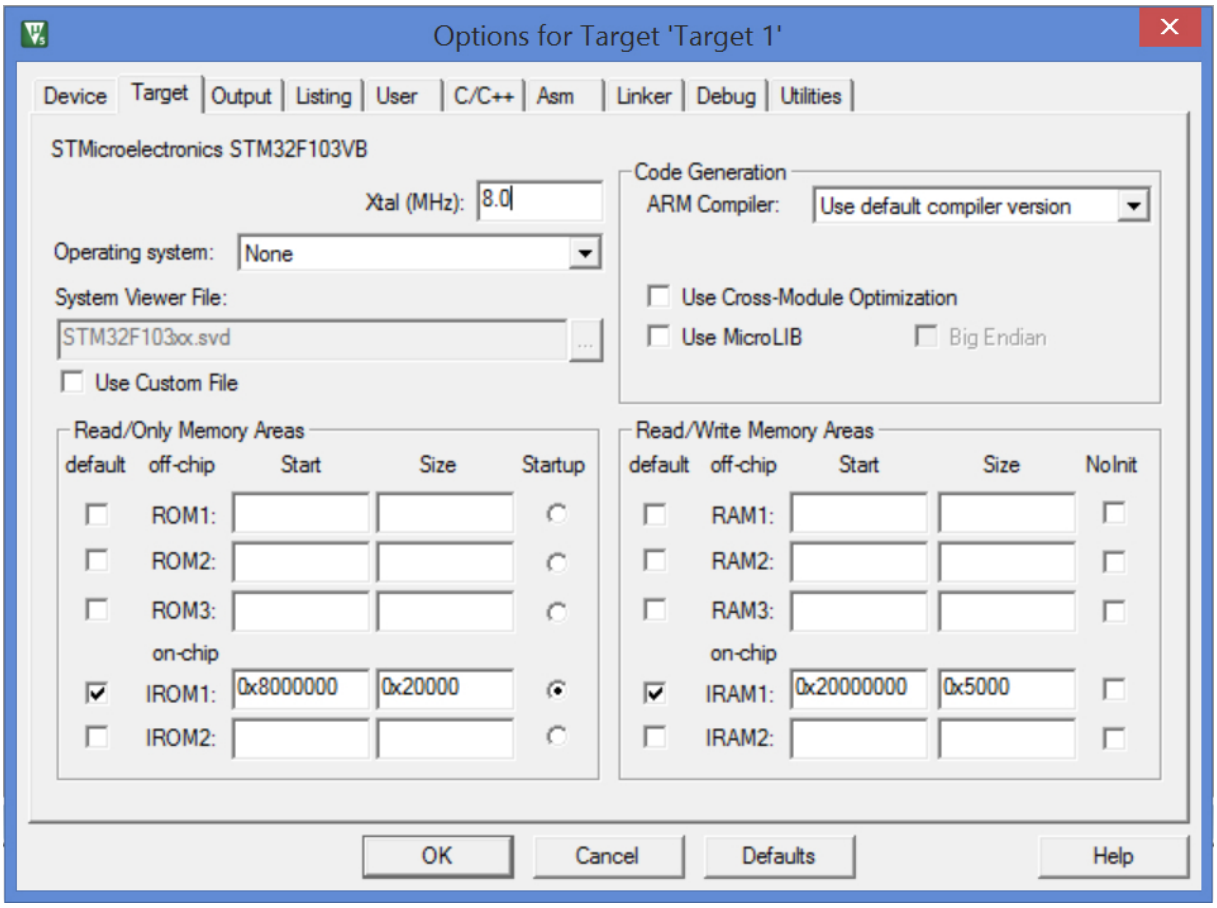
Pozostała nam grupa *RVMDK* i tu musimy sięgnąć najgłębiej, ale tylko po jeden plik, z katalogu: *Libraries\CMSIS\CM3\DeviceSupport\ST\STM32F10x\startup\arm*

startup_stm32f10x_md.s
Do projektu dodaliśmy funkcje obsługi układu generacji sygnałów taktujących RCC, obsługi portów równoległych GPIO i wewnętrznego Flasha.

4.8.1. Opcje projektu

Kolejną ważną czynnością jest ustawienie opcji projektu. Klikamy ikonę . Inna możliwość: klikamy prawym klawiszem myszy na najwyższym elemencie struktury plików *Target 1* i wybieramy polecenie *Options for Target 'Target 1...'* Przeglądamy kolejno wszystkie zakładki.

Device. Można tu zmienić typ procesora, ale przy tworzeniu projektu wybraliśmy STM32F103VB i jeżeli ten procesor jest wyświetlany, to nic nie zmieniamy (rysunek 4.5).



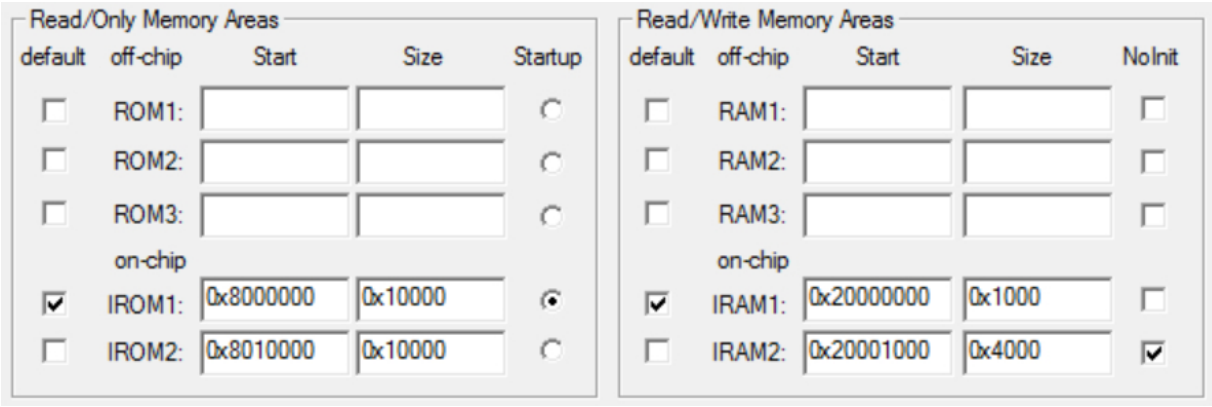
Rys. 4.7: Opcje projektu, zakładka *Target*

Target. Zakładka *Target* zawiera podstawowe, globalne opcje projektu. Definiuje, na przykład rozmieszczenie pamięci w projekcie. Przydział proponowany przez IDE pokazany jest na rysunku 4.7. Procesor STM32F103VB zawiera 128kB pamięci Flash i 20kB statycznego RAMu. Flash zaczyna się od adresu `0x800000`, 128kB to 2^{7+10} (patrz tabela 2.3), czyli `0x20000`. Pamięć S-RAM zaczyna się od adresu `0x200000`, 20kB to $20 = 16 + 4$

$$1'01\ 00'00\ 0000\ 0000_2 = 0x5000.$$

Jeżeli chcemy zdefiniować bardziej złożoną strukturę pamięci, to możemy dokonać podziału pamięci na obszary. Przykład pokazano na rysunku 4.8. Pamięś stała Flash została

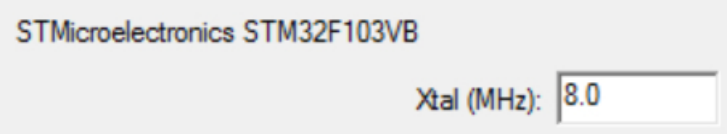
podzielona na obszar IROM1, w którym zostanie umieszczony program i stałe programu. W obszarze IROM2 linker nie umieści niczego, dopóki nie otrzyma odpowiedniej komendy. Podobnie pamięć S-RAM została podzielona na dwa obszary: wykorzystywany będzie tylko IRAM1 oznaczony jako *default*.



Rys. 4.8: Podział pamięci na obszary w zakładce *Target*

Gdy linker buduje program, to kod od etykiety "`RESET`" umieszcza w bloku pamięci oznaczonym jako *Startup*. Ten kod (tak zwana "rozbiegówka") przydziela miejsce na stos i stertę, wpisuje zmienne zainicjowane do RAMu oraz zeruje obszar pamięci przeznaczony na zmienne niezainicjowane. Jeżeli zaznaczymy opcję *NoInit*, na przykład dla obszaru IRAM2, to zawartość tego obszaru nie zostanie zmieniona w trakcie inicjalizacji systemu. Jest to użyteczne, gdy chcemy wykonać restart programowy, a w obszarze IRAM2 mamy istotne dane.

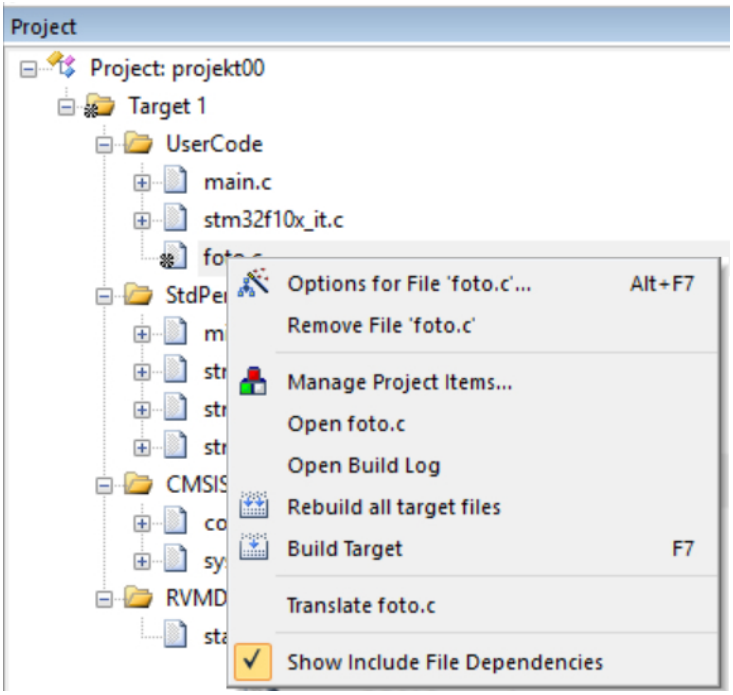
Możemy także spowodować, że określone dane będą wpisane do wybranego przez nas obszaru pamięci. Na przykład: do projektu jest dołączony plik *foto.c* ze zdjęciem, które wyświetlimy na ekranie graficznym. Klikamy prawym klawiszem myszki na tym pliku i wybieramy opcję *Options for File 'foto.c'* – rysunek 4.10. Zostanie otwarte okno pokazane na rysunku 4.11, w którym możemy wybrać obszar pamięci na dane zawarte w pliku.



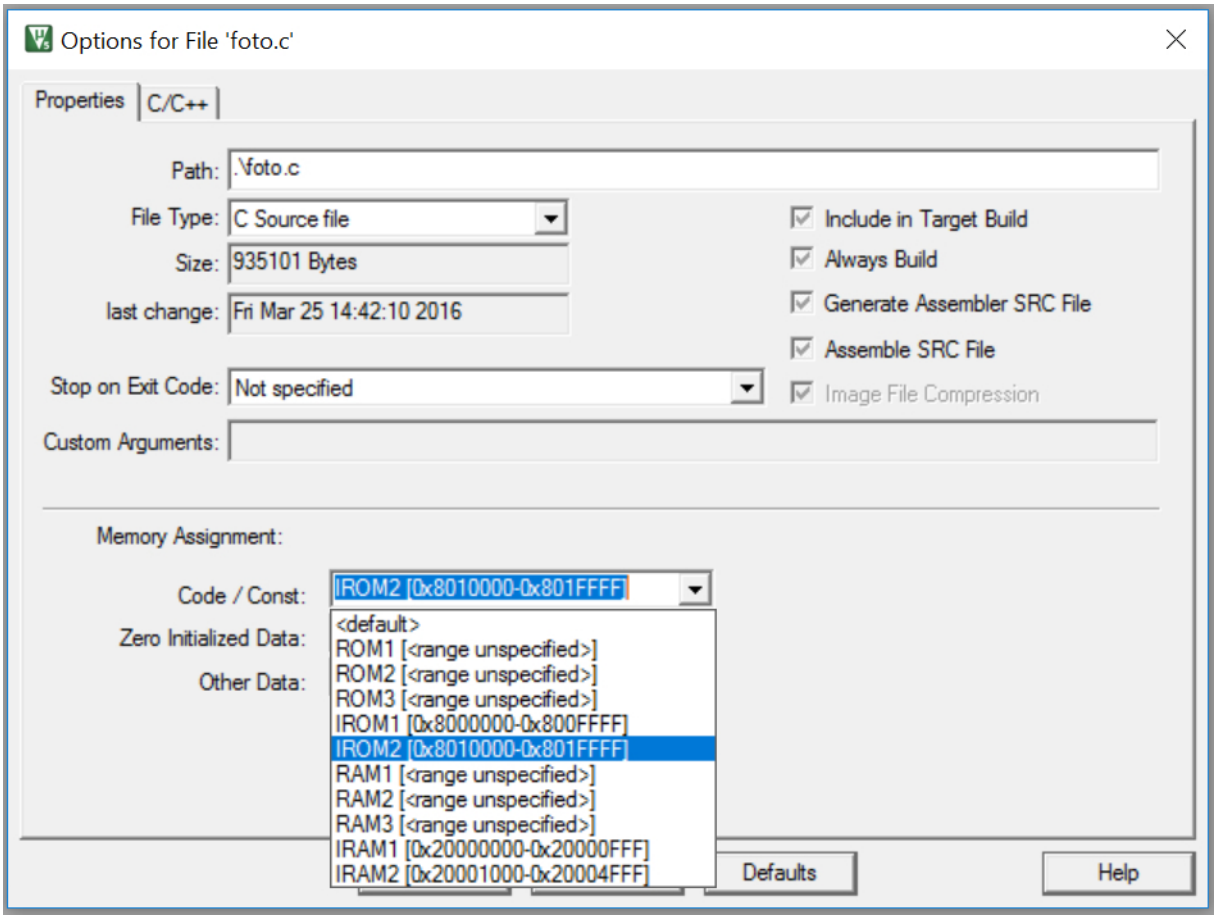
Rys. 4.9: Częstotliwość kwarcu jest używana jedynie przez symulator

W oknie *Target* należy także wpisać częstotliwość kwarcu zamontowanego na płytce. W naszym przypadku jest to 8MHz – rysunek 4.9. Umożliwi to, po uwzględnieniu konfiguracji układu taktowania mikrokontrolera RCC, właściwe symulowanie pracy systemu.

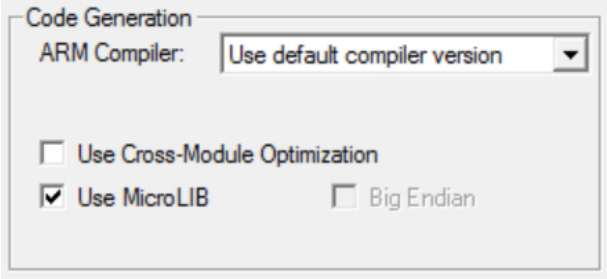
Środowisko programistyczne Keil MDK-ARM zawiera dwie biblioteki ANSI. Pierwsza to pełna biblioteka zgodna z aktualnym standardem języka C/C++. Druga, mniejsza to Keil MicroLIB zgodna ze standardem C99, która nie zawiera na przykład arytmetyki zmiennopozycyjnej liczb długich (`double`). W prostych aplikacjach, często wystarczające jest zastosowanie biblioteki MicroLIB, przez zaznaczenie opcji jak na rysunku 4.12. Dołączany do programu kod biblioteki MicroLIB jest wtedy ponad połowę krótszy od kodu biblioteki standardowej – tabela 4.4.



Rys. 4.10: Wywołanie opcji pliku *foto.c* dołączonego do projektu



Rys. 4.11: Dane z pliku *foto.c* będą wpisane do IROM2

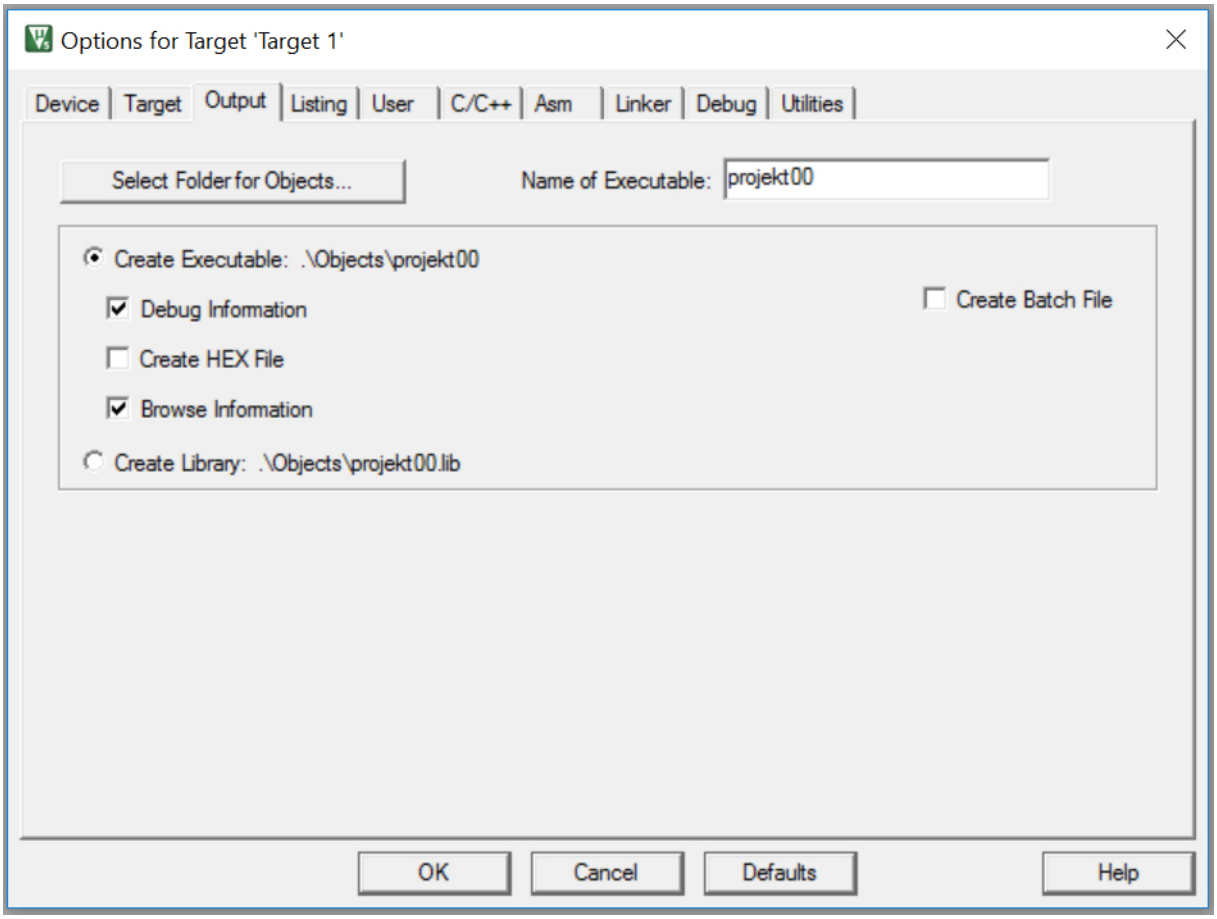


Rys. 4.12: Wybór biblioteki MicroLIB

Tabela 4.4: Porównanie długości kodów bibliotek dołączanych do programu

Procesor	Zestaw instrukcji		Standard	MicroLIB	Oszczędność
Cortex-M3/M4	Thumb-2	Kod	15'018	5'796	63 %
		ROM	18'616	8'976	54 %

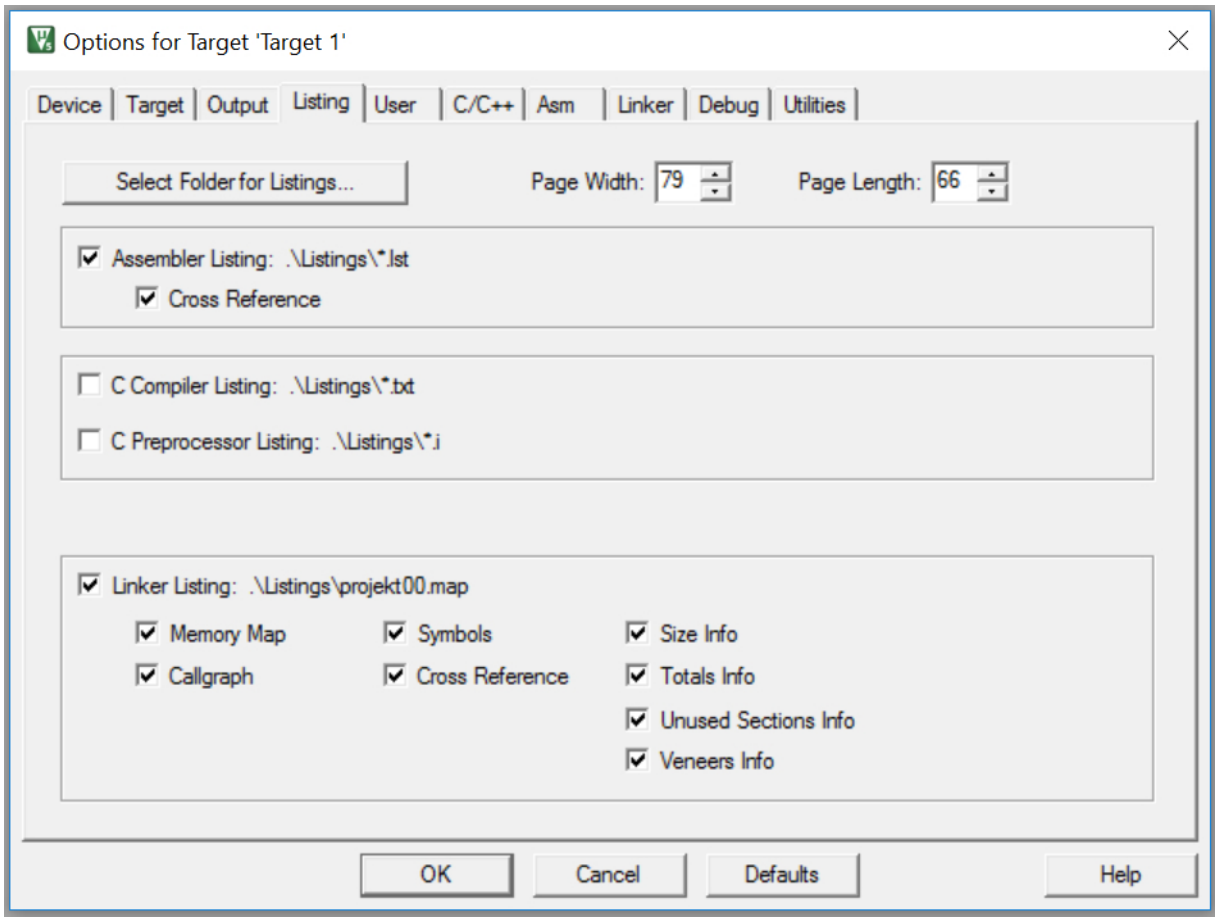
Możliwy jest wybór jeszcze jednej opcji: *Use Cross-Module Optimization* (rysunek 4.12). Spowoduje to wykonanie kilku przebiegów linkera w celu pełnej optymalizacji kodu. W początkowym etapie pracy wybór tej opcji nie jest zalecany, gdyż powoduje ona znaczne zmiany w kodzie programu: zanika bezpośredni związek pomiędzy kodami źródłowym i wynikowym. Utrudnia to debugowanie programu.



Rys. 4.13: Opcje projektu, Zakładka *Output*

Output. Są to opcje tworzenia plików wynikowych. Możemy wybrać pomiędzy samodzielnym, kompletnym programem, a biblioteką, którą dołączymy do innego projektu. Opcją domyślną jest program samodzielny z generacją informacji potrzebnych do debugowania: zaznaczone *Create Executable* oraz *Debug Information*. Włączymy także opcję *Browse Information*. Dzięki tej opcji będą tworzone informacje dla narzędzia *Source Browser*, które umożliwia wyszukiwanie funkcji i zmiennych w plikach źródłowych projektu. Jest to bardzo przydatne narzędzie z którego będziemy często korzystać. Zakładka *Output* umożliwia także zdefiniowanie oddzielnego folderu na pliki wynikowe: *Select Folder for Objects...*

Gdy projekt jest zakończony, możemy włączyć opcję *Create HEX File*. Spowoduje to wygenerowanie pliku *.hex* używanego przez programatory, w trakcie produkcji regulatorów zbudowanych w oparciu o nasz mikrokontroler. Jeżeli chcemy stworzyć bibliotekę, która będzie wykorzystywana w innym projekcie, należy włączyć opcję *Create Batch File*. Będzie wygenerowany plik *<nazwa_projektu>.bat*, który może być wykorzystany poza środowiskiem Keil MDK-ARM.



Rys. 4.14: Opcje projektu, Zakładka *Listing*

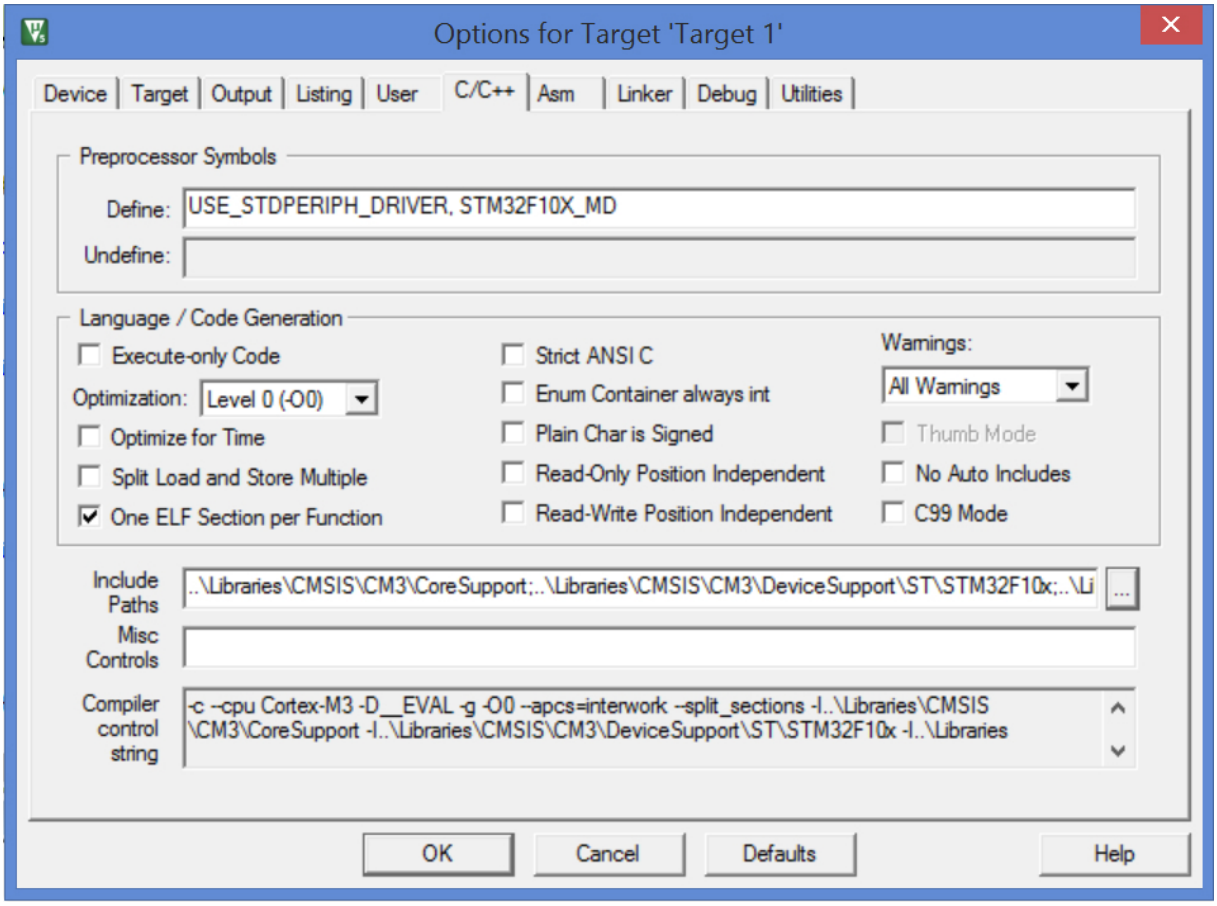
Listing. Opcje tworzenia listingów asemblerowych podczas kompilacji. IDE proponuje tworzenie listingów w katalogu *Listings*, który sam utworzył w czasie tworzenia projektu. Tworzone listingi są bardzo szczegółowe i dostarczają informacji o adresach bezwzględnych wszystkich zmiennych i funkcji projektu. Możemy włączyć opcję *C Compiler Listing*. Będą wtedy tworzone listingi z kompilacji wszystkich plików projektu: można prześledzić, jak kompilator tworzy kod maszynowy.

User. Ta zakładka zawiera pola w które można wpisać dodatkowe polecenia (*User Command*). Będą one wykonywane na poszczególnych etapach : przed kompilacją, przed i po konsolidacji. Zakładkę pozostawiamy bez zmian.

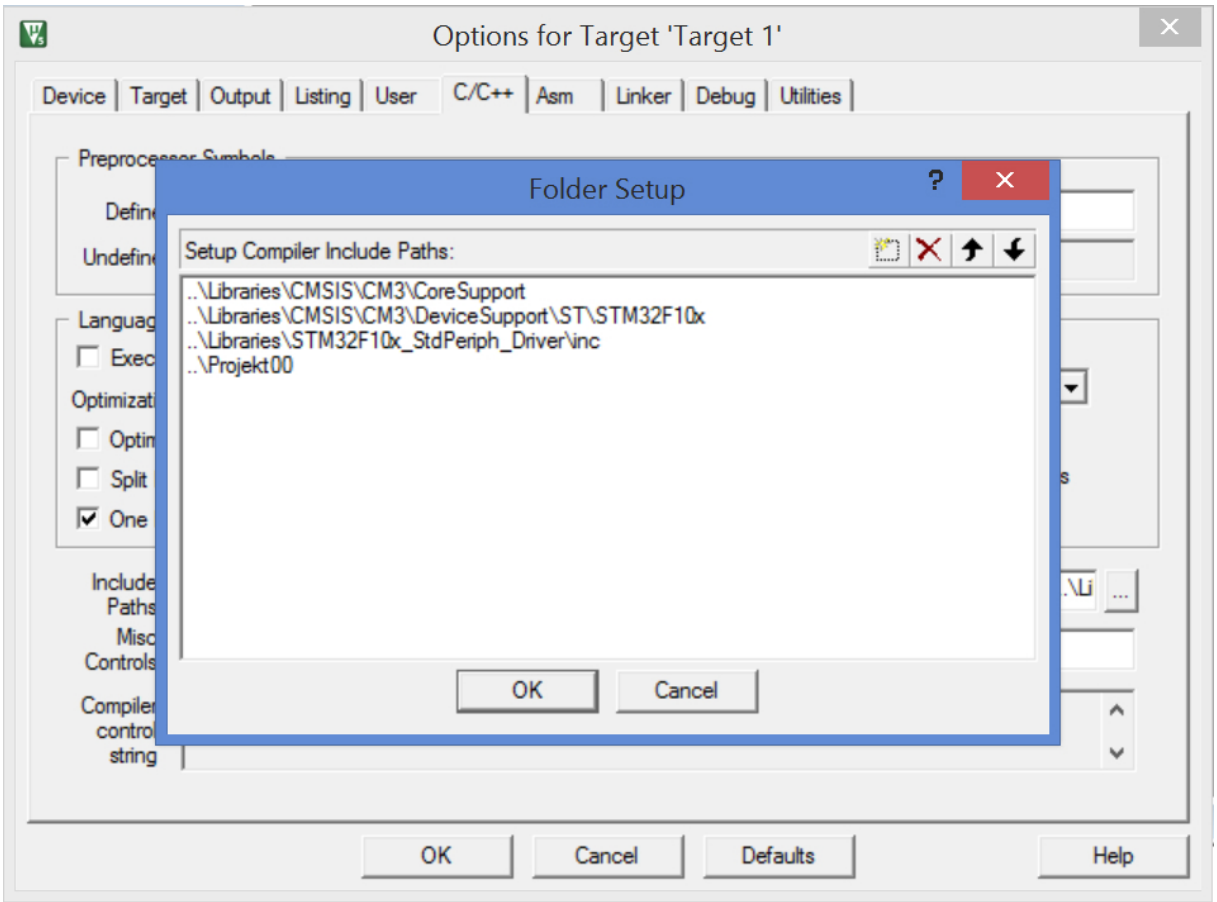
C/C++. Zakładka zawiera opcje kompilatora języka C. W polu *Define* wpiszemy dwie definicje rozdzielone przecinkiem. Pierwsza USE_STDPERIPH_DRIVER oznacza, że jest wykorzystywana biblioteka driverów peryferiali. Druga STM32F10X_MD oznacza, że program będzie kompilowany dla mikrokontrolera klasyfikowanego jako *medium-density*.

W dziale *Language/Code Generation* niczego nie zmieniamy. W szczególności należy pozostawić opcję optymalizacji kodu na poziomie zerowym. Jest to szczególnie ważne w początkowym okresie nauki, kiedy będziemy często obserwować jak kompilator tłumaczy kod źródłowy C na instrukcje maszynowe procesora. W debugerze będą one wyświetlane w postaci listingu asemblerowego. Porównywanie kodu maszynowego ze źródłowym C pomaga zrozumieć sposób działania procesora. Włączenie wyższego poziomu optymalizacji spowodowałoby, że kod maszynowy stałby się bardziej optymalny, ale zgubilibyśmy bezpośrednie odniesienie do źródła.

Kompilator optymalizuje długość kodu. Jeżeli kryterium optymalizacji ma być najkrótszy czas wykonania, należy zaznaczyć opcję *Optimize for Time*. Opcję *One ELF Section per Function* pozostawiamy włączoną. Oznacza ona, że w pliku ELF każda funkcja będzie zapisana w oddzielnym sektorze.



Rys. 4.15: Opcje projektu, Zakładka *C/C++*



Rys. 4.16: Zakładka *C/C++*, ścieżki dostępu

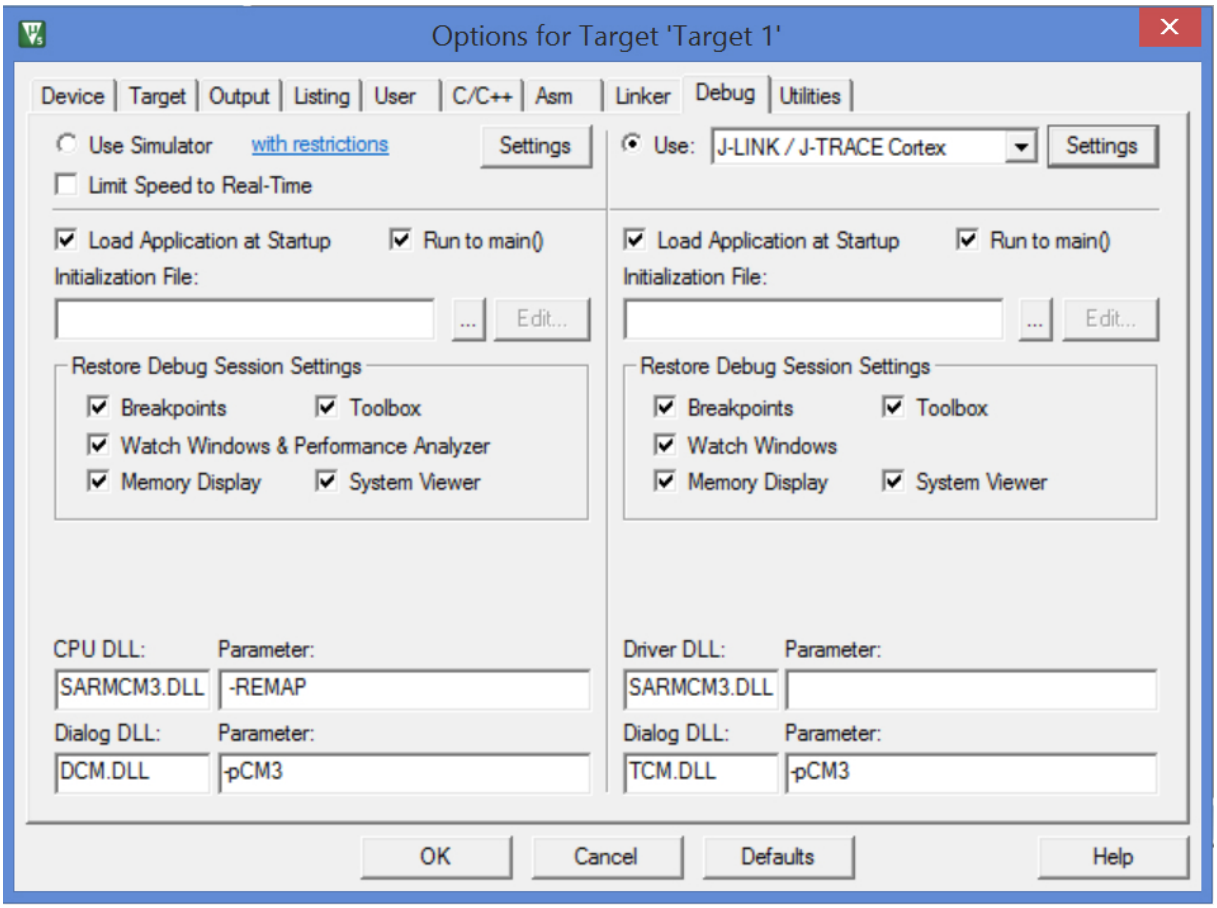
Także bez zmian pozostawiamy opcje kompilacji w polu *Compiler control string*. Są to opcje typowe, właściwe dla większości projektów. Szczegółowy opis opcji kompilacji zawarty jest w dokumencie **armcc User Guide**, który jest dostępny w zakładce *Books* u dołu okna *Projects* pod nazwą "ARM Compiler v5.06 for μVision (PDF)".

Ostania (ale bardzo ważna) czynność w tej zakładce to określenie ścieżek dostępu do plików. Jeżeli edytor pliku źródłowego zasygnalizuje błąd braku pliku, mimo że został on dołączony do projektu, to znaczy że ścieżka do niego albo nie została zdefiniowana, albo jest zdefiniowana źle. Klikamy przycisk z trzema kropkami po prawej stronie pola *Include Paths*. Pojawia się okno do którego wpisujemy kolejno ścieżki (rysunek 4.16)

- *ARMCortexM3\Libraries\CMSIS\CM3\CoreSupport*
- *ARMCortexM3\Libraries\CMSIS\CM3\DeviceSupport\ST\STM32F10x*
- *ARMCortexM3\Libraries\STM32F10x_StdPeriph_Driver\inc*
- *ARMCortexM3\Projekt00*

Asm. Opcje asemblera, pozostawiamy bez zmian. Szczegółowy opis opcji asemblera zawarty jest w dokumencie **armasm User Guide**, który jest dostępny w zakładce *Books* u dołu okna *Projects* pod nazwą "ARM Assembler v5.06 for μVision (PDF)".

Linker. Opcje linkera. Należy włączyć *Use Memory Layout from Target Dialog*.



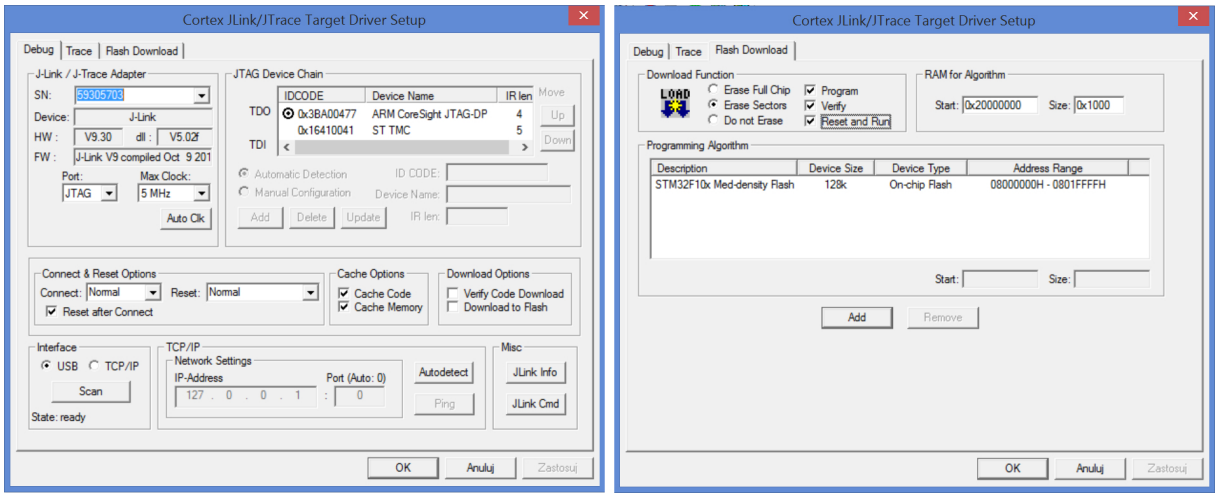
Rys. 4.17: Zakładka *Debug*

Debug. Zakładka ta zawiera opcje Debugera. Zaczniemy od uruchomienia programatora. Po prawej stronie klikniemy kółko *Use* i z listy programatorów wybierzemy **J-LINK / J-TRACE Cortex** - rysunek 4.17. Następnie klikamy przycisk *Settings*.
IDE odczytuje z programatora jego dane łącznie z numerem fabrycznym. Kolejna zakładka tego okna ***Trace***: nic nie zmieniamy, w szczególności nie włączamy opcji *Trace*, gdyż programator w wersji BASIC nie daje możliwości śledzenia programu. Zakładka ***Flash Download***: włączamy opcję *Reset and Run*. Oznacza ona, że po załadowaniu programu do Flasha, IDE wystawi i zdejmie sygnał resetu, dzięki czemu procesor rozpocznie wykonywanie programu. W oknie *Programming Algorithm* powinny być wypisane opcje pokazane na rysunku. Jeżeli są inne, to znaczy że w zakładce *Device* jest wybrany niewłaściwy procesor. Należy to tam poprawić. Naciskamy *OK* i wracamy do okna opcji projektu.

Utilities. Opcje programatora. Należy pozostawić wybraną opcję *Use Target Driver for Flash Programming*. Pozostałe opcje także bez zmian. Klawisz *OK* kończy wybór opcji.

4.8.2. Symulacja komputerowa

Prześledzimy wykonanie funkcji, która przepisuje string do miejsca przeznaczenia i zwraca ilość przepisanych znaków - patrz tabela 3.1. W oknie *Project* klikamy plik *main.c* i uzupełniamy go do postaci:

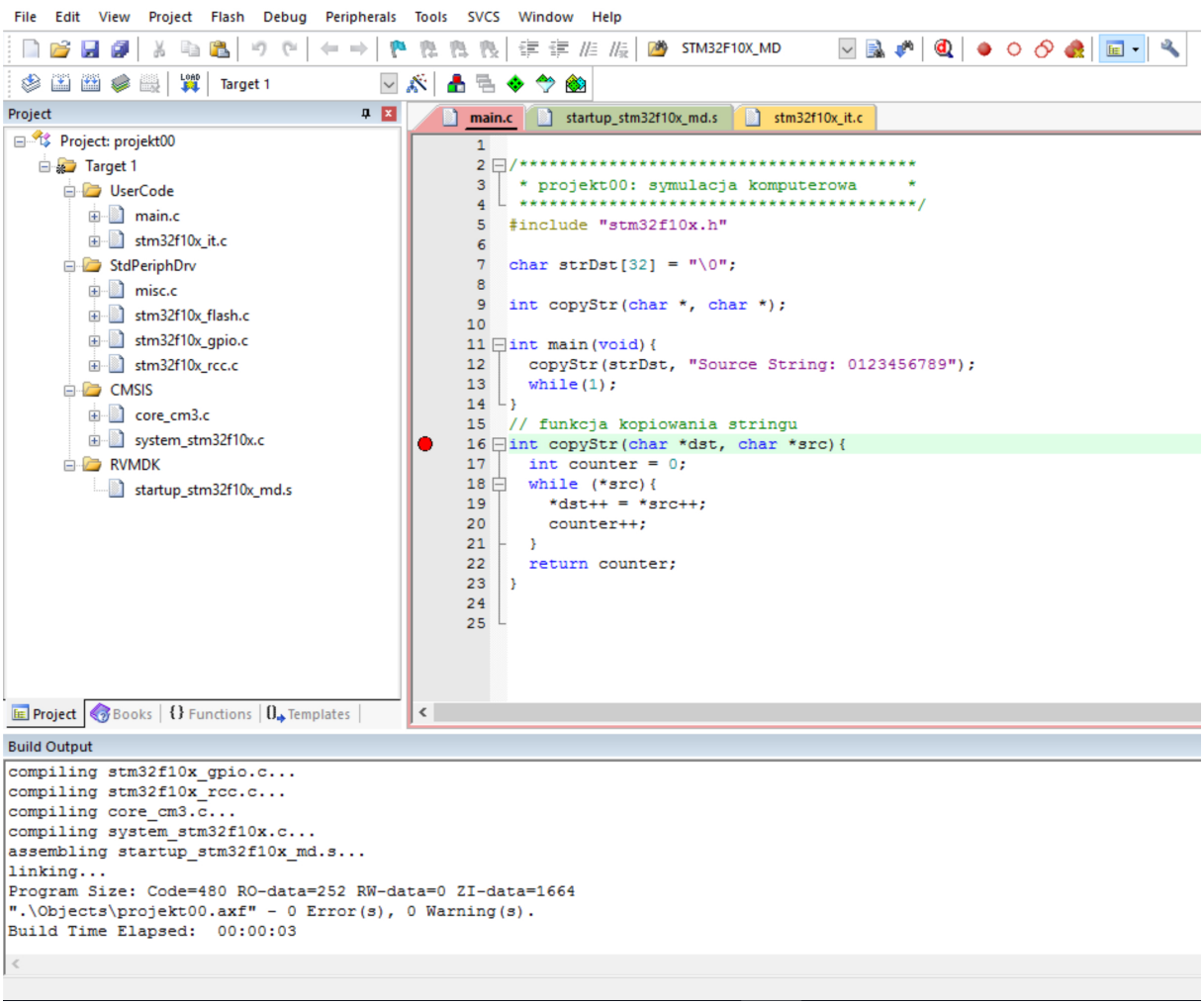


Rys. 4.18: Opcje programatora w zakładce *Debug*

```
    /* projekt00: symulacja komputerowa */
#include "stm32f10x.h"

char strDst[32] = "\0";

int copyStr(char*, char*);

int main(void) {
   copyStr(strDst, "Source String: 0123456789");
   while(1);
   return 0;     // warning: #111-D: statement is unreachable
}
// funkcja kopiowania stringu
int copyStr(char *dst, char *src){
   int counter = 0;
   while(*src){
     *dst++ = *src++;
     counter++;
   }
   return counter;
}
```

Formalnie, funkcja *main* powinna być zakończona instrukcją `return 0`. Jednak instrukcja ta nigdy nie będzie wykonana, co zostanie zasygnalizowane jako ostrzeżenie. W przyszłości będziemy pomijać takie instrukcje, ze względu na "czystość kompilacji".

Wykonujemy kompilację całego programu naciskając ikonę . Klikamy w szarym polu, obok linii 13. Pojawia się czerwona kropka - jest to pułapka programowa *Breakpoint* - rysunek 4.19. Teraz możemy przejść do symulacji. Klikamy ikonę lub w menu *Debug* wybieramy polecenie *Start/stop debug session Ctrl+F5*. Widok okna IDE zmieni się, jak na rysunku 4.20. W trybie debugowania możliwe jest śledzenie wykonania programu krok po kroku oraz podgląd wartości wszystkich rejestrów procesora, wszystkich rejestrów związanych z układami peryferyjnymi oraz dowolnego obszaru pamięci.

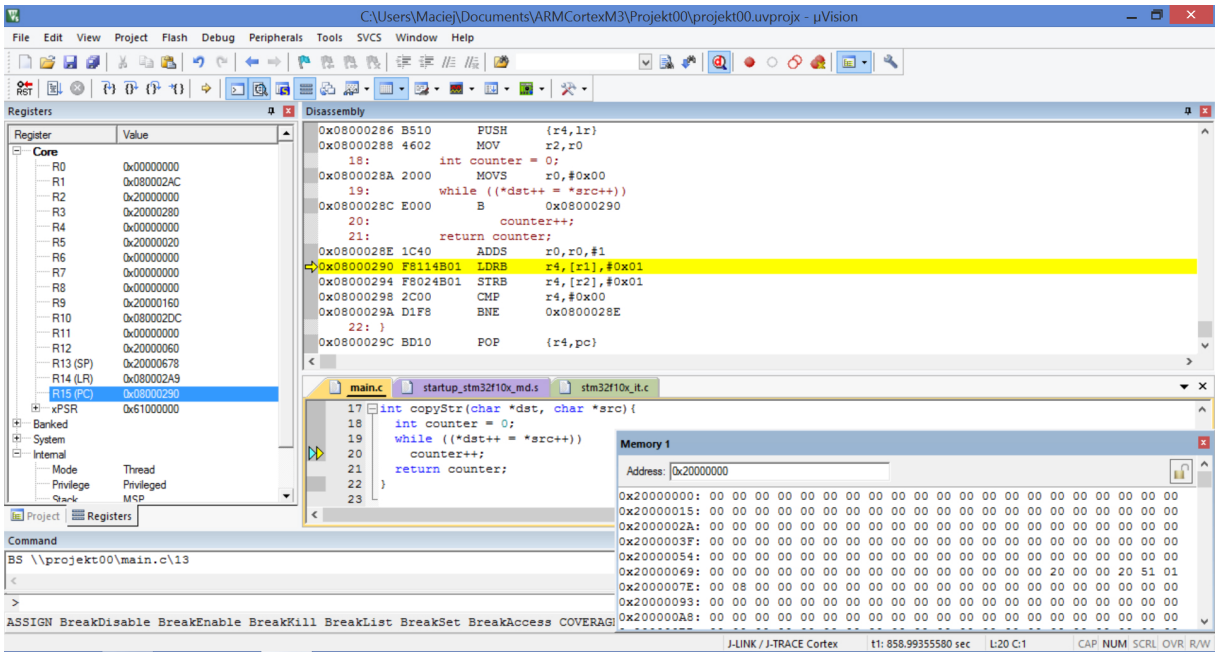


Rys. 4.19: Plik *main.c* w oknie edytora

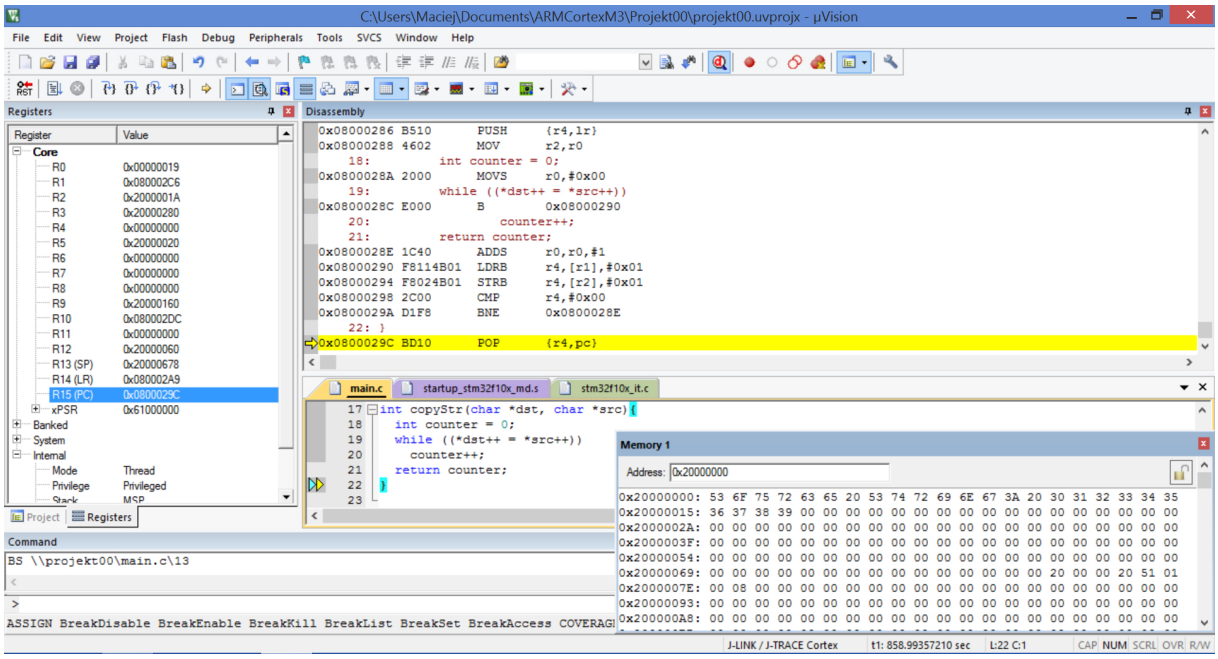
Po lewej stronie okna widoczna jest zakładka *Register*. Można tam na bieżąco oglądać zawartość rejestrów *R0...R15* oraz rejestr flag *xPSR*. Nad oknem kodu źródłowego widoczne jest okno *Disassembly* w którym określony obszar pamięci zawierającej kod maszynowy jest przedstawiony w postaci asemblerowej. Jest to uzyskane w wyniku deasemblacji kodu maszynowego, dlatego nie ma tam oryginalnych nazw etykiet jakie stosowaliśmy np. w asemblerze. Adresy w polu operandu instrukcji skoków należy traktować jak etykiety. Kod asemlerowy przemieszany jest z oryginalnym kodem źródłowym wyróżnionym innym kolorem. Instrukcja, która będzie wykonana w kolejnym kroku jest oznaczona żółtą strzałką.

Do sterowania przebiegiem symulacji służą ikony zgromadzone z lewej stony. Ikona RST to reset procesora: symulacja wraca na początek, ikona to: wykonaj program do najbliższej pułapki *Breakpoint* i na niej się zatrzymaj. Jeżeli nie wstawiliśmy pułapki, to program będzie wykonywany w sposób ciągły. Ikona to zatrzymaj wykonywanie programu.

Najechanie kursorem na ikonę wyświetla jej funkcję, na przykład ikona opisana jest jako *Step (F11), step one line*. Ikona *Step over (F10)*, to znaczy wykonaj kolejną instrukcję bez wchodzenia do wnętrza funkcji.



Rys. 4.20: Debuger, początek symulacji



Rys. 4.21: Debuger, koniec symulacji

Przed rozpoczęciem symulacji otworzymy podgląd obszaru pamięci do którego przepisywany będzie string. Można to wykonać klikając na ikonę *Memory Windows* w tym samym szeregu ikon, albo wybierając z menu *View* opcję *Memory Windows*. Możemy otworzyć cztery okna podglądu pamięci, wybieramy na przykład *Memory1*. W polu *Address* wpisujemy `0x20000000`. W naszym programie nie określiliśmy żadnych zmiennych globalnych oprócz tablicy znaków `strDst`, dlatego obszar docelowy rozpoczyna się od początku pamięci S-RAM. Początek symulacji z otwartym oknem podglądu pamięci pokazany jest na rysunku 4.20. Naciskając klawisz F11 lub klikając ikonę wykonujemy

program krok po kroku. Możemy porównać wykonywanie instrukcji z listingiem asemblera pokazanym w tablicy 3.1. W kolejnych krokach program przepisuje znaki aż do końca stringu. Wtedy wraca w miejsce wywołania z ilością przepisanych znaków w rejestrze *r0*. Następnie wykonuje jedną instrukcję NOP i wpada w pętlę nieskończoną `while(1)`. Widok ekranu po przepisaniu całego stringu pokazany jest na rysunku 4.21.

Klikając powtórnie ikonę debuggera, kończymy symulację i wracamy do edytora - rysunek 4.19. Kliknięcie na czerwoną kropkę usuwa pułapkę programową. Ikona *LOAD* ładuje program do pamięci Flash i startuje procesor.

Środowisko IDE Keil μVision oferuje ogromne możliwości debugowania. Jest nawet analizator stanów logicznych (cyfrowy oscyloskop) oraz rejestracja znaków wysyłanych /odbieranych w transmisji szeregowej. Procesory M4 i M7 są wyposażone w jeszcze mocniejsze narzędzie, jest to ETM *Embedded Trace Macrocell (trace debugging)*.

4.9. Reset and Clock Control (RCC)

4.9.1. Reset

Naciskając przycisk RESET w zestawie uruchomieniowym (**4** na rysunku 4.1) wymuszamy stan niski na nóżce NRST, co wprowadza procesor w reset. Wszystkie wewnętrzne rejestry zostają ustawione w swój stan początkowy. Po zwolnieniu przycisku, procesor rozpoczyna procedurę *boot*owania, czyli przygotowania pamięci do pracy. Sposób *boot*owania zależy od wartości logicznych na pinach BOOT0 i BOOT1, jak pokazano w tabeli 4.5. Mapa pamięci jest na rysunku 3.5.

Tabela 4.5: Tryby *boot*owania

BOOT1	**BOOT0**	**Tryb *boot*owania**	**Powielenie pamięci (*aliasing*)**
x	0	Main Flash Memory	Procesor startuje z wbudowanego Flasha
0	1	System memory	Wbudowany *boot loader*
1	1	Embedded S-RAM	Procesor startuje z wbudowanego S-RAM

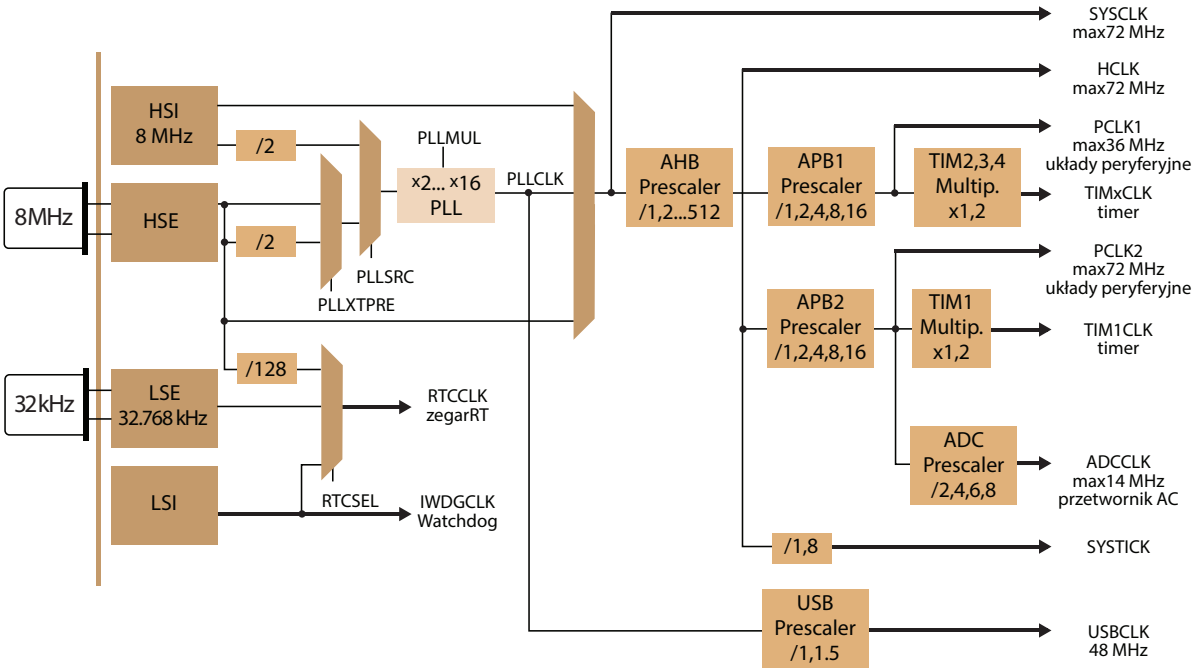
Procedura startowa zależy od wybranego trybu

- *Boot*owanie z wbudowanego Flasha: Flash jest powielany do adresu zerowego, to znaczy jest widziany przez procesor pod adresem `0x00000000` i adresem `0x08000000`.
- Pamięć systemowa *System memory* jest powielana do adresu zerowego. Możliwe jest załadowanie programu przez USART1. Szczegóły w dokumencie RM0008.
- *Boot*owanie z wbudowanego statycznego RAMu. S-RAM jest dostępny tylko od adresu (0x20000000) - nie jest powielany.

W naszych projektach będziemy startować z wewnętrznego Flasha, dlatego obie zworki BOOT (**19** na rysunku 4.1) powinny być w pozycji zero.

4.9.2. Taktowanie mikrokontrolera - sygnały zegarowe

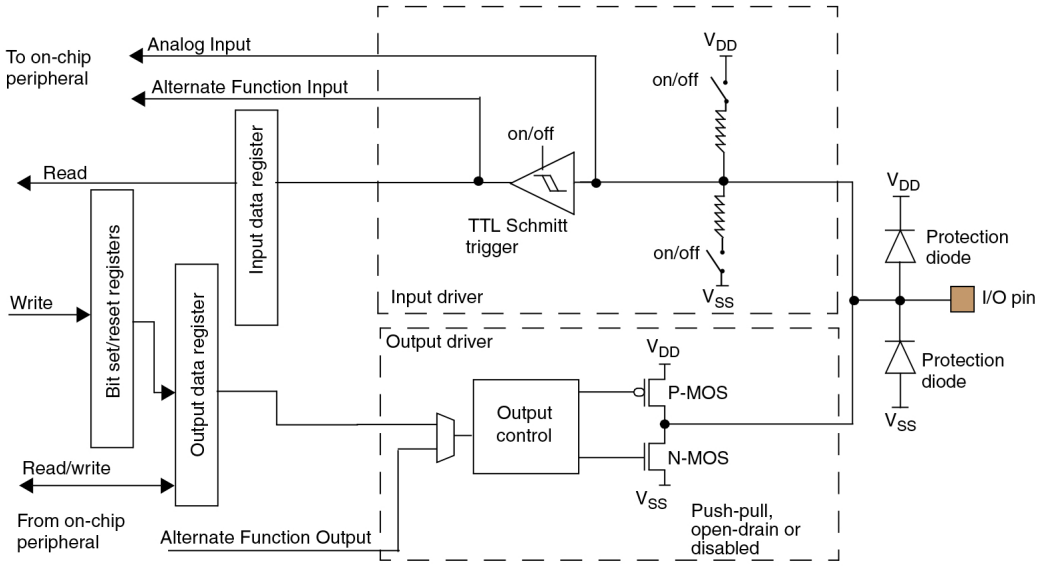
Procesor i peryferiale są zbudowane z bramek logicznych (Rys. 2.5). W rozbudowanych układach logicznych występują wyścigi i hazardy spowodowane różnicami w opóźnieniach wprowadzanych przez struktury logiczne (czas od wymuszenia wejścia do pojawienia się finalnego stanu na wyjściu). Przykładem hazardu (niewiadomy stan końcowy) spowodowanego wyścigiem (nie wiemy który sygnał będzie pierwszy) jest przerzutnik RS (Rys.



Rys. 4.22: Układ taktowania procesora i układów peryferyjnych

2.6). W przypadku sekwencji sygnałów niskich $\bar{R}$ i $\bar{S}$, stan przerzutnika Q zależy od ich kolejności (decyduje ostatni). Mikroprocesor zawiera kilkadziesiąt tysięcy bramek. Rozwiązaniem problemu jest wprowadzenie synchronicznego trybu pracy. Specjalny sygnał taktujący reguluje pracę całego systemu. Układy logiczne rozpoczynają pracę w momencie wystąpienia taktu zegara (synchronicznie). W czasie do następnego taktu zostaną wykonane wszystkie operacje logiczne i na wyjściach→wejściach bramek ustalą się właściwe poziomy sygnałów. Kolejny takt to kolejne ustalanie wartości sygnałów. Maksymalna częstotliwość zegara taktującego jest ograniczona największym opóźnieniem w układzie. Ten system ma jedną wadę: wszystkie układy wewnętrzne mikrokontrolera przełączane są w tym samym czasie i wtedy pobierany jest bardzo duży prąd na przeładowanie pojemności wewnętrznych zgodnie z opisem technologii CMOS na stronie 11. Powoduje to impulsowy pobór energii, co może prowadzić do emisji zakłóceń elektromagnetycznych. Dlatego blisko nóżek zasilających należy ustawić kondensatory blokujące (blokują zakłócenia) z których procesor pobierze energię impulsowo. Stosuje się dwa kondensatory połaczone równolegle: tantalowy o dużej pojemności i ceramiczny o szerokim paśmie (opowiada na sygnały o bardzo krótkim czasie narastania). Jest też zaleta: w czasie pomiędzy taktami, mikrokontroler praktycznie nie pobiera energii. Łącznie uzyskujemy układ o niskim poborze energii, który może być stosowany w urządzeniach przenośnych, zasilanych bateryjnie, jak telefon komórkowy. Na rysunku 2.6 przerzutnik RS jest narysowany w wersji asynchronicznej. W celu zastosowania w mikrokontrolerze, należy go uzupełnić o układ synchronizcji z wejściowym sygnałem taktującym.

Wbudowany moduł RCC *Reset and Clock Control* (Rys. 4.22) dostarcza sygnały zegarowe do wszystkich układów. Są dwa generatory wewnętrzne RC (opornik + kondensator) 8 MHz HSI *High-speed Internal* oraz 40 kHz LSI *Low-speed internal*. Po resecie te dwa sygnały taktują cały system. Jednak generatory RC są mało stabilne i lepiej jest korzystać z rezonatorów kwarcowych. Jeden o częstotliwości 8 MHz jest dołączony do wewnętrznego generatora HSE *High-speed external*, a drugi zegarkowy 32.768 kHz do LSE *Low-speed*



Rys. 4.23: Schemat wewnętrzny pinu GPIO (z dokumentu RM0008)

external. Ponadto układ RCC jest wyposażony w programowane dzielniki częstotliwości i pętlę PLL *Phase Locked Loop*. Pętla PLL to "mnożnik częstotliwości" ze sprzężeniem zwrotnym: przestrajany generator synchronizowany z wejściowym sygnałem taktującym. Pozwala na pomnożenie częstotliwości przez liczbę całkowitą. My wykorzystamy PLL do podniesienia częstotliwości taktowania do 72 MHz, mnożąc częstotliwość kwarcu 8 MHz przez 9. Dokładny opis układu RCC jest w dokumencie RM0008. W kolejnym projekcie "Projekt01_LED" zaprogramujemy RCC na maksymalne możliwe częstotliwości (korzystając z CMSIS).

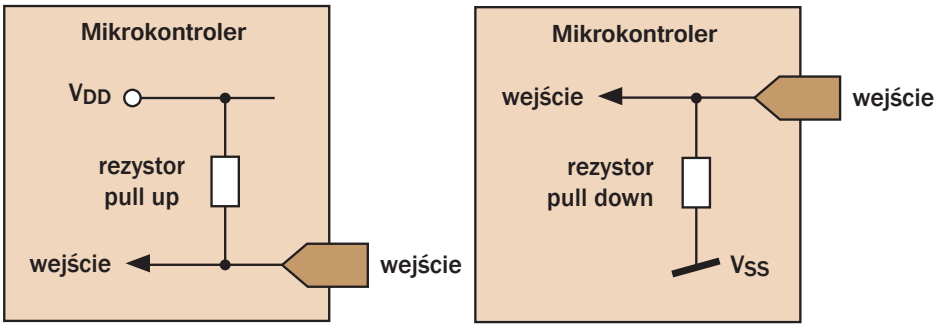
4.10. Porty GPIO

General Purpose I/O, czyli porty we/wy ogólnego przeznaczenia. Zawsze jest ograniczona ilość wyprowadzeń, czyli pinów procesora, dlatego producenci starają się każdy pin wykorzystać jak najlepiej. Piny, które mogą mieć różne funkcje i te funkcje mogą być konfigurowane także w czasie wykonywania programu, nazywane są portami GPIO. Uzyskujemy dużą elastyczność używania i ogromną wygodę w projektowaniu. Umożliwia to stosowanie mikrokontrolera w wielu różnych aplikacjach, ale opłacone jest koniecznością złożonej inicjalizacji. Piny portu GPIO możemy skonfigurować jako:

- Wejście cyfrowe z przerzutnikiem Schmitta (z histerezą)
- Wyjście cyfrowe *push-pull* albo z otwartym drenem
- Wejście /wyjście analogowe
- Inna specjalizowana funkcja jak wyjście PWM, sterowanie LCD, zewnętrzne przerwanie, sygnał USART, SPI, I^2C, USB

Ostatnią kategorię funkcji, jaką może spełniać pin portu GPIO nazywamy funkcją alternatywną (*Alternate Function*) AF. Program może dynamicznie zmieniać funkcję pinu GPIO. W tym rozdziale skupimy się na wejściach /wyjściach cyfrowych.

Linie portów GPIO są pogrupowane po 16. Procesor STM32F103 w obudowie LQFP100 ma pięć 16-bitowych portów, które są oznaczone *PortA, PortB, PortC, PortD* i *PortE*. W każdym porcie piny są oznaczone $PA0, PA1, \ldots, PA15$ aż do $PE0, PE1, \ldots, PE15$.



Rys. 4.24: Wejście z opornikiem podciągającym i ściągającym

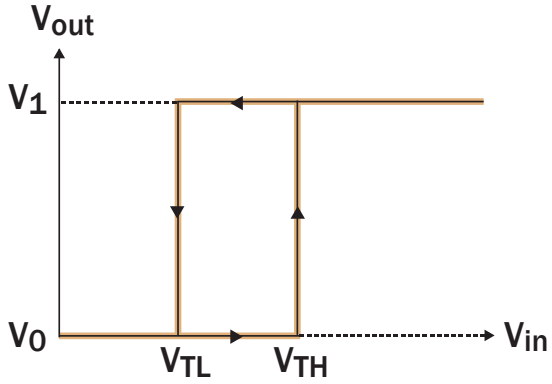
Łącznie daje to 80 pinów GPIO. Każdy *PortX* ma swój zestaw rejestrów sterujących, które umożliwiają konfigurowanie pinów portu indywidualnie oraz jednoczesny odczyt wszystkich bitów i jednoczesny zapis do wszystkich bitów portu. Istnieją także rejestry które umożliwiają ustawianie lub zerowanie pojedynczych bitów portu. Jeżeli procesor ma wbudowanych pięć portów, to ma pięć zestawów rejestrów (o jednakowej strukturze). W rozdziale (...) będzie opisane jak odczytywać, zerować, ustawiać i odwracać pojedyncze bity portów w asemblerze i w C.

4.10.1. Wejście: oporniki *Pull Up* i *Pull Down*

Pin można skonfigurować na stan wysoki, niski lub jako wejście wysokiej impedancji. Gdy wejście jest zaprogramowane jako wysokiej impedancji, to mówimy że jest w trzecim stanie lub że wisi w powietrzu. Każde wejście można tak skonfigurować, że zostanie ono połączone z zasilaniem *pull up* lub z masą *pull down* za pośrednictwem wewnętrznego opornika (rysunek 4.24). Wymusza to określony stan logiczny na wejściu, gdy nie jest do niego podłączony żaden układ zewnętrzny. Program może dynamicznie zmieniać oporniki z *pull up* na *pull down* i odwrotnie, w czasie swego działania. Jeżeli np. dołączony jest opornik podciągający, a na wejściu jest stan niski, przez wewnętrzny opornik płynie prąd. Aby ograniczyć energię wydzielaną wewnątrz układu scalonego mikrokontrolera, oporniki *pull up* i *pull down* mają stosunkowo dużą oporność, typowa wartość to 40 kΩ.

Jeżeli układ dołączony do wejścia ma dużą pojemność, wewnętrzne oporniki mogą spowolnić jego działanie, gdyż ograniczają prąd przeładowania tej pojemności. Takie wejście lepiej jest skonfigurować na stan trzeci. W szczególności, jeżeli określone wejście ma być użyte do bardzo szybkiej transmisji szeregowej jak I^2C, wewnętrzne oporniki muszą być odłączone. Z drugiej strony nie wolno zostawiać niepodłączonych wejść w stanie wysokiej impedancji, gdyż mogą gromadzić się na nich ładunki elektrostatyczne i w efekcie tego wejście może ulec trwałemu uszkodzeniu. Producenci zalecają, aby nieużywane linie portów GPIO zaprogramować jako wejścia z opornikiem *pull up* , albo *pull down*. Historycznie w technice mikroprocesorowej sygnały sterujące były aktywne niskie, dlatego stosuje się raczej opornik *pull up* - wtedy wejście jest nieaktywne.

Nie wolno podłączać nieużywanych pinów bezpośrednio do masy lub zasilania. Omyłkowe skonfigurowanie na wyjście komplementarne może doprowadzić do zwarcia i uszkodzenia wyjścia.



Rys. 4.25: Histereza

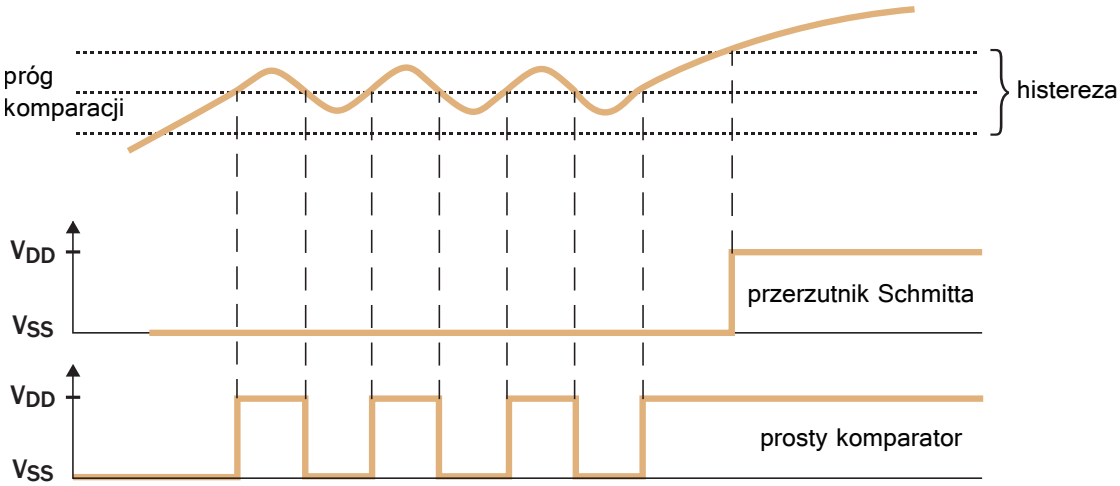
4.10.2. Wejście, przerzutnik Schmitta

W technice cyfrowej wszystkie sygnały muszą spełniać wymagania danej rodziny bramek logicznych. Są to wymagania na poziom niski i poziom wysoki sygnału. Stan pośredni jest zabroniony i może prowadzić do oscylacji. Ponadto są ściśle określone wymagania na czasy narastania i opadania zboczy, czyli maksymalny czas przejścia pomiędzy stanami logicznymi. Drugie wymaganie wynika z pierwszego. Dlatego w układach wejściowych (przyjmujących sygnały z zewnątrz) zawsze stosuje się bramki Schmitta. Na schematach (rysunek 4.23) jest to bramka logiczna z wrysowanym symbolem histerezy.

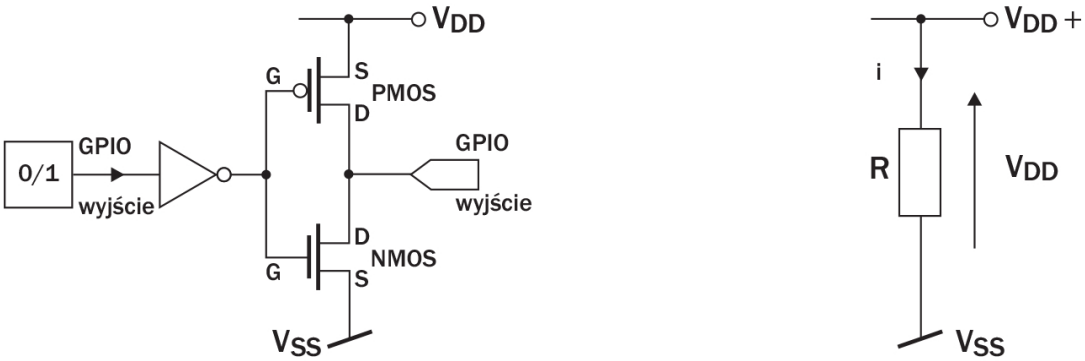
Działanie bramki Schmitta wyjaśnione jest na rysunku 4.25. Oś pozioma to napięcie wejściowe, oś pionowa to wyjściowe. Jeżeli napięcie wejściowe rośnie wolno od zera w kierunku V_{DD}, to stan wyjścia zostanie zmieniony na wysoki dopiero po przekroczeniu progu V_{TH} *Trigger High* i pozostanie wysoki aż do osiągnięcia V_{DD}. Gdy napięcie wejściowe maleje od V_{DD} do zera, to stan wyjścia zostanie zmieniony na niski dopiero gdy napięcie wejściowe osiągnie wartość mniejszą od V_{TL} *Trigger Low*. Zakres napięć wejściowych pomiędzy progami przełączania V_{TL} a V_{TH} jest nazywany histerezą. W zakresie histerezy stan wyjścia zależy od historii, czyli "skąd idziemy". Histereza powoduje, że sygnał o długich czasach narastania /opadania może zostać zamieniony na sygnał spełniający wymagania specyfikacji bramek logicznych. Ponadto, jeżeli sygnał wejściowy jest obciążony szumem o amplitudzie mniejszej niż połowa histerezy, to ten szum zostanie odcięty i nie przedostanie się dalej do systemu mikroprocesorowego. Ilustruje to rysunek 4.26. Jest na nim porównanie działania prostego komparatora bez histerezy, który też "wyostrza zbocza", z działaniem bramki Schmitta. Widać korzystny wpływ histerezy.

4.10.3. Wyjście GPIO: komplementarne lub otwarty dren

Programowo można skonfigurować wyjście GPIO jako komplementarne "popchnij - pociągnij" *Push-Pull* albo jako wyjście z otwartym drenem *Open-Drain*. W trybie *Push-Pull* z wyjścia może wypływać prąd lub do niego wpływać (możliwy jest przepływ w obu kierunkach), w trybie otwartego drenu prąd może tylko wpływać do wyjścia. Pamiętajmy że zgodnie z zasadami strzałkowania prądów i napięć (prawa strona rysunku 4.27) prąd płynie od napięcia zasilania V_{DD} o potencjale dodatnim do masy V_{SS} o potencjale ujemnym, czyli spływa z góry na dół. Elektrony zaś pracowicie wspinają się pod górę: od minusa (który



Rys. 4.26: Porównanie działania komparatora i wejścia z histerezą

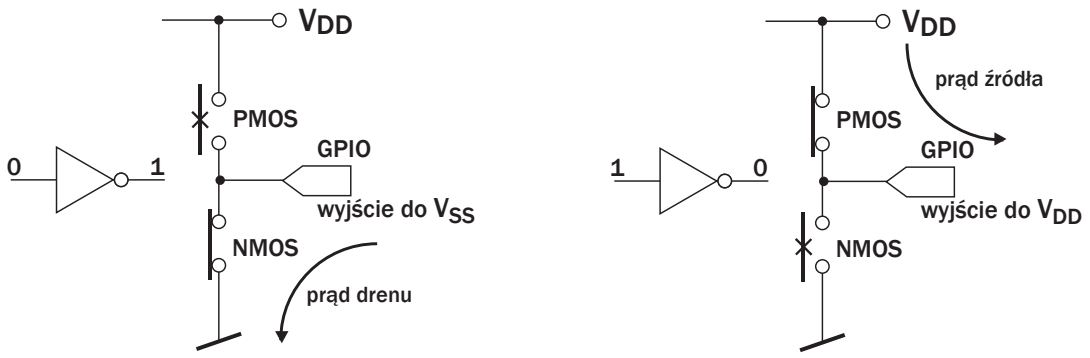


Rys. 4.27: Wyjście GPIO *Push-Pull* i strzałkowanie

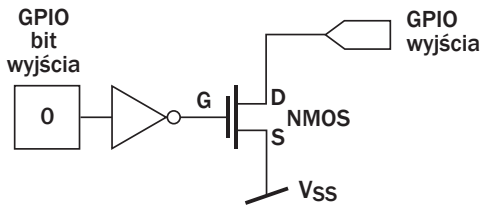
je odpycha) do plusa (który przyciąga). Ponieważ ładunek elektronu ma znak ujemny (umownie), więc kierunek prądu jest przeciwny do kierunku ruchu elektronów. Napięcie strzałkujemy od minusa do plusa (w kierunku wyższych potencjałów), czyli odwrotnie do prądu. Strzałkowanie pokazane jest z prawej strony rysunku 4.27.

Wyjście GPIO *Push-Pull*. Z lewej strony rysunku 4.27 pokazany jest schemat wyjścia GPIO w trybie *Push-Pull*. Są tam połączone dwa komplementarne (odwrotnie domieszkowane) tranzystory MOSFET (układ komplementarny oznaczamy CMOS). W danej chwili tylko jeden z nich jest włączony. Gdy do rejestru sterującego tego portu jest wpisany bit o wartości zerowej, tranzystor PMOS jest wyłączony, NMOS włączony, wyjście ma stan niski i prąd wpływa z układu sterowanego do wyjścia mikrokontrolera, rysunek 4.28. Gdy bit sterujący ma wartość logiczną jeden, PMOS jest włączony, NMOS wyłączony, wyjście ma stan wysoki i prąd z wyjścia płynie do układu sterowanego.

Zauważmy, że na rysunku 4.27 tranzystory PMOS i NMOS są połaczone "odwrotnie", to znaczy PMOS ma źródło S *Source* u góry, a dren D u dołu, a tranzystor NMOS ma źródło dołączone do masy. Jest w tym symetria: źródła do zasilań, a oba dreny do wyjścia. Wynika to z faktu, że w technologii MOSFET potencjał bramki G *Gate* odnosimy do potencjału źródła. Tranzystor sterujemy różnicą tych potencjałów, czyli napięciem bramka - źródło. W obu tranzystorach potencjały źródeł są ustalone, więc sygnał sterujący



Rys. 4.28: Wyjście GPIO *Push-Pull*, stany wysoki /niski



Rys. 4.29: Schemat wyjścia GPIO *Open-Drain*

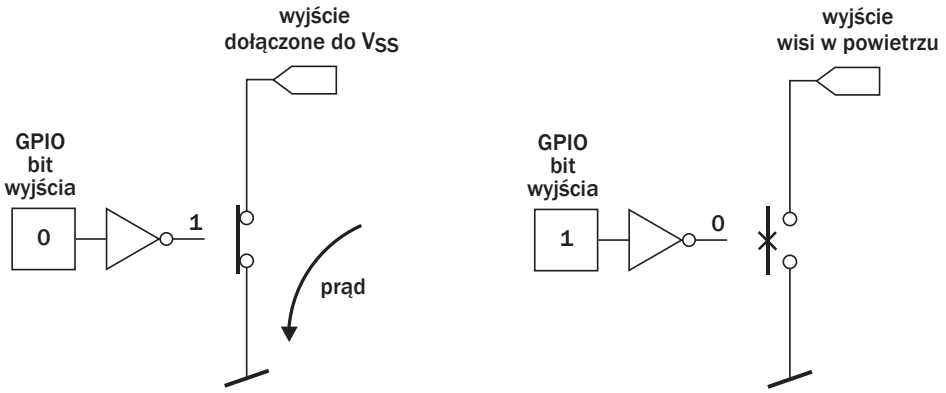
(wyjście negatora) odnosimy do napięć zasilających. Natomiast potencjał obu drenów zależy od układu dołączonego do wyjścia.

Wyjście GPIO *Open-Drain*. Schemat wyjścia w trybie otwartego drenu pokazany jest na rysunku 4.29. Gdy bit sterujący stanem wyjścia jest zerowy, załączony jest tranzystor NMOS i wyjście przyjmuje prąd z układu sterowanego. W przeciwnym przypadku (bit = 1), tranzystor jest wyłączony i prąd nie płynie. Główną zaletą wyjść z otwartym drenem jest to że można je łączyć (zwierać wyjścia). To oznacza, że kilka wyjść może sterować jednym układem zewnętrznym. Realizowana jest wtedy operacja logiczna AND (dla sygnałów "aktywny wysoki"), to znaczy że jeżeli wszystkie wyjścia są w stanie wysokim, to ich połączenie też jest w stanie wysokim (potrzebny jest jeden, wspólny dla wszystkich wyjść opornik podciągający do V_{DD}). Jeżeli którekolwiek wyjście jest w stanie niskim, to ich połączenie też jest w stanie niskim. Jeżeli stosujemy logikę "aktywny niski", to połączenie otwartych drenów realizuje działanie logiczne OR: połączenie jest aktywne, gdy choć jeden jest aktywny = niski.

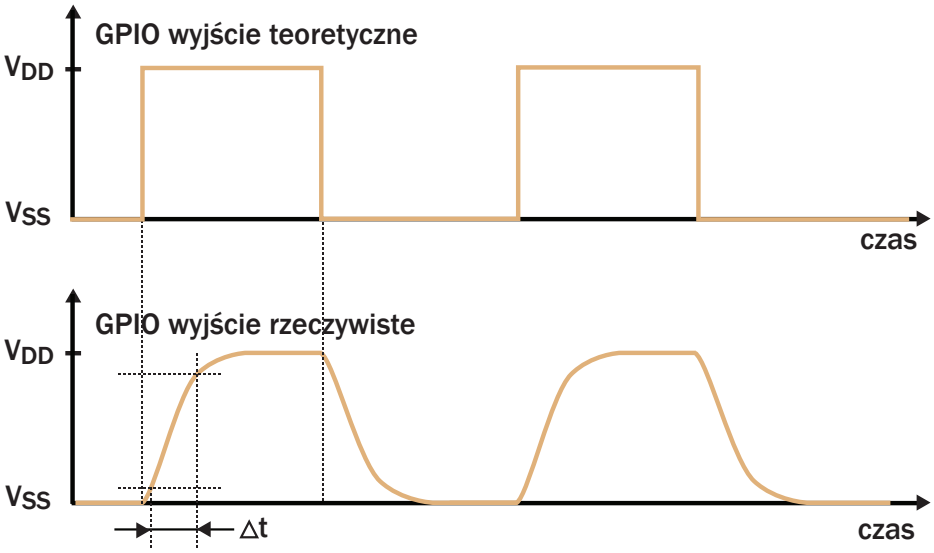
Na przykład standard komunikacji I^2C stosuje tą technikę do łączenia wielu układów nadajnik+odbiornik do tej samej lini transmisyjnej (rysunek 18.13). Tylko jeden nadajnik może nadawać (wymuszać stany niskie) w danej chwili, wszystkie pozostałe są wyłączone. Po zakończeniu transmisji linia może zostać przydzielona do innego nadajnika.

Wyjścia z otwartym drenem mogą sterować układy zasilane z wyższego napięcia niż V_{DD} - pokażemy to w podrozdziale o dołączaniu diod świecących LED.

Wyjść *Push-Pull* nie można łączyć ze sobą (zwierać), ale mają one tą zaletę, że są szybsze gdyż ładują i rozładowują pojemność dołączoną do wyjścia, silnymi prądami bezpośrednio z V_{DD} i do V_{SS}.



Rys. 4.30: Wyjście GPIO *Open-Drain*, stany załączony /wyłączony



Rys. 4.31: Nachylenie zbocza narastającego

4.10.4. Szybkość wyjścia GPIO

Nachylenie zbocza narastającego jest parametrem opisującym szybkość narastania sygnału w trakcie zmiany stanu 0→1. Jeżeli stan logiczny wyjścia GPIO zmienia się z 0 na 1 i towarzysząca temu zmiana napięcia od 0V do 3V nastąpiła w czasie 3μs, to szybkość narastania *slew rate* jest równa 1[V/μs]. Analogicznie definiuje się nachylenie zbocza opadającego. Rysunek 4.31 pokazuje sposób określania nachylenia zbocza.

Im większe jest nachylenie zbocza, tym jest mniejszy czas zmiany stanu logicznego. To oznacza, że z większą częstotliwością procesor może odwracać stan bitu. Na rysunku 4.31 pokazane jest porównanie stanu idealnego z rzeczywistym, gdy procesor okresowo odwraca stan bitu.

Jednak większa częstotliwość przełączania oznacza większą emisję zakłóceń elektromagnetycznych EMI *electromagnetic interference*. Sygnał o ostrych zboczach, czyli krótkich czasach przełączania ma szersze pasmo. To znaczy widmo mocy obejmuje szerszy zakres częstotliwości, a to oznacza większą emisję EMI. Widmo mocy sygnału, to ten sam sygnał przedstawiony w dziedzinie częstotliwości, czyli przekształcony za pomocą transformaty Fouriera DFT *Digital Fourier Transform*, FFT jest tylko algorytmem szybkiego liczenia

DFT - wynik FFT nie różni się od wyniku DFT. Mikrokontrolery Cortex-M pozwalają na zaprogramowanie maksymalnej częstotliwości przełączania portów, co oznacza wybór jednej z trzech wartości nachylenia zbocza

Tabela 4.6: I/O AC characteristics

MODEx[1:0], wartości bitów	**Maksymalna częstotliwość przełączania**	**Pojemność dołączona do wyjścia**	**Czas narastania zbocza / czas opadania zbocza**
1 0	2 MHz	50 pF	125ns / 125 ns
0 1	10 MHz	50 pF	25ns / 25 ns
1 1	50 MHz	30 pF	5ns / 5 ns

Jeżeli nie musimy szybko przełączać urządzenia dołączonego do wyjścia, warto jest obniżyć maksymalną częstotliwość przełączania. Dzięki temu zmiejszymy emisję zakłóceń. Szczegółowe dane są zawarte w dokumencie **Datasheet** (Tabela 4.2).

4.10.5. Jak dołączyć diodę LED do mikrokontrolera ?

Obciążalność prądowa każdej linii portu GPIO to ± 8 mA. Jednocześnie całkowity prąd jaki możemy "przepuścić" przez mikrokontroler STM32F103 nie może przekroczyć 150 mA. Jeżeli do każdej linii GPIO dołączymy jedną diodę LED, czyli razem 80, to średnio na diodę mamy mniej niż 2 mA. Dlatego lepiej jest zastosować jakiś zewnętrzny układ, który pozwoli nam wysterować diodę prądem na przykład 20 mA, nie obciążając mikrokontrolera. Najprostsze rozwiązanie to tranzystor bipolarny *n-p-n* w układzie otwartego kolektora. Jest to analogia do układu z otwartym drenem w tranzystorach unipolarnych.

Tabela 4.7: Tranzystory krzemowe n-p-n

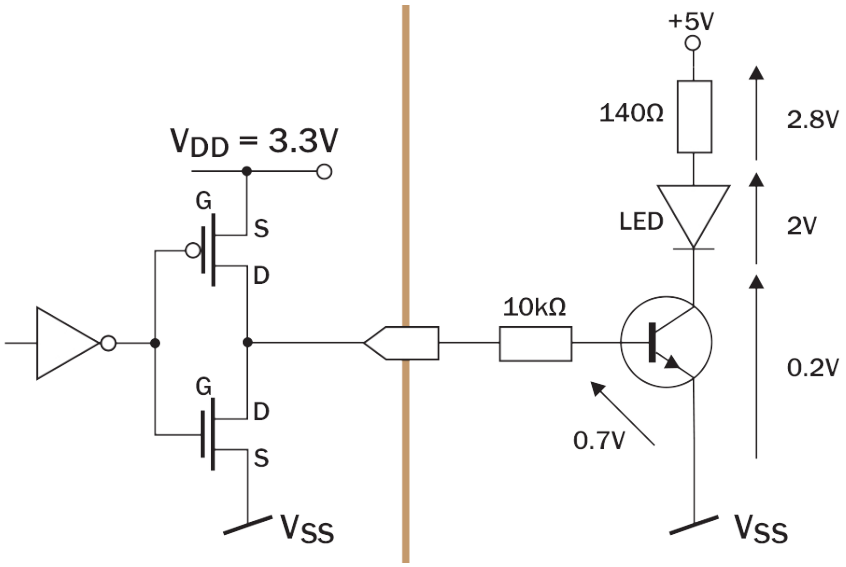
tranzystor	$V_{CE\ max}$	$I_{C\ max}$	P_D	$V_{BE\ sat}$	$V_{CE\ sat}$	$\beta = I_C/I_B$	obudowa
2N5551	160 V	600 mA	625 mW	1.0 V	0.2 V	80 - 250	TO-92
BC107	45 V	100 mA	300 mW	0.9 V	0.2 V	200 - 450	TO-18
BC337	50 V	800 mA	625 mW	0.8 V	0.7 V	100 - 250	TO-92
BC548B	30 V	100 mA	500 mW	0.7 V	0.2 V	200 - 450	TO-92

Podstawowe dane kilku popularnych tranzystorów *n-p-n* podane są w tabeli 4.7. Kolejne kolumny to: V_{CEmax} dopuszczalne napięcie kolektor - emiter, I_{Cmax} dopuszczalny prąd kolektora, P_D *Power Dissipation* dopuszczalna moc wydzielana w tranzystorze, V_{BEsat} napięcie baza-emiter w stanie nasycenia, V_{CEsat} napięcie kolektor - emiter w stanie nasycenia, β wzmocnienie prądowe.

Układ musimy zaprojektować tak, aby nie przekroczyć wartości dopuszczalnych. Zastosujemy diodę LED *High Efficiency Red*, którą wysterujemy prądem $I_{LED} = 20mA$ z napięcia $V_{CC} = 5V$. Z tabeli 2.1 odczytujemy napięcie przewodzenia tej diody: $V_{LED} =$ 2V. Wybieramy tranzystor BC548B. Dobierzemy opornik ograniczający prąd diody

$$R_C = \frac{V_{CC} - V_{LED} - V_{CEsat}}{I_{LED}} = \frac{5 - 2 - 0.2}{0.02} = \frac{2.8}{0.02} = 140\Omega \qquad \rightarrow \qquad 150\Omega/0.25W$$

Oporniki ogólnego stosowania są produkowane w trzech tolerancjach 1%, 5% i 10%. W każdej z tych tolerancji są wartości standardowe oporności, czyli te produkowane. Zastosujemy opornik o tolerancji 10%. Standardowe oporności w tej tolerancji to 10, 12, 15, 18,



Rys. 4.32: Schemat dołączenia diody LED do wyjścia GPIO

22, 27, 33, 39, 47, 56, 68, 82 oraz wszystkie pomnożone przez 10^n. Najbliższy w szeregu jest więc opornik 150Ω. Moc wydzielona w oporniku będzie równa

$$P_R = \frac{U^2}{R} = \frac{2.8^2}{150} = 52mW$$

zastosujemy więc opornik 0.25 W. Prąd kolektora: 2.8/150 = 18.7 mA. Moc w tranzystorze $I_C \cdot V_{CEsat} = 0.0187 \cdot 0.2 = 0.00374 < 0.5$ W. Zakładamy wzmocnienie prądowe tanzystora $\beta = 100$. Minimalny prąd bazy to $I_B = I_C/\beta$. Napięcie na wyjściu GPIO w stanie wysokim $V_{GPIO} = 3.3$V, czyli opornik szeregowy w bazie

$$R_B = \frac{U_{GPIO} - U_{BEsat}}{I_B} = \frac{(U_{GPIO} - U_{BEsat}) \cdot \beta}{I_C} = \frac{(3.3 - 0.7) \cdot 100}{0.0187} = 13.904k\Omega$$

Chcemy osiągnąć nasycenie, wstawimy więc opornik $10k\Omega$ /0.25W. Schemat dołączenia diody LED do wyjścia GPIO pokazany jest na rysunku 4.32.

4.11. Projekt01_LED, migająca dioda

4.11.1. Tworzymy nowy projekt

Zaczniemy od skopiowania katalogu *Projekt00* wraz z zawartością. Kopii nadamy nazwę *Projekt01_LED*. Otwieramy nowo utworzony katalog. Możemy usunąć wszystkie pliki z katalogu *Listings* i z *Objects*. We wszystkich plikach, których nazwy zaczynają się od *project00...* zmienimy je na zaczynające się od *projekt01....*

Otwieramy środowisko IDE Keil μVision. W menu *Project* wybieramy *Open Project....* Pojawi się okno, w którym otwieramy katalog *Project01_LED* i wybieramy plik *project01* → *Otwórz.*

Sprawdzimy opcje projektu: wszystko powinno być zachowane. W zakładce *Output*, w polu *Name of Executable* zmieniamy nazwę na *project01*. W zakładce *C/C++*, w polu *Include Paths* musimy usunąć ścieżkę do poprzedniego projektu *..\Projekt00* i wpisać do aktualnego *..\Projekt01_LED*. W ten sposób będziemy tworzyć następne projekty.

4.11.2. Plik *stm32f10x_conf.h*

Otwieramy plik *stm32f10x_conf.h*. Zawiera on listę plików nagłówkowych biblioteki obsługi urządzeń peryferyjnych. Musimy usunąć znak komentarza // z linii zawierających nazwy następujących plików

stm32f10x_flash.h
stm32f10x_gpio.h
stm32f10x_rcc.h
misc.h

Spowoduje to dołączenie tych plików do projektu. Plik *stm32f10x_conf.h* należy zapisać i można go zamknąć. Biblioteka *_flash* jest nam potrzebna, gdyż przed ustawieniem częstotliwości zegara SYSCLK na 72 MHz musimy dodać dwa *wait states* do cyklu odczytu Flasha. Szczegóły w rozdziale 3.1 dokumentu **PM0075** (Tablica 4.2). Bity LATENCY są w rejestrze sterującym FLASH_ACR.

4.11.3. Inicjalizacja RCC i GPIO

Otwieramy plik *main.c* Zaczniemy od dołączenia biblioteki definiującej zmienną logiczną, która definuje także dwie stałe (pisane małymi literami) *true* i *false*. Następnie wpisujemy prototypy i funkcję *main.c*

```
/*****************************************************
 * ZL27ARM: Projekt01_LED                            *
 *****************************************************/
#include "stm32f10x.h"
#include <stdbool.h>    // true, false
int  RCC_Config(void);
void GPIO_Config(void);
void Delay(unsigned int);

int main(void) {
   RCC_Config();
   GPIO_Config();
   GPIO_ResetBits(GPIOB, 0xFF00u);    // zgasimy wszystkie diody LED
   while(1) {
       GPIO_WriteBit(GPIOB, GPIO_Pin_8, Bit_SET);
       Delay(2000000);
       GPIO_WriteBit(GPIOB, GPIO_Pin_8, Bit_RESET);
       Delay(2000000);
   }
}
```

Pod *main()* umieścimy inicjalizację RCC, inicjalizację GPIO oraz opóźnienie programowe. Sygnały taktujące w RCC skonfigurujemy na najwyższe dopuszczalne częstotliwości. HCLK i SYSCLK (taktuje procesor) 72 MHz, PCLK1 38 MHz, PCLK2 72 MHz (Rys. 4.22). Wykorzystamy funkcje CMSIS z pliku *stm32f10x_rcc.c*

```
int RCC_Config(void) {
// konfiguracja sygnałów taktujących RCC
   ErrorStatus HSEStartUpStatus;    // status uruchomienia HSE
   RCC_DeInit();                    // RCC na 'default reset state'
```

```
   RCC_HSEConfig(RCC_HSE_ON);          // włącz HSE
   HSEStartUpStatus = RCC_WaitForHSEStartUp();  // czekaj na gotowość HSE
   if(HSEStartUpStatus == SUCCESS) {
       FLASH_PrefetchBufferCmd(FLASH_PrefetchBuffer_Enable);
       FLASH_SetLatency(FLASH_Latency_2);     // wait-states = 2 (72MHz)
       RCC_HCLKConfig(RCC_SYSCLK_Div1);       // HCLK = SYSCLK
       RCC_PCLK2Config(RCC_HCLK_Div1);        // PCLK2 = HCLK
       RCC_PCLK1Config(RCC_HCLK_Div2);        // PCLK1 = HCLK/2
       RCC_PLLConfig(RCC_PLLSource_HSE_Div1, RCC_PLLMul_9);  // 9*8=72MHz
       RCC_PLLCmd(ENABLE);                    // włącz PLL
       while(RCC_GetFlagStatus(RCC_FLAG_PLLRDY)==RESET); // czekaj na PLL
       RCC_SYSCLKConfig(RCC_SYSCLKSource_PLLCLK); // ustaw PLL jako zegar
       while(RCC_GetSYSCLKSource() != 0x08); // czekaj aż PLL --> SYSCLK
       // konfiguracja sygnałów taktujących używanych peryferiali
       RCC_APB2PeriphClockCmd(RCC_APB2Periph_GPIOB, ENABLE); // GPIOB
       return 0;
   }
   else
       return 1;
}

void GPIO_Config(void) {
   // konfigurowanie portów GPIO
   GPIO_InitTypeDef GPIO_InitStructure;
   GPIO_InitStructure.GPIO_Pin = 0xFF00u;     // piny 15, 14,... 9, 8
   GPIO_InitStructure.GPIO_Speed = GPIO_Speed_2MHz;
   GPIO_InitStructure.GPIO_Mode = GPIO_Mode_Out_PP;    // wyjście push-pull
   GPIO_Init(GPIOB, &GPIO_InitStructure);
}
void Delay(unsigned int counter) {
   // opóźnienie programowe, 8'000'000 to jedna sekunda
   while (counter--) {
       __NOP();
       __NOP();
   }
}
```

Analizą czasową funkcji *Delay()* zajmiemy się w rozdziale 13.4.1. Tam będzie wyjaśnone, dlaczego są dwie instrukcje `NOP`, a nie jedna.

Konfigurowanie RCC zaczynamy od wyzerowania jego ustawień. Jest to konieczne, gdyż ustawienia są powiązane ze sobą i muszą być wykonane w określonej kolejności. Następnie włączamy oscylator HSE synchronizowany zewnętrznym kwarcem 8MHz i czekamy na jego gotowość, to znaczy na stan, gdy HSE generuje stabilny sygnał taktujący. Jeżeli HSE nie wystartuje, to wracamy z wartością *false* i system jest taktowany z HSI. Natomiast jeżeli HSE wystartuje (częstotliwość jest równa częstotliwości dołączonego kwarcu: HSE = 8MHz), włączamy buforowanie Flasha i programujemy dwa *wait states*, wydłużające cykl odczytu Flasha, gdyż maksymalna częstotliwość komunikacji z Flashem to 24 MHz, a my chcemy system taktować 72MHz. Oczywiście cykl wydłużyć musimy przed zwiększeniem częstotliwości taktowania, gdyż program który przeprowadza

inicjalizację "chodzi z Flasha" i nie możemy doprowadzić do zakłócenia odczytywania instrukcji. Dzielnik HCLK programujemy na 1, czyli częstotliwości SYSCLK i HCLK będą jednakowe (docelowo 72MHz). Dzielnik PCLK2 też programujemy na 1, natomiast dzielnik PCLK2 na dwa, czyli częstotliwość PCLK2 będzie równa 36MHz. Na tym etapie nie programujemy sygnałów taktujących ADCCLK ani USBCLK, gdyż nie będziemy ich używać. Teraz możemy już uruchomić PLL z mnożnikiem dziewięć. Czekamy na status gotowości PLL, wtedy ustawiamy PLL jako zegar i czekamy na flagę potwierdzającą że wyjście z PLL jest sygnałem taktującym system. Ważną czynnością, o której musimy pamiętać jest włączenie sygnałów taktujących urządzeń peryferyjnych które będą używane w projekcie.

Kolejna funkcja to inicjalizacja portów GPIO. Mikrokontroler STM32F103VB ma pięć 16-bitowych portów. Rejestry sterujące wszystkich tych portów mają jednakową strukturę, dlatego funkcje CMSIS wykorzystują strukturę języka C do ich programowania. Należy wypełnić strukturę danych: podajemy których linii portu dotyczy inicjalizacja i jaka jest szybkość oraz typ wyjścia dla tych linii. W architekturze GPIO każda linia ma indywidualnie programowaną szybkość i typ wyjścia, inicjalizacja dotyczy tylko linii wskazanych w strukturze i nie zmienia ustawień innych linii w tym samym porcie. Na końcu wywołujemy funkcję konfigurującą ze wskazaniem którego portu dotyczy (A, B, C, D, czy E) i ze wskaźnikiem na strukturę. Obecnie programujemy tylko osiem najstarszych linii PortA, gdyż do nich są przypięte LEDy.

Ostatnią funkcją jest opóźnienie programowe, czyli najgorszy sposób uzyskiwania opóźnień w systemie. Użyjemy opóźnienia programowego gdyż nie uruchomiliśmy jeszcze układów czasowych mikrokontrolera (*timers*).

Kompilujemy i wpisujemy do Flasha nasz program. Powinniśmy uzyskać efekt w postaci migającej diody LED oznaczonej numerem 1.

4.11.4. Z jakim parametrem należy wywołać *Delay()* aby uzyskać 1 sek ?

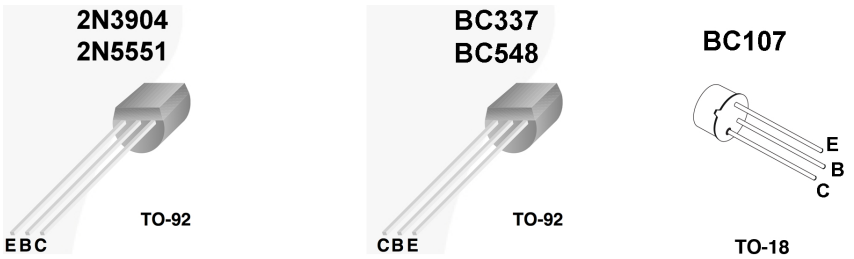
Wykorzystamy stoper w telefonie. Doświadczenie będzie polegało na pomiarze czasu opóźnienia z maksymalną wartością parametru, dzięki temu zwiększymy dokładność pomiaru. W funkcji `int main(void)` po inicjalizacji i zgaszeniu LEDów, a przed pętlą `while` należy wstawić test

```
// Pomiar współczynnika funkcji Delay
   GPIO_WriteBit(GPIOB, GPIO_Pin_8, Bit_SET);
   Delay(8000000);     // czekamy na start pomiaru
   GPIO_WriteBit(GPIOB, GPIO_Pin_8, Bit_RESET);
   GPIO_SetBits(GPIOB, GPIO_Pin_12|GPIO_Pin_13|GPIO_Pin_14|GPIO_Pin_15);
   Delay(0xFFFFFFFF);     // POMIAR: 8 minut i 57 sekund
   GPIO_ResetBits(GPIOB, GPIO_Pin_12|GPIO_Pin_13|GPIO_Pin_14|GPIO_Pin_15);
```

Po uruchomieniu programu zostanie zapalona dioda LED1, następnie zostanie zgaszona i zapalone będą cztery diody z lewej strony. Będą one świeciły około 8 minut i 57 sekund, po czym zgasną i program przejdzie do okresowego zapalania diody LED1. Należy zmierzyć stoperem czas świecenia czterech diod. Załóżmy, że uzyskaliśmy 8 minut i 57 sekund, możemy obliczyć szukaną wartość parametru wywołania

$$N = 2^{32} - 1 = 4'294'967'295, \quad t = 8 \cdot 60 + 57 = 537 \quad \longrightarrow \quad \frac{N}{t} = 7'998'077 \approx 8'000'000$$

Szukaną wartością parametru jest (około) 8'000'000. Możemy teraz wyłaczyć test, zamieniając program testu w komentarz `/* ... */`.



Rys. 4.33: Wyprowadzenia tranzystorów bipolarnych *n-p-n*

4.11.5. Dołączamy zewnętrzną diodę LED

Dołączymy zewnętrzną diodę świecącą zgodnie ze schematem 4.32 i sprawdzimy spełnienie założeń projektu. Wykorzystamy płytkę prototypową, do której przylutujemy podstawkę pod tranzystor i drugą pod diodę LED (będziemy mogli wymieniać te elementy w trakcie prób), dwa oporniki i *gold piny* do połączenia z mikrokontrolerem. Do sterowania bazy tranzystora wykorzystamy linię PE0. Diodę LED będziemy zasilać z +5V. W płytce ZL27ARM musimy wykonać drobną zmianę: środkowy *gold pin* złącza PWR_SEL wymienimy na dłuższy - będzie to nasze źródło napięcia V_{CC}=5V. Dobrą praktyką jest wykonywanie wszystkich podłączeń przy wyłączonym napięciu zasilania. Dlatego wyłączamy komputer i zasilaną z niego płytkę ZL27ARM. Łączymy zasilanie płytki prototypowej do V_{CC}=5V, masę do masy, a wejście sterujące do linii PE0 w złączu JP10 (schemat płyty ZL27ARM jest w dodatku B). Do podstawek wkładamy tranzystor (rysunek 4.33) i diodę. Sprawdzamy połączenia i możemy włączyć komputer. Sprawdzimy działanie nowego układu: dołączenie przewodu sterującego bazą tranzystora do $V_{DD} = 3.3V$ (w tym samym złączu JP10), powinno spowodować zaświecenie, a dołączenie do GND - zgaszenie diody LED. Jeżeli układ działa poprawnie dołączamy przewód sterujący bazą do pinu PE0.

W programie musimy załączyć taktowanie portu E i zainicjować linię PE0 jako wyjście *push-pull*, 2MHz. W funkcji `RCC_Config(void)`, pod komentarzem "konfiguracja sygnałów taktujących używanych peryferiali" należy załączyć taktowanie portu E

```
RCC_APB2PeriphClockCmd(RCC_APB2Periph_GPIOE, ENABLE); // GPIOE
```

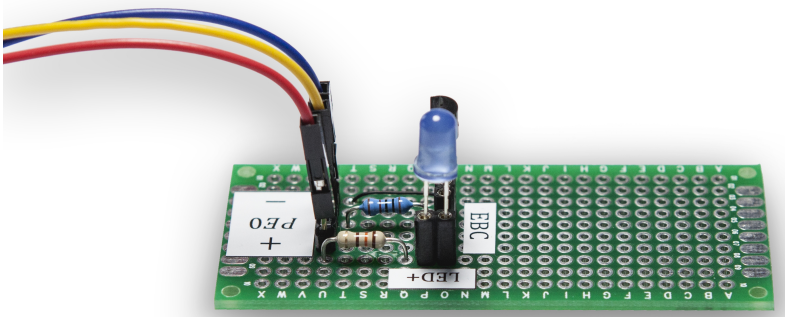
Do funkcji `GPIO_Config(void)` dodamy inicjalizację portu PE0

```
GPIO_InitStructure.GPIO_Pin = GPIO_Pin_0;           // pin 0
GPIO_InitStructure.GPIO_Speed = GPIO_Speed_2MHz;
GPIO_InitStructure.GPIO_Mode = GPIO_Mode_Out_PP;    // wyjście push-pull
GPIO_Init(GPIOE, &GPIO_InitStructure);
```

Funkcję `main(void)` rozbudujemy o sterowanie diodą zewnętrzną

```
while(1) {
    GPIO_WriteBit(GPIOB, GPIO_Pin_8, Bit_SET);
    GPIO_WriteBit(GPIOE, GPIO_Pin_0, Bit_RESET); // zgaś LED zewnętrzny
    Delay(2000000);
    GPIO_WriteBit(GPIOB, GPIO_Pin_8, Bit_RESET);
    GPIO_WriteBit(GPIOE, GPIO_Pin_0, Bit_SET);
    Delay(2000000);
}
```

Powinniśmy uzyskać naprzemienne zapalanie diody wewnętrznej i zewnętrznej.

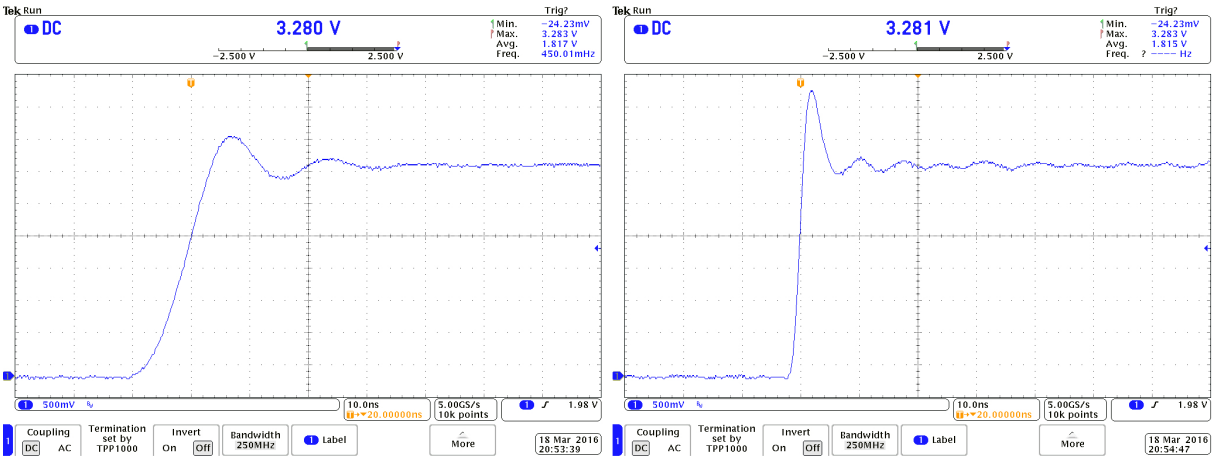


Rys. 4.34: Przykład realizacji układu tranzystor *n-p-n* i LED, na płytce prototypowej

Kolejne ćwiczenie: zmierzymy napięcia na tranzystorze i diodach LED różnych kolorów. Zaczniemy od "zakomentowania" linii gaszącej LED zewnętrzny. Uruchamiamy program po poprawce: LED zewnętrzny jest zapalony na stałe. Bez wyłączania zasilania będziemy zmieniać tranzystory oraz LEDy. Zmieniając tranzystor należy pamiętać o polaryzacji nóżek (rysunek 4.33). Odwrotne włożenie LEDa nic nie uszkodzi, ale nie będzie świecił gdyż próbujemy prądem zaporowym zmusić go do świecenia.

4.11.6. Oscyloskop: szybkość wyjścia GPIO

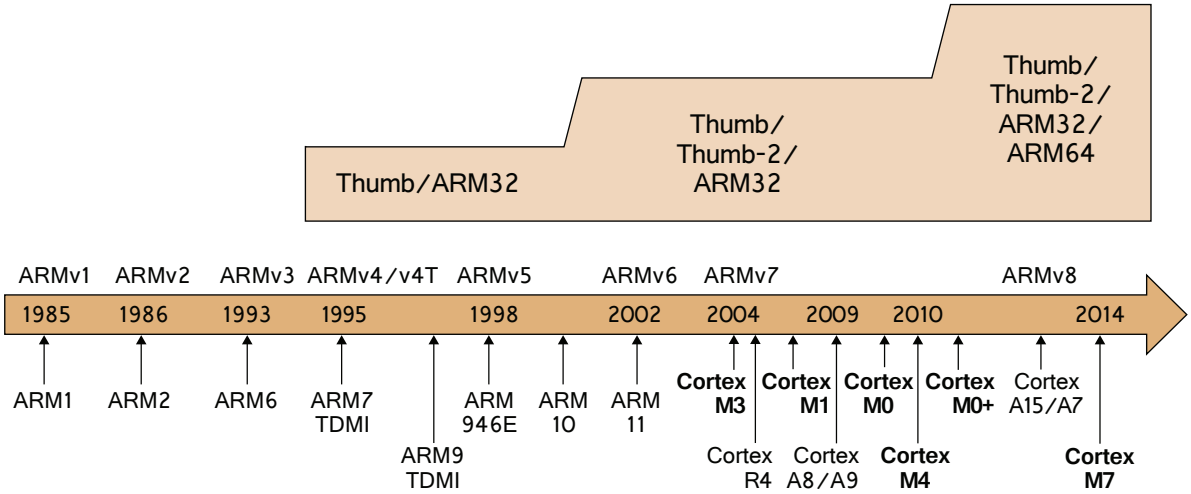
Celem tego ćwiczenia jest sprawdzenie wpływu różnych konfiguracji szybkości przełączania portu GPIO na kształt sygnału wyjściowego. Odłączymy układ zewnętrznego LEDa i pomiędzy PE0 i masę GND włączymy kondensator ceramiczny 33 pF. Usuniemy opóźnienie programowe. Wyjście `PE0` jest naprzemiennie ustawiane w stan wysoki i stan niski. Programując szybkość przełączania portu GPIOE początkowo na wartość `GPIO_Speed_2MHz`, a następnie na `GPIO_Speed_50MHz` możemy porównać kształt zbocza narastającego sygnału. Oscylogramy pokazano na rysunku 4.35 (2 MHz z lewej i 50 MHz z prawej strony).



Rys. 4.35: Porównanie oscylogramów: wyjście PE0

Rozdział 5

Zestaw instrukcji ARM ISA

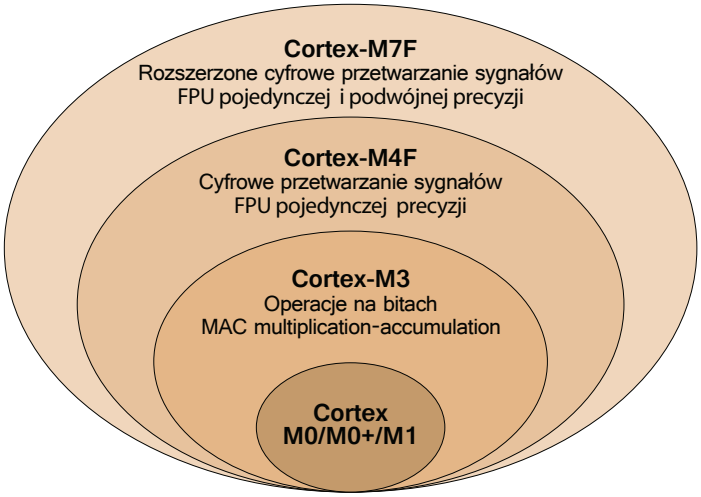


Rys. 5.1: Historia rozwoju procesorów ARM i ich zestawów instrukcji

5.1. Zestawy instrukcji ARM

Procesory ARM wspierają cztery różne zestawy instrukcji: Thumb (kciuk), Thumb-2, ARM32 i ARM64. Rysunek 5.1 pokazuje historię rozwoju procesorów ARM.

- ***Thumb***. Zestaw Thumb stworzono w celu zwiększenia gęstości kodu, dlatego wszystkie instrukcje są 16-bitowe. Osiągnięto to przez ograniczenie różnorodności operandów oraz ograniczenie zakresu dostępnych rejestrów. Zmniejszenie rozmiaru kodu zmniejsza koszt mikrokontrolera i wydłuża czas życia baterii.
- ***ARM32*** zawiera instrukcje 32-bitowe. Uzyskujemy większą elastyczność kodu, większy zakres wartości operandów i zakres skoków relatywnych, dostęp do wszystkich rejestrów. Dzięki temu zwiększa się moc obliczeniowa ("instrukcje mogą więcej"), ale zmniejsza się gęstość kodu.
- ***Thumb-2***. Jest kompromisem pomiędzy Thumb i ARM32. Szuka równowagi pomiędzy wydajnością i gęstością kodu. Zawiera cały zestaw Thumb i wybrane instrukcje ARM32. Celem jest duża gęstość, zbliżona do Thumb i wydajność porównywalna z ARM32.
- ***ARM64*** Zestaw instrukcji 64-bitowych wykorzystywany w procesorach o największych mocach obliczeniowych, stosowanych np. w serwerach.



Rys. 5.2: Rodzina ARM Cortex-M

Rodzina procesorów ARM Cortex dzieli się na trzy grupy

- Cortex-M do mikrokontrolerów (M = microcontroller).
- Cortex-R do rozbudowanych systemów czasu rzeczywistego (R = real-time)
- Cortex-A do aplikacji wymagających dużych mocy (A = application)

Cortex-M zapewnia najlepsze połączenie wydajności, ceny i efektywności energetycznej. Cortex-M integruje rdzeń procesora z pamięciami i wieloma urządzeniami wejścia / wyjścia. Rodzina Cortex-M to procesory: Cortex-M0, Cortex-M0+, Cortex-M1, Cortex-M3, Cortex-M4 i Cortex-M7. Procesory Cortex-M0/M0+/M1 mają architekturę ARMv6-M Von Neumann, natomiast Cortex-M3/M4/M7 architekturę ARMv7-M Harvard.

Wszystkie procesory Cortex-M są kompatybilne wstecz. Rysunek 5.2 pokazuje, jak rozszerzony jest zestaw instrukcji każdego procesora w stosunku do poprzedników. Program skompilowany dla procesora Cortex-M3 będzie działał na Cortex-M4 bez żadnych modyfikacji.

Koprocesor arytmetyczny zmiennoprzecinkowy FPU jest opcjonalnym wyposażeniem procesorów M4 i M7, które wykonują także instrukcje SIMD *single-instruction multiple-data* i MAC *multiply-and-accumulate* wykorzystywane w cyfrowym przetwarzaniu sygnałów.

Ta książka ogranicza się do architektury ARMv7-M, zgodnie z którą zbudowane są procesory Cortex-M3, M4 i M7. Architektura ARMv7-M zawiera zestaw instrukcji Thumb-2 i nie obsługuje pełnego zestawu ARM32. W drugiej części książki opisane są instrukcje operacji zmiennoprzecinkowych FPU dostępne w procesorach M4 i M7.

5.2. Budowa procesora Cortex-M

Mikrokontroler Cortex-M zawiera rdzeń ze zintegrowanymi peryferialami (np.NVIC) licencjonowany przez ARM, dodatkowe układy dodane przez producenta oraz magistrale do komunikacji pomiędzy rdzeniem a układami peryferyjnymi. Przykłady układów zintegrowanych z rdzeniem w tym samym krzemie, to I^2C, SPI, USART, przetworniki ADC i

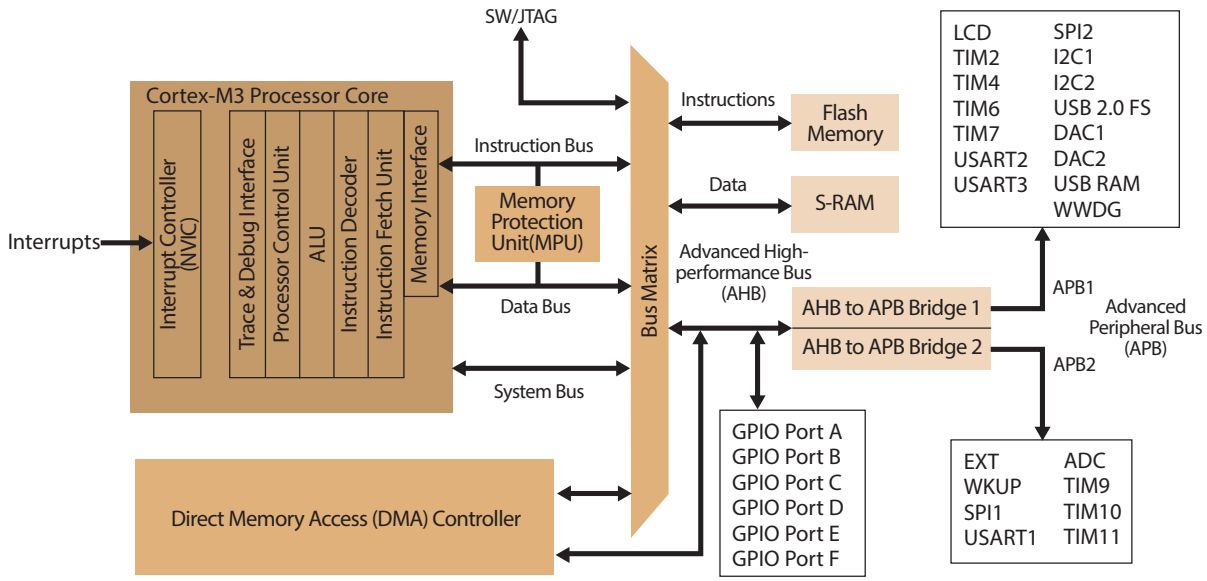
DAC. Różni producenci mogą dodawać różne układy peryferyjne. Schemat mikrokontrolera pokazany jest na rysunku 5.3

- *Bus Matrix* zapewnia komunikację pomiędzy rdzeniem a pamięcią instrukcji Flash, pamięcią danych S-RAM, układem bezpośredniego dostępu do pamięci DMA i portami GPIO. *Bus Matrix* są to połączone ze sobą różne magistrale danych.
- *Bus Matrix* umożliwia równoległą wymianę danych pomiędzy układami dołączonymi do *Bus Matrix* i umożliwia szybkie transfery.
- Do *Bus Matrix* jest dołączona magistrala AHB *Advanced High-performance Bus*, a do niej dwa mosty *Bus Bridges*, jeden most jest wolny (APB1 do 36 MHz), a drugi szybki (APB2 do 72 MHz). Do tych mostów są dołączone układy peryferyjne. Most łączy dwie różne magistrale.
- Porty GPIO mogą być skonfigurowane na funkcje alternatywne, które umożliwiają łączność peryferiali ze światem zewnętrznym.

Magistrala jest kompletem linii transmisyjnych (adresów, danych i sygnałów sterujących), które umożliwiają wymianę danych pomiędzy układami dołączonymi do magistrali, czyli na przykład pomiędzy pamięcią, układami peryferyjnymi a rdzeniem oraz pomiędzy peryferialami i pamięcią bez pośrednictwa rdzenia.

Głównymi składnikami rdzenia procesora są

- *Arithmetic Logic Unit ALU* wykonuje operacje logiczne (AND, OR) oraz arytmetyczne na liczbach całkowitych (ADD, SUB).
- *Processor Control Unit* generuje sygnały sterujące, jak kierunek transferu R/W *Read/Write*, czy strob wyboru CS *Chip Select*
- Kontroler przerwań NVIC pozwala procesorowi przerwać (zatrzymać) wykonywanie bieżącego programu i przekierowuje egzekucję do programu obsługi przerwania wywołanego zdarzeniem zewnętrznym.
- *Instruction Fetch Unit* oraz *Instruction Decoder* realizują potokowe wykonywanie programu, jak pokazano na rysunku 3.12.
- *Trace & Debug Interface* ułatwiają uruchamianie i debugowanie programu.



Rys. 5.3: Schemat blokowy mikrokontrolera ARM Cortex-M3

5.3. Format instrukcji asemblerowej

Instrukcja maszynowa zawiera binarny kod operacji *opcode*, która ma być wykonana oraz zero lub więcej operandów, które definiują dane wejściowe tej operacji. W programie asemblerowym kody operacji są zastąpione mnemonikami *mnemonics*, czyli skrótami literowymi nazw instrukcji. Mnemoniki są bardziej czytelne od kodów binarnych (ciągów zer i jedynek). Format instrukcji asemblera ARM Keil jest następujący:

```
etykieta    mnemonik  operand1, operand2, operand3        ; komentarz
```

- **etykieta** wskazuje adres (w pamięci) bieżącej instrukcji. Asembler zastępuje etykietę adresem bezwzględnym lub przesunięciem np. w stosunku do bieżącej zawartości licznika programu PC *Program Counter*. Etykieta musi być unikalna wewnątrz pliku. Etykieta musi zaczynać się z początkiem linii, bez żadnych odstępów (spacji czy tabulacji),
- **mnemonik** symbolizuje operację która ma być wykonana,
- ilość **operand**ów zależy od typu instrukcji. Niektóre instrukcje w ogóle nie mają operandów. Operandy są rozdzielone przecinkami. W pewnych instrukcjach operandem może być liczba stała, także kod znaku. Jest to operand natychmiastowy (*immediate*),
- najczęściej do pierwszego operandu zostanie wpisany wynik, czyli jest to ***destination register***. Operandy drugi i trzeci zawierają dane wejściowe operacji, czyli są to ***source operands***. Drugi operand (operand2) to najczęściej rejestr. Ostatni (operand3) to może być rejestr, stała natychmiastowa lub rejestr + przesunięcie.
- Wszystko co za średnikiem, aż do końca linii to komentarz

Przykłady instrukcji asemblerowych

```
ADD  r0, r2, r3    ; r0 = r2 + r3
SUB  r3, r0, #3    ; r3 = r0 - 3
MOV  r0, #'M'      ; r0 = kod ASCII znaku 'M', czyli 0x4D
```

Warianty instrukcji dodawania liczb całkowitych

```
ADD  r1, r2, r3    ; r1 = r2 + r3
ADD  r1, r3        ; r1 = r1 + r3
ADD  r1, r2, #4    ; r1 = r2 + 4
ADD  r1, #15       ; r1 = r1 + 15
```

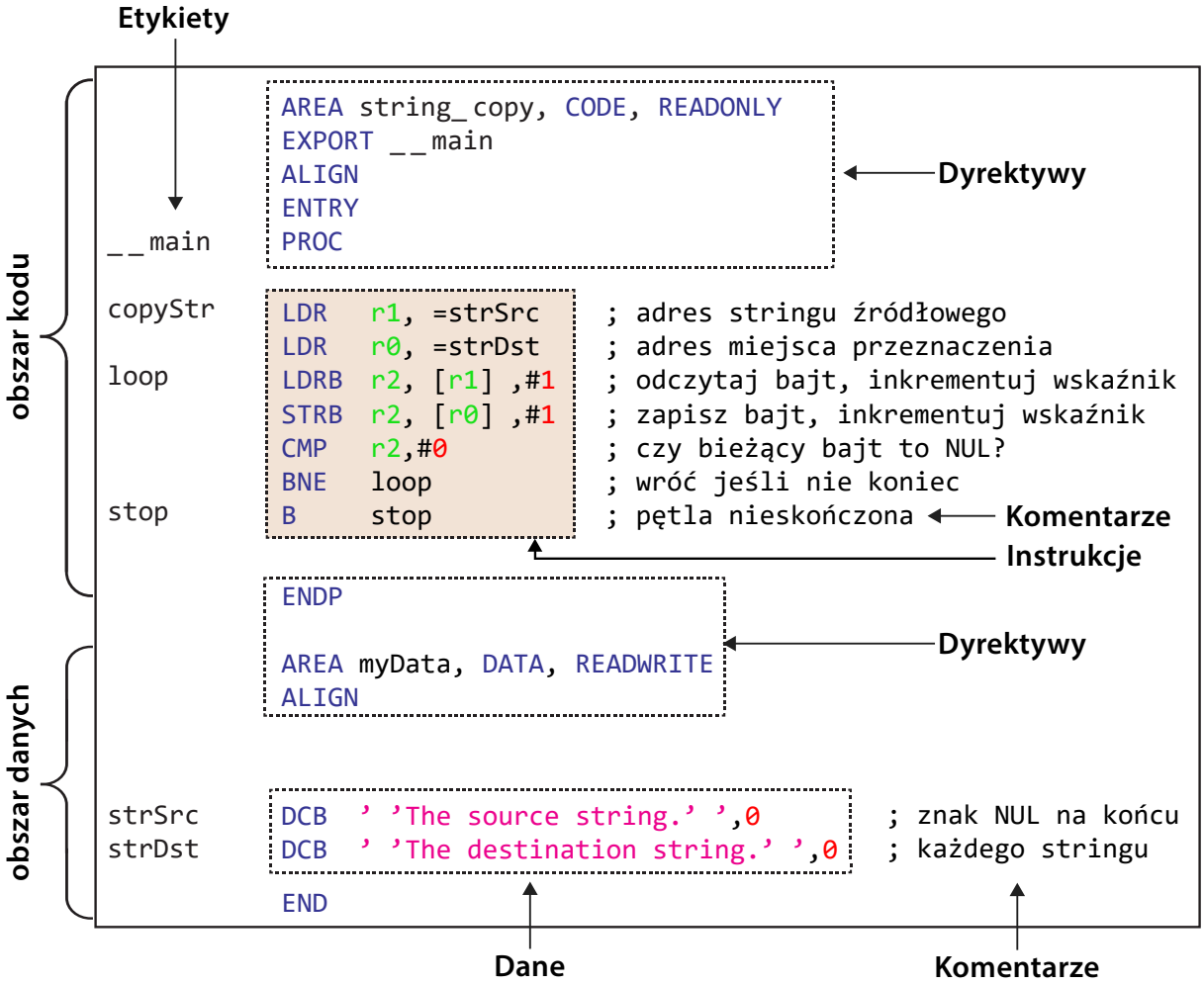
W tej samej instrukcji ilość operandów może być różna. Gdy operand *destination* jest taki sam jak pierwszy operand źródła (operand2), to operand *destination* zostaje pominięty. Drugi operand może mieć różną postać (jak w instrukcjach przesunięcia lub w adresowaniu pamięci) i jest często oznaczany symbolem `Op2` w opisie instrukcji. Na przykład instrukcja dodawania jest opisana następująco

```
ADD {Rd,} Rn, Op2    ; Rd = Rn + Op2
```

Nawiasy klamrowe { } oznaczają, że operand `Rd` jest opcjonalny, gdy jest taki sam jak `Rn`. W wielu instrukcjach ostatni operand (operand2 lub operand3) może mieć różną formę, przykładami są instrukcje przesunięcia.

5.4. Struktura programu asemblerowego

Przyjrzymy się teraz kompletnemu programowi asemblerowemu, pokazanemu na rysunku 5.4. Program kopiuje string źródłowy `strSrc` zakończony znakiem NUL, w miejsce stringu docelowego `strDst`. Program zawiera etykiety, dyrektywy, instrukcje asemblera i komentarze. Omówimy teraz kolejno wszystkie te składniki programu.



Rys. 5.4: Przykładowy program asemblerowy

Etykieta *label*. Na rysunku 5.4 etykietami są `__main`, `copyStr`, `loop`, `stop`, `strSrc`, `strDst`. Reprezentują one adres pamięci zajmowanej przez wskazywaną instrukcję lub wskazywane dane. Asembler zamienia każdą etykietę bezwzględnym adresem pamięci, lub przesunięciem względem licznika programu PC. Etykieta musi zaczynać się z początkiem linii, bez żadnych poprzedzających spacji lub innych znaków odstępu. Etykieta może być nazwą funkcji (jak `__main`) i jest wtedy adresem pierwszej instrukcji. Etykieta `__main` będzie wyeksportowana, co pozwoli linkerowi ją rozpoznać i umożliwi połączenie funkcji z resztą programu.

Dyrektywy dostarczają ważnych informacji asemblerowi. W przykładzie na rysunku 5.4, dyrektywy `PROC` i `ENDP` określają początek i koniec funkcji (podprogramu). `END` oznacza koniec pliku. `AREA` definiuje obszar instrukcji i obszar danych. `ENTRY` oznacza miejsce wejścia do całego programu. `ALIGN` realizuje wymagania na wyrównanie adresu instrukcji lub danych. `DCB` definiuje i lokuje (umieszcza w pamięci) dane.

Instrukcje asemblera są to rozkazy maszynowe, które sterują wykonywaniem programu (skoki) i manipulują danymi. Występują także pseudoinstrukcje, które nie generują rozkazu maszynowego dokładnie odpowiadającego ich mnemonikowi, ale powodują określone działanie asemblera. Przykładem pseudoinstrukcji jest `LDR r1, =strSrc`. Asempler umieści pełny 32-bitowy adres w określonym miejscu pamięci programu (pod kodem funkcji) i wygeneruje instrukcję która załaduje ten adres do rejestru `r1` z zastosowaniem adresowania względnego. Pseudoinstrukcje bardzo ułatwiają życie programiście.

Komentarz jest to krótka notatka wyjaśniająca intencje programisty i zaczyna się od średnika. Umieszczanie komentarzy w programie jest bardzo ważne, bo gdy wracamy do własnego programu sprzed tygodnia, to nie zawsze pamiętamy co on miał robić. Asembler pomija wszystkie znaki od średnika do końca linii.

Obszar danych *data area* jest miejscem w którym pamiętane są dwa stringi `strSrc` oraz `strDst`. Program przydziela miejsce w pamięci i nadaje im wartości początkowe. NUL jest znacznikiem końca stringu.

Obszar kodu *code area* zawiera funkcję `__main`, która jest odpowiednikiem funkcji `main(void)` w kodzie źródłowym C.

Każda Instrukcja procesora Cortex-M może być zakwalifikowana do jednej z czterech kategorii:

- instrukcje arytmetyczne *arithmetic*, przesunięcia *shift* i logiczne *logic*, rozdział 6
- przesyłania danych *store* & *load*, rozdział 7
- porównywania *compare* i skoków warunkowych *branch*, rozdział 8
- wszystkie pozostałe, czyli niewielka ilość instrukcji takich jak `NOP` = nic nie rób

Lista instrukcji procesorów Cortex-M4/M7 obejmuje ponadto:

- instrukcje dedykowane do cyfrowego przetwarzania sygnałów
- instrukcje arytmetyki zmiennoprzecinkowej FP

5.5. Dyrektywy asemblera

W programach asemblerowych dyrektywy dostarczają kluczowych informacji dla kompilatora i linkera, takich jak deklaracje stałych i nazw symbolicznych, definicje danych, alokacja obszarów pamięci, określenie struktury programu i punktu wejścia `main()`.

Dyrektywa `AREA`

Cały program składa się z bloków kodu i bloków danych. Dyrektywa `AREA` wskazuje asemblerowi początek nowego bloku kodu lub bloku danych. Blok kodu zawiera listę instrukcji, blok danych to deklaracje i inicjalizacje zmiennych. Dla linkera, bloki są podstawowymi, niezależnymi i niepodzielnymi obszarami pamięci. Każdy blok musi mieć unikalną nazwę w ramach całego programu. Program asemblerowy musi mieć co najmniej jeden blok kodu programu. Domyślnie, bloki kodu są umieszczane w pamięci stałej `READONLY`, a bloki danych w pamięci o dostępie swobodnym `READWRITE`.

Tabela 5.1: Podstawowe dyrektywy asemblera

AREA	Tworzy nowy blok kodu lub danych
ENTRY	Deklaruje punkt wejścia do programu → początek funkcji *main()*
ALIGN	Wyrównuje obszar pamięci do zadanej wartości
DCB	Lokuje (umieszcza w pamięci) jeden (lub więcej) bajt (8 bitów)
DCW	Lokuje jedno (lub więcej) pół-słowo (16 bitów)
DCD	Lokuje jedno (lub więcej) słowo (32 bity)
SPACE	Lokuje wyzerowany blok pamięci o zadanej wielkości
FILL	Lokuje blok pamięci i wypełnia go zadaną wartością
EQU	Nadaje nazwę symboliczną stałej liczbowej
RN	Nadaje nazwę symboliczną rejestrowi procesora
EXPORT	Deklaruje symbol dostępny dla innych plików programu
IMPORT	Udostępnia bieżącemu programowi symbol zdefiniowany w innych plikach
INCLUDE/GET	Dołącza inny plik źródłowy do bieżącego programu
PROC	Deklaruje początek bieżącego programu (funkcji, podprogramu)
ENDP	Wyznacza koniec bieżącego programu
END	Oznacza koniec pliku źródłowego

Tabela 5.2: Podstawowy szkielet programu asemblerowego

```
        AREA, myData, DATA, READWRITE   ; Definiuje sekcję danych
Array   DCD 1, 2, 3, 4, 5               ; Tablica liczb całkowitych

        AREA, myCode, CODE, READONLY    ; Definiuje sekcję kodu programu
        EXPORT  __main                  ; Etykieta dostępna dla linkera
        ENTRY                           ; Punkt wejścia do całego programu
__main  PROC                            ; Początek funkcji (podprogramu)
        ...                             ; Ciało funkcji
        ENDP                            ; Koniec funkcji (podprogramu)
        END                             ; Koniec pliku
```

Dyrektywa ENTRY

Dyrektywa ta wskazuje na pierwszą instrukcję całej aplikacji. Musi być jedna i tylko jedna dyrektywa ENTRY w całej aplikacji, bez względu na to ile jest bloków kodu. Gdy nie ma żadnej dyrektywy ENTRY, linker zgłosi błąd. Gdy jest więcej niż jedna, błąd zgłosi asembler. Jeżeli cała aplikacja jest pisana w C, dyrektywa ENTRY jest w jednym z programów bibliotecznych, a nie w kodzie pisanym przez programistę.

Dyrektywa END

Dyrektywa END wskazuje na koniec pliku. Każdy plik z programem asemblerowym musi kończyć się tą dyrektywą. Kiedy jeden plik używa dyrektyw INCLUDE lub GET w celu dołączenia innego pliku, asembler powraca do pliku nadrzędnego (tego który wywołuje) po napotkaniu END w pliku dołączanym. Dyrektywa END w pliku najbardziej nadrzędnym (ewentualnie ostatnim) kończy całą aplikację.

Definicja funkcji lub podprogramu, dyrektywy PROC i ENDP

PROC i ENDP oznaczają początek i koniec funkcji (podprogramu). Jeden plik źródłowy może zawierać wiele podprogramów. Dyrektywy PROC i ENDP definiują początek i koniec

każdego podprogramu. Dyrektywy PROC i ENDP nie mogą być zagnieżdżone, czyli nie można definiować podprogramów wewnątrz innych podprogramów.

W całej aplikacji musi być albo jedna funkcja napisana w kodzie źródłowym C o nazwie `main()`, albo jedna asemblerowa o nazwie `__main`.

Dyrektywy alokacji danych DCB, DCW, DCD, DCQ, SPACE oraz FILL
Program asemblerowy definiuje obszary danych. Te obszary są wypełniane wartością stałą lub zmiennymi o nadanych wartościach początkowych. Tablica 5.3 zawiera listę dyrektyw alokacji danych. W tablicy 5.4 są przykłady deklarowania stringu, liczb całkowitych, wyzerowanego i wypełnionego obszaru pamięci oraz kilku zmiennych w różnych formatach. Operandy dyrektywy `FILL` to: ilość bajtów, wzorzec wypełnienia, ilość bajtów wzorca (może być 1, 2 albo 4).

Tabela 5.3: Dyrektywy alokacji danych

Dyrektywa	Opis	Obszar pamięci
DCB	Definiuj bajty	Rezerwuje dane 8-bitowe
DCW	Definiuj pół-słowa	Rezerwuje dane 16-bitowe
DCD	Definiuj słowa	Rezerwuje dane 32-bitowe
DCQ	Definiuj podwójne słowa	Rezerwuje dane 64-bitowe
SPACE	Definiuj bajty wyzerowane	Rezerwuje zadaną ilość zerowych bajtów
FILL	Definiuj zainicjowane bajty	Rezerwuje i wypełnia każdy bajt wartością

Tabela 5.4: Przykłady dekladacji danych

```
          AREA, myData, DATA, READWRITE
hello     DCB ''Hello World!'',0   ; String zakończony znakiem NUL
dollar    DCB 2,10,0,200           ; Liczby całkowite, 8-bitowe
scores    DCD 2,3.5,-0.8,4.0       ; Liczby rzeczywiste (float), 32-bitowe
miles     DCW 100,200,50,0         ; Liczby całkowite 16-bitowe
p         SPACE 255                ; Wyzerowany obszar 255 bajtów
f         FILL 20,0xFF,1           ; Obszar wypełniony jedynkami, 20 bajtów
binary    DCB 2_01010101           ; Jeden bajt zainicjowany liczbą binarną
octal     DCB 8_73                 ; Jeden bajt zainicjowany liczbą oktalną
char      DCB 'A'                  ; Jeden bajt zainicjowany znakiem ASCII
```

Dyrektywy EQU i RN
Dyrektywy `EQU` i `RN` pomagają uczynić program asemblerowy bardziej zrozumiałym. `EQU` pozwala nadać nazwę stałej liczbowej. Jej użycie jest bardzo podobne do `#define` w programach C.

Tabela 5.5: Przykłady definicji `EQU`

```
; Interrupt Number Definition (IRQn)
BusFault_IRQ   EQU -11     ; Cortex-M Bus Fault Interrupt
SVCall_IRQ     EQU  -5     ; Cortex-M Supervisor Call (SVC) Interrupt
PendSV_IRQ     EQU  -2     ; Cortex-M Pend SVC Interrupt
SysTick_IRQ    EQU  -1     ; Cortex-M System Tick Interrupt
```

Dyrektywa RN pozwala nazywać rejestry

Tabela 5.6: Przykłady definicji RN

```
Licznik       RN  6       ; w r6 jest licznik ułamka
Mianownik     RN  5       ; w r5 jest mianownik ułamka
```

Dyrektywy EXPORT oraz IMPORT

Obie te dyrektywy udostępniają i pozwalają rozpoznawać symbole (np. etykiety) zdefiniowane w różnych plikach należących do projektu. Dyrektywa EXPORT deklaruje symbol i udostępnia go linkerowi. IMPORT powoduje, że symbol, który nie jest zdefiniowany w bieżącym pliku staje się dostępny dla asemblera. Dyrektywa IMPORT działa podobnie jak słowo kluczowe extern w C.

Dyrektywa ALIGN

W celu osiągnięcia maksymalnej możliwej wydajności wiele procesorów wymaga, aby adresy początkowe programów i danych były potęgą dwójki: 2^n. Na przykład adresy pamięci wyrównane do całych słów muszą być podzielne przez 4, czyli 2^2. Procesory Cortex-M pozwalają na dostęp do danych, które nie są wyrównane do słów, ale odbywa się to kosztem wydajności. W takim przypadku potrzebne mogą być dwa dostępy do pamięci zamiast jednego.

Tabela 5.7: Przykład wyrównania kodu i danych

```
     AREA example, CODE, ALIGN=3   ; Początek pamięci wyrównany do 8
     ADD r0, r1, r2                ; Początek programu wyrównany do 8

     AREA myData, DATA, ALIGN=2    ; Początek pamięci wyrównany do 4
a    DCB 0xFF                      ; Pierwszy bajt 4-bajtowego słowa
     ALIGN 4, 3                    ; Wyrównaj do ostatniego bajtu słowa
b    DCB 0x33                      ; Zapisz ostatni bajt 4-bajtowego słowa
c    DCB 0x44                      ; Kolejny bajt zniszczy wyrównanie
     ALIGN                         ; wartość domyślna to ALIGN 4
d    DCD 12345                     ; Przeskoczy 3 bajty i zapisze słowo
```

Dyrektywy INCLUDE i GET

Te dyrektywy umożliwiają dołączenie innego pliku do bieżącego. Jest to użyteczne, gdy chcemy dołączyć symbole stałych, które zostały zdefiniowane z wykorzystaniem dyrektywy EQU w innym pliku.

Tabela 5.8: INCLUDE umożliwia dołączenie stałych zdefiniowanych w innym pliku

```
        INCLUDE constants.s          ; Dołącz symbole stałych
        AREA, myCode, CODE, READONLY ; Definiuje sekcję kodu programu
        EXPORT  _main                ; Etykieta będzie dostępna dla linkera
        ENTRY                        ; Punkt wejścia do całego programu
_main   PROC                         ; Początek funkcji (podprogramu)
        ...                          ; Ciało funkcj
        ENDP                         ; Koniec funkcji (podprogramu)
        END                          ; Koniec pliku
```

Rozdział 6

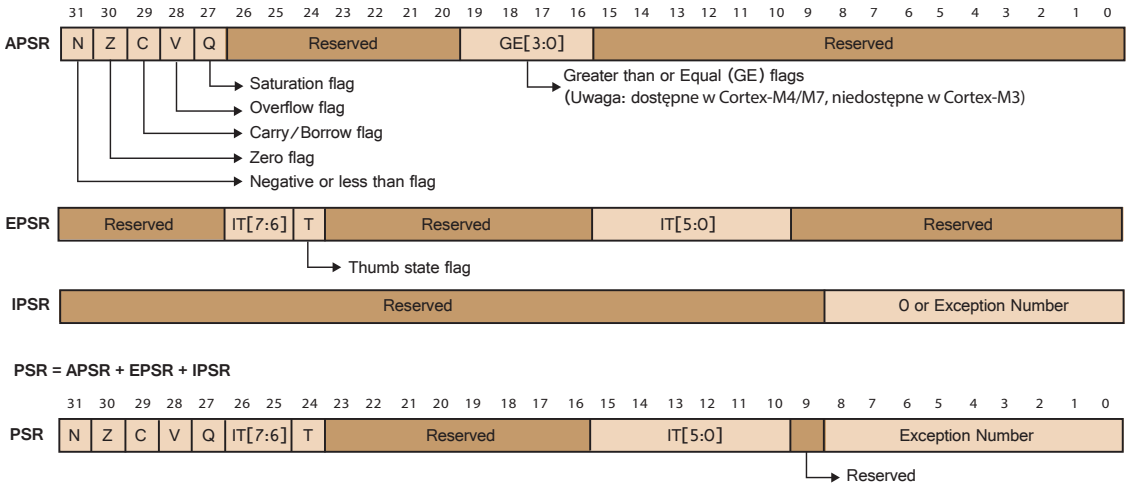
Instrukcje arytmetyczno - logiczne

6.1. Rejestr Flag

Procesor Cortex-M ma pięć flag statusów: liczba ujemna (N), zero (Z), przeniesienie (C), przepełnienie (V) i nasycenie (Q).

- Flaga (N) *Negative* jest ustawiana, gdy wynik jest ujemny i gaszona, gdy dodatni. Procesor przepisuje najstarszy bit wyniku do flagi (N).
- Flaga (Z) *Zero* jest ustawiana, gdy wszystkie bity wyniku są zerowe (wtedy w kodzie U2 wynik jest zerem) i zerowana w przypadku przeciwnym.
- Flaga przeniesienia (C) *Carry* jest przeniesieniem z najstarszego bitu wyniku, patrz rozdział 2.7
- Flaga przepełnienia (V) *Overflow* jest ustawiana gdy występuje przepełnienie wyniku działania na liczbach ze znakiem, patrz rozdział 2.8
- Flaga nasycenia (Q) *Saturation* jest ustawiana, gdy instrukcje `SSAT` i `USAT` wykażą nasycenie, patrz rozdział 6.5

Większość instrukcji procesorów Cortex-M występuje w wariancie z modyfikacją flag i bez modyfikacji. Flagi są najstarszymi bitami rejestru statusu PSR *Program Status Register*, który jest złożeniem trzech rejestrów: APSR *Application Program Status Register*, IPSR *Interrupt Program Status Register* oraz EPSR *Execution Program Status Register*. Ponieważ używane bity tych trzech rejestrów nie pokrywają się, to procesor składa je w jeden rejestr PSR, oznaczany czasami jako xPSR, jak pokazano na rysunku 6.1.



Rys. 6.1: Rejestry statusów APSR, EPSR, IPSR

W rejestrze **APSR** flagi GE wskazują czy wynik jest większy /równy, czy mniejszy od zera. Są dostępne tylko w procesorach Cortex-M4 i M7.

W rejestrze **EPSR** flaga T wskazuje czy procesor realizuje zestaw instrukcji Thumb, czy ARM32. Ponieważ procesory Cortex-M wspierają tylko zestaw Thumb-2, flaga ta ma zawsze wartość jeden. Flagi IT[7:6] oraz IT[5:0] zawierają wskaźniki związane z blokiem IF-THEN (IT).

W rejestrze **IPSR** dziewięć najmłodszych bitów to numer aktualnie obsługiwanego przerwania. Procesor Cortex-M może być w dwóch trybach: obsługi przerwania *Handler Mode* i w trybie normalnej pracy *Thread Mode*. Po RESET procesor jest w trybie *Thread Mode* i w tym trybie bity IPSR[8:0] są wyzerowane. Rozdział 11 zawiera opis systemu przerwań.

Rejestry flag mogą być czytane i zapisywane za pomocą specjalnych instrukcji

- `MRS` (Move Register ← Special) odczytaj rejestr specjalnego przeznaczenia (APSR, IPSR, EPSR, IEPSR, IAPSR, EAPSR, PSR, MSP, PSP, PRIMASK, BASEPRI, BASEPRI_MAX, FAULTMASK, CONTROL) do rejestru ogólnego przeznaczenia
- `MSR` (Move Special ← Register) zapisz rejestr specjalnego przeznaczenia z rejestru ogólnego.

Poniżej kilka przykładów

```
MRS r0, apsr     ; zawartość APSR przepisz do R0
MRS r0, ipsr     ; zawartość IPSR przepisz do R0
MRS r0, epsr     ; zawartość EPSR przepisz do R0
MRS r0, xpsr     ; połączone rejestry APSR+IPSR+EPSR przepisz do R0
MSR apsr_nzcvq, r0  ; zapisz flagi N, Z, C, V, Q z rejestru R0
MSR apsr_g, r0      ; przepisz bity R[19:16] do GE[3:0] w APSR (nie M3)
MSR apsr_nzcvqg, r0 ; zapisz flagi N, Z, C, V, Q, GE z rejestru R0
```

6.2. Modyfikowanie flag

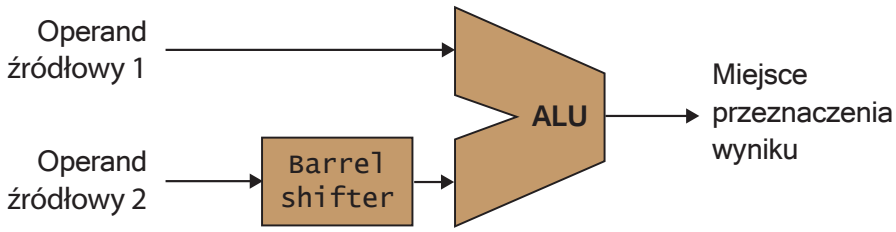
Instrukcje arytmetyczne i logiczne występują także w wersji z modyfikacją flag. Jeżeli mnemonik zawiera przyrostek `S`, to znaczy że instrukcja modyfikuje flagi stosownie do wyniku operacji. Na przykład instrukcja `ADDS` modyfikuje flagi N, V, C i Z w wyniku dodawania, a instrukcja `ADD` nie. Jeżeli instrukcja nie modyfikuje flag, to ich stan pozostaje niezmieniony.

Instrukcje porównania, takie jak `CMP` *Compare*, `CMN` *Compare Negative*, `TST` *Test*, `TEQ` *Test Equivalence*, modyfikują flagi, mimo że nie mają przyrostka `S`. Porównajmy obie wersje instrukcji dodawania liczb całkowitych

```
ADD r1,r2,r3   ; r1=r2+r3, nie zmienia flag N, Z, C, V
ADDS r1,r2,r3  ; r1=r2+r3, modyfikuje flagi N, Z, C, V stosownie do wyniku:
```

- flagę przepełnienia (V) *Overflow* zakładając, że dodawane są liczby ze znakiem w kodzie U2
- flagę przeniesienia (C) *Carry* zakładając, że dodawane są liczby całkowite bez znaku
- ustawia flagę (Z) *Zero* jeśli wszystkie bity wyniku są zerowe
- przepisuje najstarszy bit wyniku do flagi (N) *Negative*

Arytmometr procesorów Cortex-M wyposażony jest w *Barrel Shifter*, czyli szybki układ kombinacyjny, który w jednym takcie zegara może przesunąć (rotować) drugi operand,



Rys. 6.2: *Barrel Shifter* może w jednym cyklu przesunąć drugi operand o n bitów

rysunek 6.2. Jeżeli *Barrel Schifter* jest użyty, operand źródłowy może zmodyfikować flagę *Carry*. Na przykład logiczny operator dwuargumentowy `ANDS` modyfikuje flagi N i Z. W poniższej instrukcji, flaga N będzie ustawiona, gdy najstarszy bit wyniku jest jedynką, flaga Z będzie ustawiona gdy wszystkie bity wyniku są zerami.

```
ANDS r1, r2, r3     ; r1 = r2 AND r3
```

Instrukcja logiczna wykonuje działanie na kolejnych bitach niezależnie od wartości pozostałych bitów, dlatego nie modyfikuje flagi przepełnienia V. W jaki sposób instrukcja `ADDS` może zmodyfikować flagę C *Carry*? Odpowiedź leży w drugim operandzie. Jeżeli drugi operand używa rejestru *Barrel Shifter*, to procesor zmodyfikuje flagę *Carry* w wyniku przesunięcia /rotacji drugiego argumentu (patrz rysunek 6.3), na przykład

```
ANDS r1, r2, r3, LSL #3     ; r1 = r2 AND (r3 <<3)
```

Instrukcja `MOVS`, gdy używa *Barrel Shifter* także modyfikuje flagi N, Z i C

```
MOVS r2, r1, LSR #3     ; modyfikuje N, Z i C
```

Ale *Barrel Shifter* nie modyfikuje flag, gdy jest użyty w instrukcjach arytmetycznych. Na przykład w poniższej instrukcji, stan flag zależy wyłącznie od wyniku działania, a nie od logicznego przesunięcia drugiego operandu

```
ADDS r1, r2, r3, LSL #3     ; r1 = r2 + (r3 << 3)
```

Programista piszący w asemblerze jest odpowiedzialny za prawidłowe stosowanie i interpretowanie flag. W przypadku języków wysokiego poziomu, jak C, to kompilator dba o właściwe stosowanie flag. Jak wyjaśniono w rozdziale 2.8, procesor modyfikuje flagi, a programista musi wiedzieć, jakie operacje wykonuje, i którą flagę należy wykorzystać.

6.3. Instrukcje przesunięcia i obrotu

Jak pokazano na rysunku 6.2, drugi operand działań arytmetycznych i logicznych jest wyposażony w *Barrel Shifter*. Jest to szybki układ kombinacyjny, który w jednym takcie zegara przesuwa lub rotuje drugi operand o zadaną ilość bitów (pozycji). Dostępne operacje przesunięcia /rotacji to (rysunek 6.3):

- **LSL** *Logical Shift Left* przesunięcie logiczne w lewo o n bitów. Na zwalniane pozycje wchodzą zera. Ta operacja jest równoważna mnożeniu operandu przez 2^n.

- **LSR** *Logical Shift Right* przesunięcie logiczne w prawo o n bitów. Na zwalniane pozycje wchodzą zera. Ta operacja jest równoważna dzieleniu operandu przez 2^n.
- **ASR** *Arithmetic Shift Right* przesunięcie arytmetyczne w prawo o n bitów. Najstarszy bit jest powielany, czyli liczba zachowuje swój znak. Ta operacja jest równoważna dzieleniu liczby ze znakiem przez 2^n.
- **ROR** *Rotate Right* obrót zawartości rejestru 32-bitowego o zadaną ilość bitów, rysunek 6.3. Nie ma instrukcji "rotate left" - ten sam efekt można uzyskać obracając odpowiednio w prawo, na przykład obrót o jeden w lewo to obrót o 31 w prawo.
- **RRX** *Rotate Right with eXtend* obrót w prawo (tylko o jeden bit) zawartości rejestru rozszerzonej o przeniesienie C. W odróżnieniu od pozostałych instrukcji, RRX wykonuje obrót tylko o jeden bit.

Dzięki połączeniu układu *Barrel Shifter* i jednostki arytmetyczno-logicznej ALU (rysunek 6.2) operand drugi sam pozostaje niezmieniony, a po przekształceniu jest wpisywany do rejestru przeznaczenia. Przykłady

```
LSL r1, r2        ; r1 = r1 << r2
LSL r1, #3        ; r1 = r1 << 3
LSL r1, r2, #3    ; r1 = r2 << 3
LSL r1, r2, r3    ; r1 = r2 << r3
ROR r1, r2        ; r1 = obróć r1 o r2 bitów
RRX r1, r2        ; r1 = (r2+C) obrócone o jeden bit
```

Bit C na rysunku 6.3 **to nie jest flaga *Carry*** z rejestru APSR/xPSR. Żadna instrukcja przesunięcia /obrotu nie modyfikuje flag sama z siebie: potrzebny jest przyrostek `S`. Ponadto te instrukcje nie modyfikują flagi przepełnienia V.

```
LSL r1, #3        ; r1 = r1 << 3, nie modyfikuj flag
LSLS r1, #3       ; r1 = r1 << 3, modyfikuj flagi N, Z, C
```

Układ *Barrel Shifter* umożliwia zastąpienie wolnego mnożenia /dzielenia, szybkim przesuwaniem

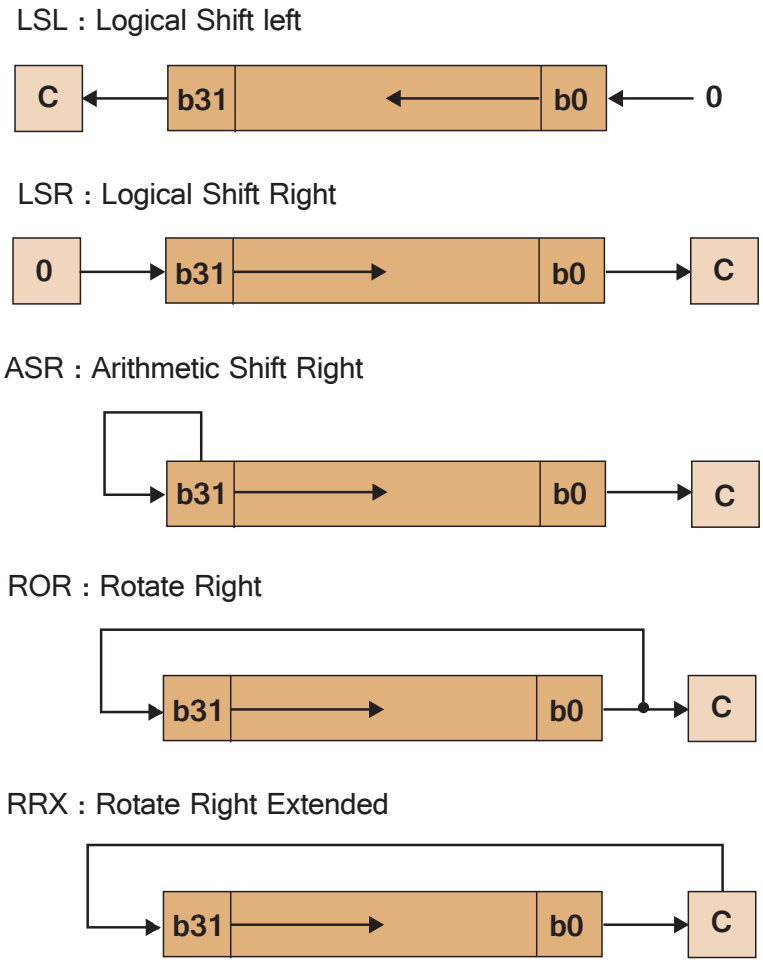
```
ADD r0, r2, r1, LSL #3    ; r0 = r2 + r1 << 3, to oznacza r0 = r2 + 8 * r1
ADD r0, r2, r1, LSR #3    ; r0 = r2 + r1 >> 3, to oznacza r0 = r2 + r1 / 8
```

6.4. Instrukcje arytmetyczne

6.4.1. Instrukcje dodawania i odejmowania

Większość tych instrukcji ma dwa operandy źródłowe, a 32-bitowy wynik jest umieszczany w rejestrze przeznaczenia. Pierwszy operand jest rejestrem, drugi może być rejestrem, stałą bezpośrednią lub rejestrem połączonym z układem przesuwającym *Barrel Shifter*. Przykłady działań z trzema rejestrami

```
SUB r3, r2, r1    ; r3 = r2 - r1
SUC r3, r2, r1    ; r3 = r2 - r1 + Carry -1
```



Rys. 6.3: Operacje przesunięcia *shift* i rotacji *rotate*

Tabela 6.1: Instrukcje arytmetyczne z wynikiem 32-bitowym

`ADD {Rd,} Rn, Op2`	Add, $Rd \leftarrow Rn + Op2$
`ADC {Rd,} Rn, Op2`	Add with Carry, $Rd \leftarrow Rn + Op2 + Carry$
`SUB {Rd,} Rn, Op2`	Subtract, $Rd \leftarrow Rn - Op2$
`SBC {Rd,} Rn, Op2`	Subtract with Carry, $Rd \leftarrow Rn + Op2 + Carry - 1$
`RSB {Rd,} Rn, Op2`	Reverse subtract, $Rd \leftarrow Op2 - Rn$
`MUL {Rd,} Rn, Rm`	Multiply, $Rd \leftarrow (Rn * Rm)[31:0]$
`MLA Rd, Rn, Rm, Ra`	Multiply with accumulate, $Rd \leftarrow (Ra + (Rn * Rm))[31:0]$
`MLS Rd, Rn, Rm, Ra`	Multiply and subtract, $Rd \leftarrow (Ra - (Rn * Rm))[31:0]$
`SDIV {Rd,} Rn, Rm`	Signed divide, $Rd \leftarrow Rn/Rm$
`UDIV {Rd,} Rn, Rm`	Unsigned divide, $Rd \leftarrow Rn/Rm$
`SSAT Rd, #n, Rm{,shift #s}`	Signed saturate
`USAT Rd, #n, Rm{,shift #s}`	Unsigned saturate

```
RSB r3, r2, r1     ; r3 = r1 - r2
```

Przykłady z operandem natychmiastowym

```
SUB r3, r2, #456     ; r3 = r2 - 456
RSB r3, r2, #789     ; r3 = 789 - r2
```

Przykłady instrukcji wykorzystujących *Barrel Shifter*

```
RSB r0, r0, r0, LSL #5     ; r0 = r0 << 5 - r0 = 31 * r0
ADD r0, r0, r0, LSL #3     ; r0 = r0 + r0 << 3 = 9 * r0
```

Jeżeli instrukcja ma trzy operandy, to drugi operand nie może być liczbą stałą (nie dotyczy to instrukcji `SSAT` i `USAT`), na przykład instrukcja poniżej ma błąd składni, a kolejna poprawnie wykonuje zamierzone odejmowanie

```
SUB r0, #1, r3     ; niedopuszczalna składnia → syntax error
RSB r0, r3, #1     ; r0 = 1 - r3
```

Kolejny przykład to dodawanie dwóch liczb całkowitych 64-bitowych (ze znakiem lub bez). Pierwszy składnik sumy przechowywany jest w parze rejestrów (r1:r0), bardziej znacząca część w r1 i mniej znacząca w r0. Drugi składnik analogicznie w (r3:r2). Wynik umieścimy w (r5:r4).

```
; dodawanie 64-bitowe: C(r5:r4) = B(r3:r2) + A(r1:r0)
ADDS r4, r2, r0     ; C[31:0] = B[31:0] + A[31:0], modyfikuj Carry
ADC  r5, r3, r1     ; C[63:32] = B[63:32] + A[63:32] + Carry
```

6.4.2. Instrukcje krótkiego dzielenia i mnożenia

Wynik mnożenia dwóch rejestrów może mieć więcej niż 32 bity. Ale rejestr w którym zostanie umieszczony ma tylko 32 bity, dlatego zapisywane są bity wyniku [31:0], a starsze [63:32] są odrzucane. Przykłady

```
MUL  r6, r4, r2     ; r6 = [31:0](r4 * r2), mnożenie liczb ze znakiem
UMUL r6, r4, r2     ; r6 = [31:0](r4 * r2), mnożenie liczb bez znaku
MLA  r6, r4, r2, r1 ; r6 = [31:0](r4 * r2) + r1
MLS  r6, r4, r2, r1 ; r6 = [31:0](r4 * r2) - r1
SDIV r3, r2, r1     ; r3 = r2 / r1, liczby ze znakiem
UDIV r3, r2, r1     ; r3 = r2 / r1, liczby bez znaku
```

6.4.3. Instrukcje długiego mnożenia

Istnieją także instrukcje które zapamiętują cały, 64-bitowy wynik mnożenia dwóch liczb 32-bitowych (tabela 6.2). Dwa rejestry są potrzebne do przechowania liczby 64-bitowej. Jeden (`RdHi`) zawiera bardziej znaczące bity [63:32], drugi (`RdLo`) - mniej znaczące [31:0]. Instrukcje `UMULL` i `UMLAL` zakładają, że oba operandy (i wynik) są liczbami całkowitymi bez znaku. Instrukcje `SMULL` i `SMLAL` zakładają, że oba operandy (i wynik) są liczbami całkowitymi ze znakiem. Przykłady:

Tabela 6.2: Instrukcje arytmetyczne z wynikiem 64-bitowym

`UMULL RdLo,RdHi,Rn,Rm`	Unsigned Long Multiply, $(RdHi, RdLo) \leftarrow unsigned(Rn * Rm)$
`SMULL RdLo,RdHi,Rn,Rm`	Signed Long Multiply, $(RdHi, RdLo) \leftarrow signed(Rn * Rm)$
`UMLAL RdLo,RdHi,Rn,Rm`	Unsigned Long Multiply with Accumulate, $(RdHi, RdLo) \leftarrow unsigned((RdHi, RdLo) + Rn * Rm)$
`SMLAL RdLo,RdHi,Rn,Rm`	Signed Long Multiply with Accumulate, $(RdHi, RdLo) \leftarrow signed((RdHi, RdLo) + Rn * Rm)$

```
UMULL r3, r4, r0, r1    ; r4:r3 = r0 * r1, r4 = bardziej znaczące bity
SMULL r3, r4, r0, r1    ; r4:r3 = r0 * r1, r3 = mniej znaczące bity
UMLAL r3, r4, r0, r1    ; r4:r3 = r4:r3 + r0 * r1
SMLAL r3, r4, r0, r1    ; r4:r3 = r4:r3 + r0 * r1
```

6.5. Instrukcje nasycenia

Instrukcje te pozwalają ograniczyć wejście (z nasyceniem) do zadanego zakresu (ze znakiem lub bez). Gdy wartość wejściowa przekracza zakres, to jest ona ograniczana do wartości maksymalnej /minimalnej. Jeżeli wartość mieści się w zakresie, to nie jest zmieniana. Instrukcje nasycenia mają dwa operandy źródłowe: jeden to operand natychmiastowy (liczba określająca zakres), drugi to rejestr z wartością wejścia.

Instrukcja **SSAT** ogranicza liczbę ze znakiem do zakresu $-2^{n-1} \leqslant x \leqslant 2^{n-1} - 1$

$$SSAT(x) = \begin{cases} 2^{n-1} - 1 & gdy \quad x > 2^{n-1} - 1 \\ x & gdy \quad -2^{n-1} < x < 2^{n-1} - 1 \\ -2^{n-1} & gdy \quad x < -2^{n-1} \end{cases}$$

Instrukcja **USAT** ogranicza liczbę bez znaku do zakresu $0 \leqslant x \leqslant 2^{n-1} - 1$

$$USAT(x) = \begin{cases} 2^{n} - 1 & gdy \quad x > 2^{n} - 1 \\ x & gdy \quad 0 < x < 2^{n} - 1 \end{cases}$$

Poniżej dwa przykłady, gdy $n = 11$

```
SSAT r2, #11, r1    ; zakres wyjścia: -2^10 ⩽ r2 ⩽ 2^10 - 1
USAT r2, #11, r1    ; zakres wyjścia: 0 ⩽ r2 ⩽ 2^11 - 1
```

6.6. Dwuargumentowe działania logiczne

Działania logiczne na bitach traktują zawartość rejestru jako zbiór bitów, a nie jak liczbę całkowitą. Działania logiczne dotyczą pojedynczych bitów. Możemy, na przykład wyzerować /ustawić lub odwrócić określony bit w rejestrze, nie zmieniając pozostałych bitów tego rejestru.

Negacja bitu NOT jest działaniem jednoargumentowym. Są trzy podstawowe działania dwuargumentowe na bitach: AND, OR, XOR oraz dwa ważne złożenia NAND i NOR. Definicje pokazane są na rysunku 2.4.

Tabela 6.3: Instrukcje logiczne w Cortex-M

`AND {Rd,} Rn, Op2`	Bitwise logic AND, $Rd \leftarrow Rn \ \& \ Operand2$
`ORR {Rd,} Rn, Op2`	Bitwise logic OR, $Rd \leftarrow Rn \mid Operand2$
`EOR {Rd,} Rn, Op2`	Bitwise logic exclusive OR, $Rd \leftarrow Rn \ \wedge \ Operand2$
`ORN {Rd,} Rn, Op2`	Bitwise logic OR NOT, $Rd \leftarrow Rn \mid NOT(Operand2)$
`BIC {Rd,} Rn, Op2`	Bit clear, $Rd \leftarrow Rn \ \& \ NOT(Operand2)$
`BFC Rd, #lsb, #width`	Bit field clear, $Rd[(width + lsb - 1) : lsb] \leftarrow 0$
`BFI Rd,Rn,#lsb,#width`	Bit field insert, $Rd[(width+lsb-1) : lsb] \leftarrow Rn[(width-1) : 0]$
`MVN Rd, Op2`	Logically negate all bits, $Rd \leftarrow NOT(Operand2)$

Poniżej podane są przykłady działania instrukcji logicznych, rejestry skrócono do czterech bitów, `r1=0xA, r2=0xC`

```
                r1  1 0 1 0                                  r1  1 0 1 0
                r2  1 1 0 0                                  r2  1 1 0 0
 AND r0,r1,r2   r0  1 0 0 0                   ORR r0,r1,r2    r0  1 1 1 0


                                                             r1  1 0 1 0
                r1  1 0 1 0                                  r2  1 1 0 0
                r2  1 1 0 0                             NOT  r2  0 0 1 1
 EOR r0,r1,r2   r0  0 1 1 0                   ORN r0,r1,r2    r0  1 0 1 1


                                                              r1  1 0 1 0
                                                              r2  1 1 0 0
             r2  1 1 0 0                                 NOT  r2  0 0 1 1
 MVN r0, r2  r0  0 0 1 1                     BIC r0,r1,r2     r0  0 0 1 0
```

6.6.1. Maski bitowe

Maska bitowa służy do manipulowania wybranymi bitami w rejestrze. Ma taką samą długość, jak ciąg bitów który nas interesuje. W masce umieszczamy jedynki na pozycjach, które mają być testowane /zmieniane i zera na pozycjach, które mają być ignorowane. W poniższych przykładach rejestr `r0` zawiera ciąg bitów i w tym ciągu interesuje nas bit `b5`

`ANDS r1, r0, #(1<<5)`	**Test bitu**: flaga Z będzie ustawiona, gdy `b5` w `r0` jest zerem i wyzerowana gdy `b5` jest jedynką
`ORR r0, r0, #(1<<5)`	**Ustawienie bitu**: `OR` bitu z jedynką ustawia ten bit, z zerem - nie zmienia. Zostaną ustawione wszystkie bity dla których w masce są jedynki
`BIC r0, #(1<<5)`	**Zerowanie bitu**: `AND` bitu z zerem zeruje ten bit, z jedynką - nie zmienia. Zostaną wyzerowane wszystkie bity dla których w masce są jedynki
`EOR r0, r0, #(1<<5)`	**Odwrócenie bitu**: `XOR` bitu z jedynką neguje ten bit, z zerem - nie zmienia. Zostaną zanegowane (odwrócone) wszystkie bity dla których w masce są jedynki

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
Reserved															
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Res.	USART 1EN	Res.	SPI1 EN	TIM1 EN	ADC2 EN	ADC1 EN	Reserved		IOPE EN	IOPD EN	IOPC EN	IOPB EN	IOPA EN	Res.	AFIO EN
	rw		rw	rw	rw	rw			rw	rw	rw	rw	rw		rw

Rys. 6.4: Rejestr RCC_APB2ENR

6.6.2. Definiowanie masek za pomocą dyrektywy EQU

W dokumencie RM0008 (tabela 4.2) opisany jest rejestr *APB2 peripheral clock enable register* RCC_APB2ENR. Znaczenie bitów pokazane jest na rysunku 6.4. Po RESET wszystkie bity tego rejestru są wyzerowane. Ustawienie bitu `b2` (IOPA_EN) włącza taktowanie portu GPIOA, bitu `b3` włącza taktowanie GPIOB i tak dalej aż do bitu `b6` → GPIOE. Napiszemy teraz krótki program asemblerowy, który włączy taktowanie portów GPIOA, GPIOB i GPIOC.

```
RCC_BASE     EQU 0x40021000    ; początek rejestrów sterujących RCC
RCC_APB2ENR  EQU 0x18          ; adres w bloku rejestrów sterujących RCC
RCC_GPIOAEN  EQU 0x00000004    ; bit b2: IOPA_EN (maska bitowa)
RCC_GPIOBEN  EQU 0x00000008    ; bit b3: IOPB_EN (maska bitowa)
RCC_GPIOCEN  EQU 0x00000010    ; bit b4: IOPC_EN (maska bitowa)

LDR r7, =RCC_BASE              ; r7 = adres bloku rejestrów sterujących RCC
LDR r1, [r7,#RCC_APB2ENR]      ; odczytaj rejestr RCC_APB2ENR do r1
ORR r1, r1, #RCC_GPIOAEN       ; ustaw bit b2
ORR r1, r1, #RCC_GPIOBEN       ; ustaw bit b3
ORR r1, r1, #RCC_GPIOCEN       ; ustaw bit b4
STR r1, [r7,#RCC_APB2ENR]      ; załącz taktowanie portów GPIOA/B/C
```

6.6.3. Operatory logiczne w C

Operator języka C: iloczyn logiczny `A&&B` traktuje ciąg bitów jak liczbę, której nadaje wartość logiczną `FALSE` gdy liczba jest zerem i `TRUE`, gdy jest różna od zera. Operator iloczyn bitowy `A&B` wykonuje działanie na bitach. Wynikiem działania (`0x10 & 0x01`) jest `0x00`, gdyż jest to działanie na bitach. Natomiast wynikiem (`0x10 && 0x01`) jest `0x01`, czyli `TRUE && TRUE` = `TRUE` .

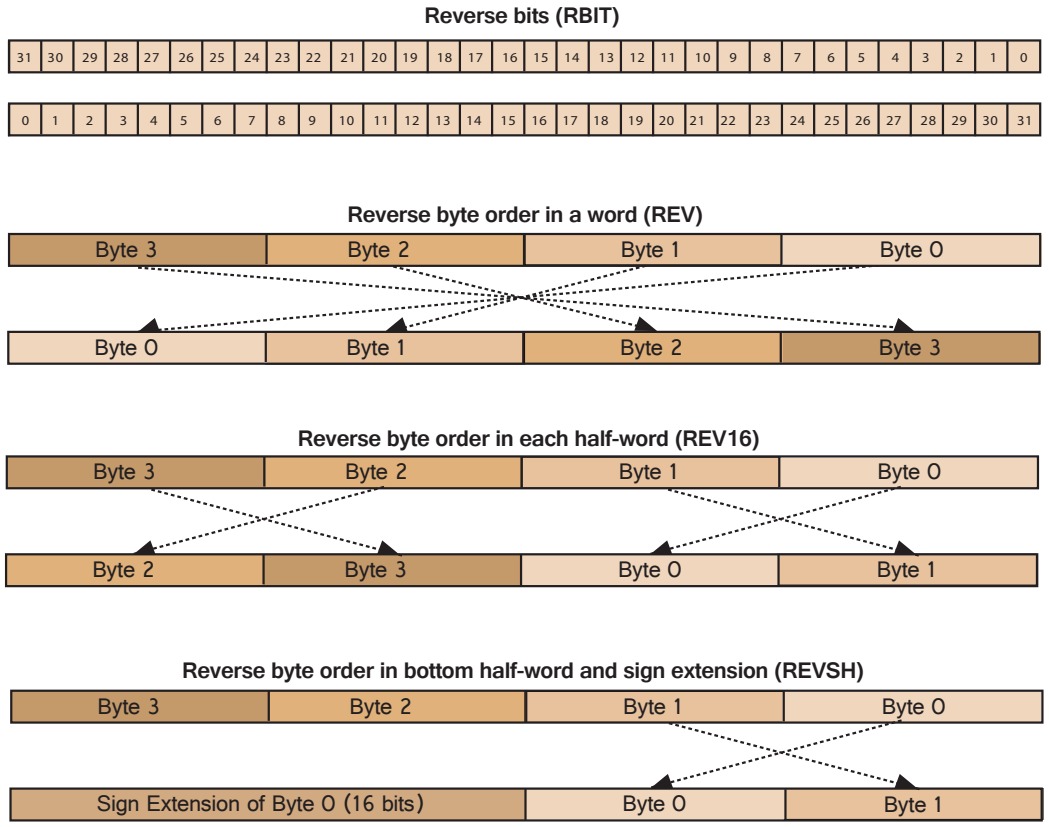
Podobnie działają operatory: suma logiczna `A||B` i suma bitowa `A|B` oraz negacja logiczna `!A` i negacja bitowa `~A`. Na przykład wartością wyrażenia (`~0x01`) jest `0xFFFFFFFE`, natomiast wartością (`!0x01`) jest `0x00` czyli `FALSE`. Operatory języka C są wymienione w dodatku B.

6.7. Kolejność bitów i bajtów

Instrukcje odwracające kolejność bajtów są potrzebne, szczególnie gdy wymieniamy dane pomiędzy dwoma systemami z *Little Endian* i *Big Endian*, patrz rozdział 7.2. Przykładem odwracania kolejności bitów jest obliczanie szybkiej transformaty Fouriera FFT. Przemieszanie danych wejściowych FFT jest znane jako odwrócenie bitowe (*bit reversal*),

Tabela 6.4: Instrukcje zmiany kolejności bitów i bajtów

`RBIT Rd, Rn`	Odwrócenie kolejności bitów w rejestrze
`REV Rd, Rn`	Odwrócenie kolejności bajtów w rejestrze
`REV16 Rd, Rn`	Odwrócenie kolejności bajtów w każdej połówce rejestru
`REVSH Rd, Rn`	Odwrócenie kolejności bajtów w młodszej połówce rejestru i arytmetyczne (w kodzie U2) uzupełnienie starszej połówki



Rys. 6.5: Zmiany kolejności bitów i bajtów

ponieważ nowe uporządkowanie danych uzyskuje się przez odwrócenie porządku bitów w binarnej reprezentacji indeksu. Poniżej kilka przykładów

```
0 0 1 0'0 0 1 1'0 1 0 0'0 1 0 1'0 1 1 0'0 1 1 1'1 0 0 0'1 0 0 1  = 0x23456789
1 0 0 1'0 0 0 1'1 1 1 0'0 1 1 0'1 0 1 0'0 0 1 0'1 1 0 0'0 1 0 0  = 0x91E6A2C4
```

```
LDR   r0, =0x23456789 ; r0 = 0x23456789
RBIT  r1, r0          ; odwrócenie bitowe,  r1 = 0x91E6A2C4
REV   r1, r0          ; odwrócenie bajtowe, r1 = 0x89674523
REV16 r1, r0          ; odwrócenie bajtowe w połówkach, r1 = 0x45238967
REVSH r1, r0          ; odwrócenie z uzupełnieniem, r1 = 0xFFFF8967
```

Tabela 6.5: Instrukcje wykonujące *sign extension* oraz *zero extension*

`SXTB {Rd,} Rm {,ROR #n}`	Sign extend a byte
`SXTH {Rd,} Rm {,ROR #n}`	Sign extend a halfword
`UXTB {Rd,} Rm {,ROR #n}`	Zero extend a byte
`UXTH {Rd,} Rm {,ROR #n}`	Zero extend a halfword

6.8. Zmiana typu liczby całkowitej

Gdy stosujemy kod uzupełnień do dwóch, to ten sam arytmometr może wykonywać operacje na liczbach całkowitych ze znakiem i bez znaku. Dlatego powszechnie stosowane jest kodowanie U2. Jeżeli reprezentację liczby ze znakiem rozszerzamy do większej ilości bitów, musimy powielić bit znaku, aby zachować znak i wartość liczby. Jest to *sign extension*. W przypadku liczb bez znaku, musimy dopisać zera. Jest to *zero extension*.

Poniżej jest przykład wydłużania liczb całkowitych. Plik nagłówkowy *stdint.h* definiuje typy zmiennych całkowitych ze znakiem `int8_t, int16_t, int32_t` oraz bez znaku `uint8_t, uint16_t, uint32_t`, odpowiednio 8 -, 16 - i 32 - bitowe.

```
#include <stdint.h>

int8_t  a = -1;        // a signed 8-bit integer,  a = 0xFF
int16_t b = -2;        // a signed 16-bit integer, a = 0xFFFE
int32_t c;             // a signed 32-bit integer
c = a;                 // sign extension, c = 0xFFFFFFFF
c = b;                 // sign extension, c = 0xFFFFFFFE

uint8_t  d = 255;      // an unsigned 8-bit integer, d = 0xFF
uint32_t e;            // an usigned 32-bit integer
e = d;                 // zero extension, e = 0x000000FF
```

Przykłady w asemblerze

```
LDR r0, =0x456789AB  ; r0 = 0x456789AB
SXTB r1, r0          ; r1 = 0xFFFFFFAB, sign extend a byte
SXTH r1, r0          ; r1 = 0xFFFF89AB, sign extend a halfword
UXTB r1, r0          ; r1 = 0x000000AB, zero extend a byte
UXTH r1, r0          ; r1 = 0x000089AB, zero extend a halfword
```

6.9. Instrukcje porównań

Są cztery instrukcje porównywania danych (tablica 6.6).

Tabela 6.6: Instrukcje porównań

`CMP Rn, Op2`	Compare	modyfikuj flagi do	`Rn - Op2`
`CMN Rn, Op2`	Compare Negative	modyfikuj flagi do	`Rn + Op2`
`TST Rn, Op2`	Test	modyfikuj flagi do	`Rn AND Op2`
`TEQ Rn, Op2`	Test equivalence	modyfikuj flagi do	`Rn XOR Op2`

- **CMP** *Compare*. Instrukcja odejmuje drugi operand od zawartości rejestru `Rn`. Procesor modyfikuje flagi N, Z, C i V zgodnie z wynikiem odejmowania, a sam wynik odrzuca. To znaczy, że żaden rejestr nie jest zmieniany.
- **CMN** *Compare Negative*. Instrukcja dodaje drugi operand od zawartości rejestru `Rn`. Procesor modyfikuje flagi N, Z, C i V zgodnie z wynikiem odejmowania, a sam wynik odrzuca. To znaczy, że żaden rejestr nie jest zmieniany.
- **TST** *Test*. Instrukcja wykonuje działanie logiczne dwuargumentowe AND wartości drugiego operandu oraz rejestru `Rn`. Procesor modyfikuje flagi N i Z. Jeżeli drugi operand używa *Barrel Shifter*, możliwa jest zmiana stanu flagi C. Sam wynik działania AND jest odrzucany.
- **TEQ** *Test equivalence*. Instrukcja wykonuje działanie logiczne exclusive OR wartości drugiego operandu oraz rejestru `Rn`. Procesor modyfikuje flagi N i Z. Jeżeli drugi operand używa *Barrel Shifter*, możliwa jest zmiana stanu flagi C. Sam wynik działania XOR jest odrzucany.

Instrukcja `TEQ` ustawi flagę *Zero* Z=1 gdy oba operandy są sobie równe i wyzeruje Z=0, gdy są różne. Instrukcja `TST` ustawia Z=1, gdy brak pokrywających się jedynek. Jeżeli drugi operand jest maską bitową, to `TST` sprawdza, czy wszystkie bity wskazywane przez maskę są zerami, wtedy Z=1. Poniżej kilka przykładów porównywania danych.

```
CMP r0, #3          ; r0 - 3, porównaj r0 z liczbą 3
CMN r0, #3          ; r0 - (-3) → r0 + 3, porównaj r0 z liczbą -3
CMP r0, r1          ; r0 - r1, porównaj r0 z r1
TEQ r0, #'?'        ; czy w r0 jest kod ASCII znaku '?' (0x3F)

MOVS r1, #1         ; r1 = 0x00000001
LSL  r1, #31        ; r1 = 0x80000000
TST  r0, r1         ; czy w r0 jest liczba ujemna
```

6.10. Przesyłanie danych pomiędzy rejestrami

Instrukcje przesyłania danych pomiędzy rejestrami dzielą się na:

- przesyłanie pomiędzy rejestrami ogólnego przeznaczenia `r0 - r12` oraz `r13` (SP), `r14` (LR) i `r15` (PC) - instrukcje `MOV` i `MVN`,
- przesyłanie pomiędzy rejestrami ogólnego i specjalnego przeznaczenia - instrukcje `MRS` i `MSR`

Rejestry specjalne to: APSR, IPSR, EPSR, IEPSR, IAPSR, EAPSR, PSR, MSP, PSP, PRIMASK, BASEPRI, BASEPRI_MAX, FAULTMASK oraz CONTROL.

Tabela 6.7: Instrukcje przesyłania danych pomiędzy rejestrami

`MOV Rd, Op2`	Rd ← Operand2
`MVN Rd, Op2`	Rd ← NOT(Operand2)
`MRS Rd, spec_reg`	Rd ← spec_reg
`MSR spec_reg, Rm`	spec_reg ← Rm

Instrukcje `MOV` i `MVN` mogą zapisać stałą natychmiastową do rejestru, jak będzie to pokazane w następnym rozdziale. Poniżej kilka przykładów

```
MOV r0, r1            ; przepisz r1 do r0
MOV r0, r1, LSL #3    ; r0 = r1 << 3
MOV r0, PC            ; przepisz licznik programu PC do r0
MOV r0, SP            ; przepisz wskaźnik stosu SP do r0
```

Kolejne instrukcje przepisują zawartość rejestrów specjalnych do rejestrów ogólnych

```
MRS r1, APSR          ; przepisz APSR do r1
MRS r2, PSR           ; przepisz xPSR do r2
```

Zapis rejestrów specjalnego przeznaczenia

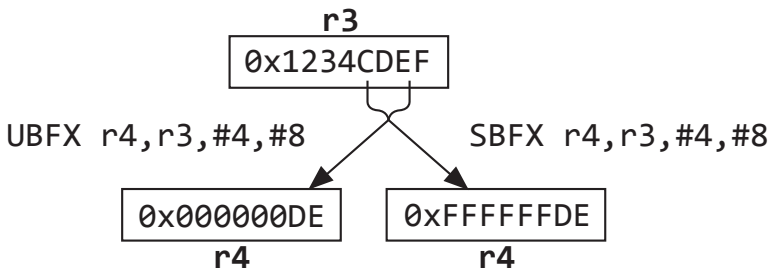
```
MRS APSR, r0          ; zapisz flagi statusów
MRS BASEPRI, r0       ; blokada obsługi przerwań o priorytecie ⩽ r0
```

6.11. Przepisz określony fragment rejestru

Tabela 6.8: Instrukcje przepisania fragmentu rejestru

`SBFX Rd, Rn, #lsb, #width`	Signed Bit Field Extract
`UBFX Rd, Rn, #lsb, #width`	Unsigned Bit Field Extract

Parametr `#lsb` (zakres od 0 do 31) określa numer bitu od którego zaczyna się interesujący nas fragment ciągu bitów w rejestrze `Rn`. Parametr `#width` (zakres od 1 do 32) to ilość bitów. Bity "wycięte" z rejestru źródłowego `Rn` są przepisywane na najmniej znaczące pozycje rejestru docelowego `Rd`, czyli zaczynając od `b0`. Instrukcja `UBFX`, niezapisane najstarsze bity w `Rd` uzupełnia zerami. Instrukcja `SBFX`, niezapisane najstarsze bity w `Rd` uzupełnia znakiem, czyli najstarszym bitem wyciętego pola. Rysunek 6.6 pokazuje przykład działania obu instrukcji.



Rys. 6.6: Przykład działania instrukcji `SBFX` i `UBFX`

Rozdział 7

Instrukcje przesyłania danych

Instrukcje "ładuj" *Load* wpisują do rejestru określoną wartość. Może to być stała natychmiastowa umieszczona w kodzie programu, lub zawartość komórki pamięci. Instrukcje "składuj" *Store* przepisują zawartość rejestru do komórki pamięci.

7.1. Wpisanie danej do rejestru

Wiele instrukcji asemblerowych używa stałych natychmiastowych *immediate numbers*, które są umieszczane w kodzie instrukcji lub jako liczby stałe w kodzie programu. Jednym z zastosowań stałych natychmiastowych jest wpisanie do rejestru określonej, stałej wartości.

Tabela 7.1: Instrukcje wpisania stałej do rejestru

`MOVS Rd, #<immed_8>`	Zapisz bajt (0-255) do rejestru, ustaw flagi
`MOV Rd, #<immed_12>`	Zapisz stałą 12-bitową do rejestru
`MVN Rd, #<immed_12>`	Zapisz do rejestru negację stałej (bit po bicie)
`MOVT Rd, #<immed_16>`	Do bardziej znaczącej połowy rejestru [31:16] zapisz stałą 16-bitową, Mniej znacząca połowa [15:0] pozostanie niezmieniona.
`MOVW Rd, #<immed_16>`	Do mniej znaczącej połowy rejestru [15:0] zapisz stałą 16-bitową, wyzeruj bardziej znaczącą połowę [31:16]
`LDR Rt, =#<immed_12>`	Zamieniane na instrukcje `MOV` albo `MVN`
`LDR Rt, =#<immed_32>`	Pseudoinstrukcja

7.1.1. Instrukcje przesyłania danych `MOV` i `MVN`

Literały numeryczne są to ciągi znaków lub pojedynczy znak *in quotes* ('w psich uszach'), zamieniane na liczbę całkowitą

- `1234567890` liczby dziesiętne
- `0x0123456789ABCDEF` liczby hexadecymalne
- *n_cyfry* liczby o podstawie *n*, np. `2_0101010101` lub `8_01234567`
- 'znak', np. `'A'`

Stałe natychmiastowe są poprzedzone znakiem `#`. W poniższych przykładach do rejestru `r0` będzie wpisana ta sama liczba całkowita, `MOVS` to instrukcja 16-bitowa, `MOV` jest 32-bitowa.

```
MOVS r0, #122          ; r0 ← 122 dziesiętnie
MOVS r0, #0x7A         ; r0 ← 0x7A Hexa
MOVS r0, #2_01111010   ; r0 ← 0111 1010 binarnie
MOVS r0, #8_172        ; r0 ← 172 oktalnie
MOVS r0, #'z'          ; r0 ← 'z' kod ASCII
```

Wpiszemy 32-bitową stałą do rejestru

```
MOVW r1, #0xFFFF
MOVT r1, #0xFFFF     ; r1 = 0xFFFFFFFF
```

Jedynka na dowolnej pozycji

```
MOVS r1, #1          ; r1 = 0x00000001
LSL  r1, #31         ; r1 = 0x80000000
```

Zero na dowolnej pozycji

```
MVN  r1, #1          ; r1 = 0xFFFFFFFE
ROR  r1, #1          ; r1 = 0x7FFFFFFF
```

7.1.2. Pseudoinstrukcje LDR i ADR

W pamięci RAM mamy zadeklarowane dwie tablice `array1 = 0x2000 0004` oraz `array2 = 0x2000 000C`. Poniższy podprogram pobiera z pierwszej tablicy słowo 32-bitowe, dodaje do niego stałą `0x11325374` i wynik zapisuje do drugiej tablicy. Kolorem *purple* zaznaczona jest postać źródłowa programu asemblerowego, kolor czarny to wynik deasemblacji odpowiedniego fragmentu pamięci programu.

```
    75:                                PUSH  {r1-r3,lr}
0x0800058C  B50E        PUSH        {r1-r3,lr}
    76:                                LDR  r0,=array1     ; pseudoinstrukcja
0x0800058E  4803        LDR         r0,[pc,#12]     ; @0x0800059C
    77:                                LDR  r1,=array2     ; pseudoinstrukcja
0x08000590  4903        LDR         r1,[pc,#12]     ; @0x080005A0
    78:                                LDR  r2,[r0] ; instrukcja maszynowa
0x08000592  6802        LDR         r2,[r0,#0]
    79:                                LDR  r3,=0x11325374 ; pseudoinstrukcja
0x08000594  4B03        LDR         r3,[pc,#12]     ; @0x080005A4
    80:                                ADD  r2,r3
0x08000596  441A        ADD         r2,r2,r3
    81:                                STR  r2,[r1]
0x08000598  600A        STR         r2,[r1,#0]
    82:                                POP  {r1-r3,pc}
0x0800059A  BD0E        POP         {r1-r3,pc}

0x0800059C  0004        DCW         0x0004
0x0800059E  2000        DCW         0x2000
0x080005A0  000C        DCW         0x2000
0x080005A2  2000        DCW         0x000C
0x080005A4  5374        DCW         0x5374
0x080005A6  1132        DCW         0x1132
```

Zawartość pamięci przed i po wykonaniu podprogramu

0x20000000	00	00	00	00	04	03	02	01	00	00	00	00	00	00	00	00	00	00	00	00	00	00
0x20000000	00	00	00	00	04	03	02	01	00	00	00	00	78	56	34	12	00	00	00	00	00	

W powyższym podprogramie użyto instrukcji `LDR` ładowania rejestru z pamięci. Instrukcja ta może być zwykłą instrukcją maszynową albo pseudoinstrukcją w zależności

od zastosowanych operandów. Jeżeli operandami są adresy lub liczby 32-bitowe, asembler umieszcza te adresy i liczby w postaci tablicy **pod** kodem podprogramu i stosuje adresowanie względem aktualnej zawartości licznika progamu PC. Uzyskujemy kod relokowalny.

Kolejną pseudoinstrukcją jest `ADR`, która pozwala zapisać do rejestru adres wybranej etykiety. Ta pseudoinstrukcja także jest tłumaczona na instrukcję z adresowaniem względem licznika programu PC

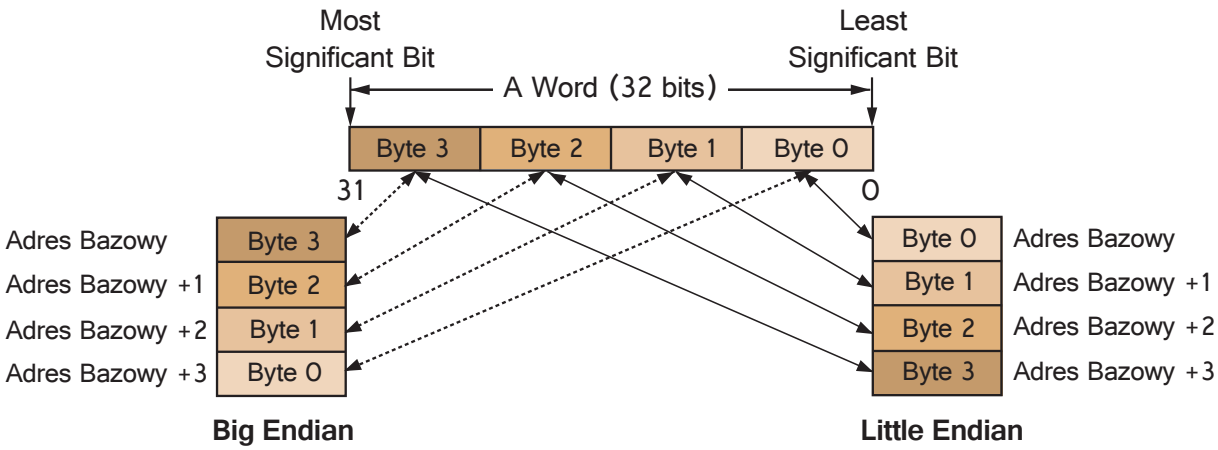
```
  76: loop                          LDR  r4,=0xFFF
0x0800058E    F64074FF    MOVW    r4,#0xFFF
  77:                               ADR  r4, label
0x08000592    A401        ADR     r4,{pc}+2     ; @0x08000598
  78:                               LDR  r1,=-1
0x08000594    F04F31FF    MOV     r1,#0xFFFFFFFF
  79: label                         MOV  r0,r1
0x08000598    4608        MOV     r0,r1
  80:                               ADR  r5, loop
0x0800059A    F2AF050E    ADR.W   r5,{pc}-0x0C     ; @0x0800058E
```

Podczas gdy `ADR` umożliwia tylko załadowanie adresu etykiety do rejestru, pseudoinstrukcja `LDR` jest bardziej elastyczna. To jak asembler przetłumaczy `LDR` na kody maszynowe zależy od operandów

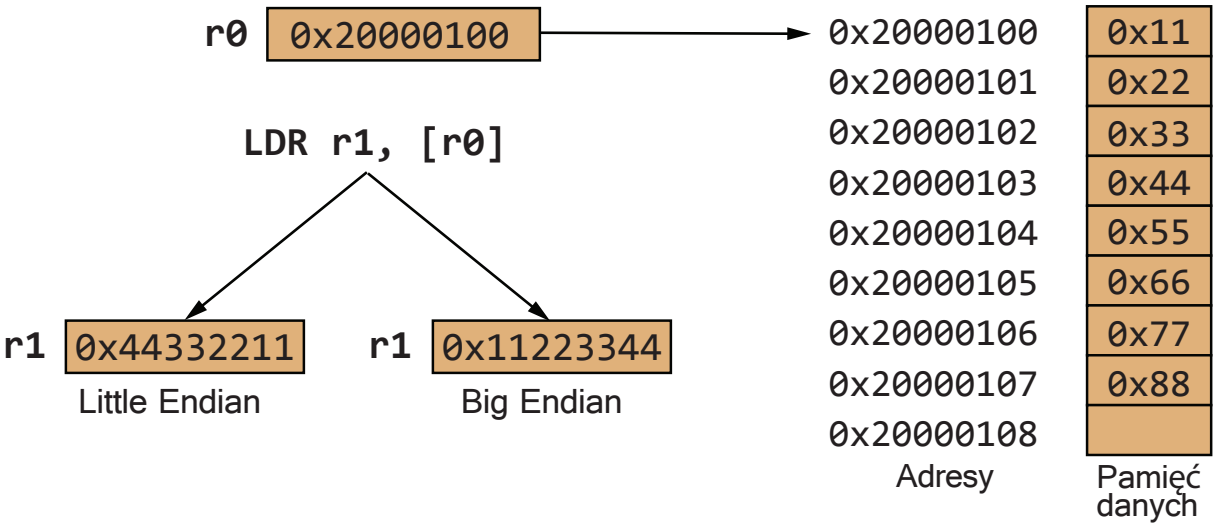
- Jeżeli operand natychmiastowy mieści się w 12 bitach (na przykład `0xFFF`), asembler zamieni pseudoinstrukcję `LDR` na `MOV` albo `MVN`
- Jeżeli operand przekracza 12 bitów, w szczególności gdy jest to adres, asembler utworzy tablicę stałych pod kodem bieżącego programu i użyje adresowania względem aktualnej wartości licznika programu PC

7.2. Big Endian oraz Little Endian

Endian określa kolejność bajtów w pamięci, jak pokazano na rysunku 7.1



Rys. 7.1: Porównanie *Big Endian* z *Little Endian*



Rys. 7.2: Porównanie *Big Endian* z *Little Endian*

- *Little Endian* oznacza, że najmniej znaczący bajt słowa jest pamiętany jako pierwszy, a najbardziej znaczący jako ostatni, czyli zaczynamy od "małego końca",
- *Big Endian* oznacza, że jako pierwszy pamiętany jest najbardziej znaczący bajt, czyli zaczynamy od "dużego końca".

Mikroprocesor STM32F103 pracuje w trybie *Little Endian* i nie można tego zmienić. Komunikację z systemem zewnętrznym typu *Big Endian* ułatwia instrukcja `REV`, która odwraca porządek bajtów w słowie.

Ogólnie architektura ARMv7-M przewiduje, że w trakcie RESET odczytywany jest stan określonego pinu procesora, który decyduje o kolejności bajtów. Jeżeli wybrany zostanie *Big Endian*, to dotyczy on wyłącznie danych, gdyż instrukcje są zawsze *Little Endian*. Bit `b15 ENDIANESS` w rejestrze `SCB_AIRCR` (tylko do odczytu) informuje o kolejności używanej w procesorze. W przypadku ST32F103 wartość bitu zawsze jest zerowa = *Little Endian*, gdyż ten procesor nie jest wyposażony w pin decydujący o *Endianess*.

7.3. Dostęp do danych w pamięci

Instrukcja `LDR` *Load* pobiera liczbę z określonego miejsca pamięci i wpisuje ją do wskazanego rejestru. Instrukcja `STR` *Store* wykonuje czynność odwrotną, czyli zapisuje zawartość rejestru do pamięci. Jeżeli program ma dostęp do określonego miejsca pamięci to adres tego miejsca musi być w rejestrze. W przykładzie poniżej, rejestr `r0` zawiera adres. Program ładuje słowo 32-bitowe z pamięci do rejestru , dodaje `0x02` i wynik zapisuje do pamięci. Jest to metoda *Load - Modify - Store*.

```
LDR  r0,=array2     ; r0 = 0x2000000C
LDR  r1,[r0]        ; r1 = 0x12345678
ADD  r1,#2          ; r1 = 0x1234567A
STR  r1,[r0]        ; r1 → 0x2000000C
```

7.4. Adresowanie pamięci

7.4.1. *Pre-index*, *Post-index* oraz *Pre-index with Update*

Każdy z trybów adresowania pokazanych w tablicy 7.2 ma adres bazowy i bajt przesunięcia *byte offset*.

Tabela 7.2: Trzy tryby adresowania pamięci

Tryb adresowania	Przykład	Operacja
Pre - index	`LDR r1, [r0, #4]`	`r1 ← memory[r0+4]`, `r0 pozostaje niezmienione`
Pre - index with update	`LDR r1, [r0, #4]!`	`r1 ← memory[r0+4]`, `r0 ← r0 + 4`
Post - index	`LDR r1, [r0], #4`	`r1 ← memory[r0]`, `r0 ← r0 + 4`

- W trybie **pre-index** adres w pamięci jest sumą adresu bazowego w rejestrze i bajtu przesunięcia. Adres bazowy pozostaje niezmienny.
- W trybie **pre-index with update** adres w pamięci jest sumą adresu bazowego w rejestrze i bajtu przesunięcia. Po wykonaniu operacji przesłania danych z pamięci, adres bazowy jest modyfikowany o bajt przesunięcia.
- W trybie **post-index** przesłanie danych następuje z adresu bazowego. Po operacji przesłania adres bazowy jest modyfikowany o bajt przesunięcia.

Bajt przesunięcia może być zarówno liczbą dodatnią jak i ujemną. Przedstawione tryby adresowania w równym stopniu dotyczą instrukcji `STR`.

7.4.2. Instrukcje `Load` i `Store`

W tablicy 7.3 przedstawione są instrukcje `Load` i `Store` w trybie *pre - index*. Te same instrukcje pracują także w pozostałych trybach adresowania: **pre-index with update** oraz **post-index**.

Tabela 7.3: Instrukcje `Load` i `Store`

`LDR Rt, [Rn, #offset]`	odczytaj słowo (32-bity) z pamięci `Rt ← mem[Rn+offset]`
`LDRB Rt, [Rn, #offset]`	odczytaj bajt (8-bitów) z pamięci `Rt ← mem[Rn+offset]`
`LDRH Rt, [Rn, #offset]`	odczytaj pół-słowa (16-bitów) z pamięci do rejestru `Rt ← mem[Rn+offset]`
`LDRSB Rt, [Rn, #offset]`	odczytaj bajt z pamięci i wpisz z rozszerzeniem znaku do rejestru `Rt ← SignExtend(mem[Rn+offset])`
`LDRSH Rt, [Rn, #offset]`	odczytaj pół-słowa z pamięci i wpisz z rozszerzeniem znaku do rejestru `Rt ← SignExtend(mem[Rn+offset])`
`LDM Rn, register_list`	odczytaj słowa (32-bity) z pamięci i wpisz do rejestrów
`STR Rt, [Rn, #offset]`	zapisz słowo (32-bity) do pamięci `mem[Rn+offset] ← Rt`
`STRB Rt, [Rn, #offset]`	zapisz najmniej znaczący bajt z rejestru do pamięci `mem[Rn+offset] ← Rt`
`STRH Rt, [Rn, #offset]`	zapisz mniej znaczące pół-słowa do pamięci `mem[Rn+offset] ← Rt`
`LDM Rn, register_list`	zapisz słowa z listy rejestrów do pamięci

W instrukcjach `LDM` i `STM` kolejność rejestrów na liście nie ma znaczenia. Rejestry zostaną zapisane w kolejności ich numerów, począwszy od tego o najmniejszym numerze.

Ta zasada obowiązuje we wszystkich instrukcjach używających listy rejestrów, na przykład w instrukcjach PUSH i POP. Załóżmy, że mamy zdefiniowaną tablicę w pamięci danych RAM

```
          AREA myData, DATA, READWRITE
array1    DCD 1, 2, 3, 4, 5
```

Odczytamy zawartość pamięci do rejestrów z wykorzystaniem **pre - index**

```
LDR r0, =array1        ; r0 ← adres tablicy
LDR r1, [r0]           ; r1 ← 0x00000001
LDR r2, [r0,#4]        ; r2 ← 0x00000002
LDR r3, [r0,#8]        ; r3 ← 0x00000003
LDR r4, [r0,#12]       ; r4 ← 0x00000004
LDR r5, [r0,#16]       ; r5 ← 0x00000005
```

Odczytamy zawartość pamięci do rejestrów z wykorzystaniem **post - index**

```
LDR r0, =array1        ; r0 ← 0x20000004
LDR r1, [r0], #4       ; r1 ← 0x00000001,  r0 ← 0x20000008
LDR r2, [r0], #4       ; r2 ← 0x00000002,  r0 ← 0x2000000C
LDR r3, [r0], #4       ; r3 ← 0x00000003,  r0 ← 0x20000010
LDR r4, [r0], #4       ; r4 ← 0x00000004,  r0 ← 0x20000014
LDR r5, [r0], #4       ; r5 ← 0x00000005,  r0 ← 0x20000018
```

Odczytamy zawartość pamięci z wykorzystaniem **pre - index with update**

```
LDR r0, =array1        ; r0 ← 0x20000004
LDR r1, [r0]           ; r1 ← 0x00000001,  r0 =  0x20000004
LDR r2, [r0], #4       ; r0 ← 0x20000008,  r2 ← 0x00000002
LDR r3, [r0], #4       ; r0 ← 0x2000000C,  r3 ← 0x00000003
LDR r4, [r0], #4       ; r0 ← 0x20000010,  r4 ← 0x00000004
LDR r5, [r0], #4       ; r0 ← 0x20000014,  r5 ← 0x00000005
```

7.5. Load i Store z listą rejestrów

Kilka rejestrów może zostać załadowanych jednocześnie, z ciągłego obszaru pamięci, za pomocą jednej instrukcji LDMxx. Skojarzona z nią instrukcja STMxx pozwala na zapisanie zawartości kilku rejestrów do pamięci. Cztery dostępne tryby adresowania pokazane są w tabeli 7.4

Tabela 7.4: Tryby adresowania pamięci w instrukcjach LDM i STM

Tryb adresowania	Opis	Instrukcje
IA	Increment After	STMIA, LDMIA
IB	Increment Before	STMIB, LDMIB
DA	Decrement After	STMDA, LDMDA
DB	Decrement Before	STMDB, LDMDB

- IA: po przesłaniu słowa (32-bity) rejestr ↔ pamięć, adres pamięci jest zwiększany o 4

- IB: adres pamięci jest zwiększany o 4 przed przesłaniem słowa rejestr ↔ pamięć
- DA: po przesłaniu słowa rejestr ↔ pamięć, adres pamięci jest zmniejszany o 4
- DB: adres pamięci jest zmniejszany o 4 przed przesłaniem słowa rejestr ↔ pamięć

Format asemblerowy instrukcji:

```
LDMxx Rn{!}, {register_list}
STMxx Rn{!}, {register_list}
```

gdzie rejestr Rn zawiera adres bazowy, a xx symbolizuje jeden z trybów wymienionych w tabeli 7.4. Znak wykrzyknika jest opcjonalny. Jeżeli zostanie zastosowany, to adres w rejestrze Rn będzie zmieniany (zwiększany /zmniejszany) o 4 po każdym transferze słowa pomiędzy pamięcią a rejestrem. Kolejność zapisywania rejestrów nie zależy od ich miejsca na liście. W trakcie odczytywania danych z pamięci, rejestry są zapisywane w kolejności ich nazw: najmłodszy adres→ najmniejszy numer rejestru. Podobnie, zapis do pamięci zaczyna się od rejestru o najmniejszym numerze, kolejno, do największego numeru w nazwie.

Załóżmy, że rejestr r0 zawiera adres początku obszaru w pamięci r0 = 0x2000 0100. Instrukcja STMxx r0,{r6,r1,r4,r2} spowoduje przepisanie zawartości rejestrów z listy do kolejnych komórek pamięci, w zależności od trybu adresowania

	STMIA r0,{.}	STMIB r0,{.}	STMDA r0,{.}	STMDB r0,{.}
0x2000 00EC				
0x2000 00F0				r1
0x2000 00F4			r1	r2
0x2000 00F8			r2	r4
0x2000 00FC			r4	r6
r0 = 0x2000 0100	r1		r6	
0x2000 0104	r2	r1		
0x2000 0108	r4	r2		
0x2000 010C	r6	r4		
0x2000 0110		r6		

Załóżmy, że zawartość komórek pamięci jest jak w tabeli poniżej i r0 = 0x2000 0100. Instrukcja LDMxx r0,{r6,r1,r4,r2} spowoduje przepisanie kolejnych komórek pamięci do rejestrów z listy, w zależności od trybu adresowania

		LDMIA r0,{.}	LDMIB r0,{.}	LDMDA r0,{.}	LDMDB r0,{.}
0x2000 00EC	-5				
0x2000 00F0	-4				r1 = -4
0x2000 00F4	-3			r1 = -3	r2 = -3
0x2000 00F8	-2			r2 = -2	r4 = -2
0x2000 00FC	-1			r4 = -1	r6 = -1
0x2000 0100	1	r1 = 1		r6 = 1	
0x2000 0104	2	r2 = 2	r1 = 2		
0x2000 0108	3	r4 = 3	r2 = 3		
0x2000 010C	4	r6 = 4	r4 = 4		
0x2000 0110	5		r6 = 5		

Procesor STM32F103 nie wspiera instrukcji STMIB, STMDA, LDMIB i LDMDA. Dlatego zostały oznaczone kolorem czerwonym. Próba wpisania wymienionych instrukcji do programu asemblerowego powoduje komunikat: `error: IB adressing not allowed for STM` i analogicznie w pozostałych przypadkach.

Rozdział 8

Instrukcje rozgałęzień *Branch*

Najczęściej instrukcje są wykonywane w takiej kolejności, w jakiej zostały napisane w asemblerze. Gdy wykonanie instrukcji zostanie zakończone, licznik programu PC jest zwiększany i procesor przechodzi do kolejnej instrukcji. Ale dynamiczna zmiana zawartości rejestru PC, w trakcie wykonywania programu, powoduje zmianę porządku egzekucji (*flow of control*) programu. Są cztery główne sposoby zmiany *flow of control*.

1. Instrukcje rozgałęzień /skoków (*branch instructions*)
2. Wykonanie warunkowe (*conditional execution*)
3. Wywołanie podprogramu (*calling a subroutine*)
4. Obsługa przerwań (*Interrupts*)

W tym rozdziale zajmiemy się dwoma pierwszymi sposobami rozejść programowych. Wywoływanie podprogramów omówione jest w rozdziale 10, a obsługa przerwań w 11.

8.1. Testowanie warunków

Wiemy, że przy porównywaniu liczb ze znakiem użyteczne informacje mogą być zawarte w innych flagach niż gdy porównujemy liczby bez znaku. Wiele instrukcji z zestawu ARM umożliwia stosowanie dwuliterowych przyrostków, które często są inne dla liczb ze znakiem i bez znaku. Porównanie zawarte jest w tabeli 8.1

Tabela 8.1: Porównanie przyrostków dla liczb ze znakiem i bez znaku

Symbol	Liczba ze znakiem	Liczba bez znaku	Testowana relacja
==	EQ	EQ	Equal to
!=	NE	NE	Not equal to
>	GT	HI	Greater than
⩾	GE	HS	Greater than or equal to
<	LT	LO	Less than
⩽	LE	LS	Less than or equal to

Zestawienie wszystkich przyrostków określających testowane warunki zawarte jest w tabeli 8.2. Jeżeli instrukcja nie ma przyrostka określającego warunek jej wykonania, to wartością domyślną jest "AL" czyli "zawsze". Wykorzystanie warunkowej wersji instrukcji zostanie pokazane na przykładzie obliczania wartości absolutnej liczby ze znakiem (w `r1`).

```
CMP   r1,#0        ; modyfikuje flagi N, Z, C oraz V
RSBLT r1,r1,#0     ; gdy r1 < 0, to wykonaj r1 = 0 - r1
```

Tabela 8.2: Przyrostki warunków testowanych w zestawie instrukcji ARM

Przyrostek	**Opis**	**Flagi**	**Implementacja logiczna**
`EQ`	**EQ**ual	`Z = 1`	Z
`NE`	**N**ot **E**qual	`Z = 0`	$\bar{Z}$
`CS/HS`	unsigned **H**igher or **S**ame	`C = 1`	C
`CC/LO`	unsigned **LO**wer	`C = 0`	$\bar{C}$
`MI`	**MI**nus (negative)	`N = 1`	N
`PL`	**PL**us (positive or zero)	`N = 0`	$\bar{N}$
`VS`	o**V**erflow **S**et	`V = 1`	V
`VC`	o**V**erflow **C**lear	`V = 0`	$\bar{V}$
`HI`	unsigned **HI**gher	`C = 1 & Z = 0`	$C\bar{Z}$
`LS`	unsigned **L**ower or **S**ame	`C = 0 & Z = 1`	$\bar{C} + Z$
`GE`	signed **G**reater or **E**qual	`N = V`	$NV + \bar{N}\bar{V}$
`LT`	signed **L**ess **T**han	`N != V`	$N\bar{V} + \bar{N}V$
`GT`	signed **G**reater **T**han	`Z = 0 && N = V`	$Z(NV + \bar{N}\bar{V})$
`LE`	signed **L**ess than or **E**qual	$Z = 1 \mid\mid N! = V$	$Z + N\bar{V} + \bar{N}V$
`AL`	**AL**ways		

Instrukcja `CMP r0,r1` wykonuje odejmowanie `r0 - r1`, modyfikuje flagi, a wynik odrzuca. Jeżeli w obu porównywanych rejestrach są liczby bez znaku, to flaga *Borrow* (przeniesienie wsteczne), która nie jest zrealizowana fizycznie w procesorze, jest równa negacji flagi *Carry* (strona 19). Dlatego przyrostki `HS, LO, HI` oraz `LS` testują flagę *Carry* i flagę *Zero* (jeśli to konieczne).

Jeżeli oba rejestry w instrukcji `CMP r0,r1` zawierają liczby ze znakiem, to testowane są flagi V *Overflow* i N *Negative*.

- Gdy odejmujemy (`r0 - r1`) dwie poprawne liczby tego samego znaku, to wynik nie przekroczy zakresu związanego z formatem liczby. Na przykład $5-3=2$, $(-5)-(-2)=-3$, czyli moduł wyniku jest mniejszy od modułu większej z liczb. Nie ma przepełnienia $V=0$. Gdy `r0` $\geqslant$ `r1` to $N=0$, gdy `r0` $<$ `r1` to $N=1$.
- Gdy odejmujemy (`r0 - r1`) dwie poprawne liczby różnych znaków i nie wystąpi przepełnienie, to o relacji decyduje flaga N jak powyżej. Jeżeli zostanie zapalona flaga V, to znaczy, że bit znaku jest niepoprawny, czyli odwrotny i o relacji decyduje negacja flagi N. Powtórzmy przykłady ze strony 21

```
      Carry 1 ← 0 0 0                          Carry 0 ← 1 1 1
                1 1 0 1   -3                               0 0 1 1   +3
                1 0 1 0   -6                               0 1 0 1   +5
V = 1⊕0 = 1     0 1 1 1   +7          V = 0⊕1 = 1          1 0 0 0   -8
```

Przykład z lewej strony: `r0 = -3, r1 = +6`, liczby różnych znaków. Instrukcja `CMP r0,r1` wykonuje odejmowanie `r0 - r1` dodając do `r0` negację `r1`. Wynik nie mieści się w formacie, zapalana jest flaga V. Bit znaku jest odwrotny (zero zamiast jedynki), czyli $\bar{N}=1$, poprawna relacja to `r0` $<$ `r1`. Jest tak rzeczywiście, gdyż $-3<6$.

Przykład z prawej strony: `r0 = +3, r1 = -5`, liczby różnych znaków. Instrukcja `CMP r0,r1` wykonuje odejmowanie `r0 - r1` dodając do `r0` negację `r1`. Wynik nie mieści się w formacie, zapalana jest flaga V. Bit znaku jest odwrotny (jedynka zamiast zera), czyli $\bar{N}=1$, poprawna relacja to `r0` $\geqslant$ `r1`. Jest tak rzeczywiście, gdyż $3 \geqslant -5$.

Podsumowanie naszych rozważań zawarte jest w tabeli 8.3

Tabela 8.3: Znaczenie flag V i N gdy instrukcja `CMP r0,r1` porównuje liczby ze znakiem

	N = 0	**N = 1**
V = 0	`r0` $\geqslant$ `r1`	`r0` $<$ `r1`
V = 1	`r0` $<$ `r1`	`r0` $\geqslant$ `r1`

Na podstawie wyników zebranych w tabeli 8.3 możemy, dla liczb ze znakiem, określić następującą regułę

- Jeżeli $N = V$, to `r0` jest większy lub równy `r1`
- Jeżeli $N \neq V$, to `r0` jest mniejszy niż `r1`

Dlatego przyrostki dla liczb ze znakiem: `GE, LT, GT`, oraz `LE` sprawdzają równość, albo brak równości flag V i N (tabela 8.2).

Tabela 8.4: Instrukcje skoków bezwarunkowych *Branch*

Instrukcja	**Opis**
`B label`	Skok do etykiety $PC = label$
`BX Rm`	Skok do adresu w rejestrze Rm $PC = Rm$

8.2. Instrukcje rozgałęzień

Instrukcja skoku bezwarunkowego *Branch*: `B label` ładuje adres skojarzony z etykietą `label` do licznika programu PC bez sprawdzania żadnych warunków. Powoduje to przekierowanie wykonania programu do instrukcji oznaczonej tą etykietą. Instrukcja `BX Rm` powoduje przeniesienie wykonania programu do adresu zawartego w rejestrze `Rm` (tabela 8.4).

Dołączenie do instrukcji `B` przyrostków z tabeli 8.2 zamienia ją w cały zestaw instrukcji warunkowych działających zgodnie z zasadą: *jeżeli warunek jest spełniony idź do label.* Gdy warunek nie jest spełniony wykonywana jest instrukcja kolejna za instrukcją skoku warunkowego. Jeżeli porównujemy dwie liczby `0xFFFFFFFF` i `0x00000001`, to która jest większa? Zależy to od interpretacji: jeżeli są to liczby ze znakiem, to pierwsza (-1) jest mniejsza od drugiej (1). Jeżeli bez znaku, to pierwsza $(2^{32} - 1)$ jest większa. W przypadku liczb ze znakiem zastosujemy instrukcję `BGT`, bez znaku `BHI`, zgodnie z tablicą 8.1. Przykład kodu C i tłumaczenia na asembler w obu przypadkach podany jest w rozdziale 2.9.

W zestawie instrukcji Cortex-M istnieją dwie instrukcje 16-bitowe skoku warunkowego z porównaniem zawartości rejestru do zera: `CBZ` *Compare and Branch on Zero* oraz `CBNZ` *Compare and Branch on NonZero*. Każda z nich ma w kodzie 5 bitów na adres relatywny (obliczany względem PC). Skok może być wykonany tylko w przód (w kierunku większych adresów) w zakresie od 4 do 130 bajtów kodu instrukcji. Każda z tych instrukcji może być zastąpiona dwiema instrukcjami bez ograniczenia długości skoku.

`CBZ r1, label`	`CMP r1, #0` `BEQ label  ; branch if equal`
`CBNZ r1, label`	`CMP r1, #0` `BNE label  ; branch if not equal`

Tabela 8.5: Instrukcje skoków warunkowych *Branch*

Instrukcja	**Opis**	**Testowane flagi**
`BEQ label`	Branch if **EQ**ual	`Z = 1`
`BNE label`	Branch if **N**ot **E**qual	`Z = 0`
`BCS/BHS label`	Branch if unsigned **H**igher or **S**ame	`C = 1`
`BCC/BLO label`	Branch if unsigned **LO**wer	`C = 0`
`BMI label`	Branch if **MI**nus (negative)	`N = 1`
`BPL label`	Branch if **PL**us (positive or zero)	`N = 0`
`BVS label`	Branch if o**V**erflow **S**et	`V = 1`
`BVC label`	Branch if o**V**erflow **C**lear	`V = 0`
`BHI label`	Branch if unsigned **HI**gher	`C = 1 & Z = 0`
`BLS label`	Branch if unsigned **L**ower or **S**ame	`C = 0 & Z = 1`
`BGE label`	Branch if signed **G**reater or **E**qual	`N = V`
`BLT label`	Branch if signed **L**ess **T**han	`N != V`
`BGT label`	Branch if signed **G**reater **T**han	`Z = 0 && N = V`
`BLE label`	Branch if signed **L**ess than or **E**qual	$Z = 1 \mid\mid N! = V$

Istnieją także specjalne instrukcje skoków bezwarunkowych służące do wywoływania podprogramów, czyli zachowujące adres powrotu w *Link Register* LR.

Tabela 8.6: Instrukcje skoków bezwarunkowych do wywoływania podprogramów

Instrukcja	**Opis**	**Operacja**
`BL label`	Branch with link	$LR = PC + 4, \quad PC = label$
`BLX Rm`	Branch with link and exchange	$LR = PC + 4, \quad PC = Rm$

Tabela 8.7: Warunkowe wersje instrukcji dodawania `ADD`

Instrukcja	**Opis**	**Testowane flagi**
`ADD{S}EQ r2,r1,r0`	Add if **EQ**ual	`Z = 1`
`ADD{S}NE r2,r1,r0`	Add if **N**ot **E**qual	`Z = 0`
`ADD{S}HS r2,r1,r0`	Add if unsigned **H**igher or **S**ame	`C = 1`
`ADD{S}LO r2,r1,r0`	Add if unsigned **LO**wer	`C = 0`
`ADD{S}MI r2,r1,r0`	Add if **MI**nus (negative)	`N = 1`
`ADD{S}PL r2,r1,r0`	Add if **PL**us (positive or zero)	`N = 0`
`ADD{S}VS r2,r1,r0`	Add if o**V**erflow **S**et	`V = 1`
`ADD{S}VC r2,r1,r0`	Add if o**V**erflow **C**lear	`V = 0`
`ADD{S}HI r2,r1,r0`	Add if unsigned **HI**gher	`C = 1 & Z = 0`
`ADD{S}LS r2,r1,r0`	Add if unsigned **L**ower or **S**ame	`C = 0 & Z = 1`
`ADD{S}GE r2,r1,r0`	Add if signed **G**reater or **E**qual	`N = V`
`ADD{S}LT r2,r1,r0`	Add if signed **L**ess **T**han	`N != V`
`ADD{S}GT r2,r1,r0`	Add if signed **G**reater **T**han	`Z = 0 && N = V`
`ADD{S}LE r2,r1,r0`	Add if signed **L**ess than or **E**qual	$Z = 1 \mid\mid N! = V$

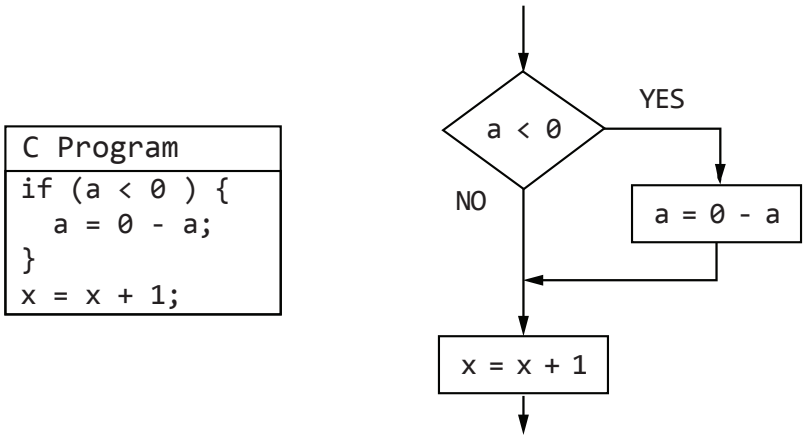
8.3. Warunkowe wykonanie instrukcji

Oprócz czterech instrukcji służących porównywaniu (`CMP`, `CMN`, `TEQ` i `TST`), większość instrukcji zestawu Thumb-2 modyfikuje flagi po dodaniu przyrostka `S`. Cechą wyróżniającą procesory ARM jest to, że instrukcje mogą być wykonywane warunkowo po dodaniu przyrostków z tabeli 8.2. Prawie wszystkie instrukcje mają swoje wersje warunkowe: są wykonywane tylko wtedy, gdy odpowiedni warunek jest spełniony. Przykładem jest instrukcja rozgałęzienia *Branch*, jej wersje warunkowe podano w tabeli 8.5.

Rozważymy teraz instrukcję dodawania `ADD` i jej wersje warunkowe. Domyślnie, instrukcja `ADD r2,r1,r0` jest wykonywana zawsze, bez względu na stan flag. Wersje warunkowe tej instrukcji podane są w tabeli 8.7. Nawiasy {} w mnemoniku oznaczają, że `S` jest opcjonalne: na przykład "dodaj jeśli równe" jest w wersji z modyfikacją flag `ADDSEQ` i bez modyfikacji flag `ADDEQ`. Przykład:

```
CMP    r1,r0        ; odejmij r1-r0, ustaw flagi
ADDSPL r3,r3,#1     ; jeżeli r1 ⩾ r0, to inkrementuj r3 i ustaw flagi
```

8.4. Instrukcja kompilatora `If-then`



W języku C instrukcja `if - then` wykonuje zawiązany z nią blok programu, gdy wartość wyrażenia logicznego jest `TRUE` (różna od zera). W przeciwnym przypadku (`FALSE` ↔ zero) blok ten jest pomijany. W przykładzie wyliczany jest moduł zmiennej `a` oraz inkrementowana jest zmienna `x`. Zakładamy, że liczby są ze znakiem i `r0 = a` oraz `r1 = x`. Poniżej są dwie implementacje asemblerowe: ze skokiem warunkowym i z instrukcją warunkową

```
        ; r0 = a, r1 = x
        CMP   r0,#0          ; porównaj 'a' z zerem (a - 0)
        BGE   endif          ; idź do 'endif' jeśli a ⩾ 0
        RSBS  r0,r0,#0       ; a = −a
endif   ADDS  r1,r1,#1       ; x += 1
```

```
        ; r0 = a, r1 = x
        CMP   r0,#0          ; porównaj 'a' z zerem (a - 0)
        RSBLT r0,r0,#0       ; a = −a, gdy a < 0
        ADDS  r1,r1,#1       ; x += 1
```

8.4.1. Złożone wyrażenie logiczne z OR

Wyrażenia logiczne mogą być ciągiem działań logicznych łączonych za pomocą operatorów AND (&&), OR (||) i NOT (!). Priorytety operatorów języka C podane są w tabeli B.2. Przykładowe wyrażenia logiczne zapisane w kodzie źródłowym C:

```
x > 8 && x < 12        x == 8 || x == 12        !(x == 8 || x == 12)
```

Złożone wyrażenie logiczne z operatorem OR może być zrealizowane jako ciąg instrukcji asemblera realizujących kolejne porównania. Przykład:

Program C	**Program asemblerowy**
`int x;`	`; r0 = x`
`unsigned int b = 0;`	`MOVS r1,#0 ; r1 = b = 0`
`if(x <= 8 || x >= 12){`	`CMP r0,#8 ; porównaj x z 8`
`  b = 1;`	`BLE than ; skocz gdy x ⩽ 8`
`}`	`CMP r0,#12 ; porównaj x z 12`
	`BLT endif ; skocz gdy x < 12`
	`than MOVS r1,#1 ; b = 1`
	`endif`

8.4.2. Złożone wyrażenie logiczne z AND

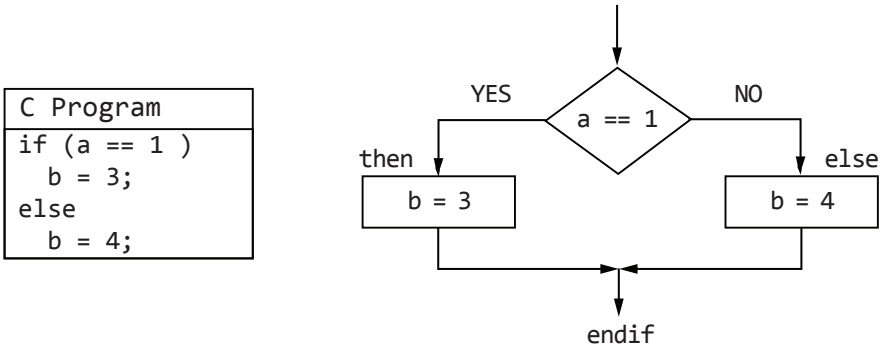
Wyrażenie logiczne z AND możemy zamienić na równoważne wyrażenie z OR stosując prawo de Morgana

$$NOT(\boldsymbol{A}\ AND\ \boldsymbol{B}) = (NOT\boldsymbol{A})\ OR\ (NOT\boldsymbol{B})$$

czyli `!(x > 8 && x < 12)` = `!(x > 8) || !(x < 12)` = `(x ⩽ 8) || (x ⩾ 12)`

Program C	**Program asemblerowy**
`int x;`	`; r0 = x`
`unsigned int b = 0;`	`MOVS r1,#0 ; r1 = b = 0`
`if(x > 8 && x < 12){`	`CMP r0,#8 ; porównaj x z 8`
`  b = 1;`	`BLE endif ; skocz gdy x ⩽ 8`
`}`	`CMP r0,#12 ; porównaj x z 12`
	`BGE endif ; skocz gdy x ⩾ 12`
	`MOVS r1,#1 ; b = 1`
	`endif`

8.5. Instrukcja kompilatora `If-then-else`



Gdy warunek jest spełniony, wykonywany jest blok programowy oznaczony `than`, gdy nie jest spełniony, blok `else`. Przykład

Program C	**Program asemblerowy**
`int x;`	`; r0 = x`
`unsigned int b;`	
`if(x == 1)`	`CMP r0,#1 ; porównaj x z 1`
`  b = 3;`	`BNE else ; skocz gdy x ≠ 1`
`else`	`MOVS r1,#3 ; b = 3`
`  b = 4;`	`B endif`
	`else MOVS r1,#4 ; b = 4`
	`endif`

8.6. Instrukcje pętli

Pętla jest instrukcją, której zadaniem jest powtarzanie wykonania bloku instrukcji (**ciała pętli**). W języku C każda pętla posiada **wyrażenie sterujące**. Za każdym razem, kiedy wykonywane jest ciało pętli (jednokrotne wykonanie to **iteracja**), następuje obliczenie wyrażenia sterującego. Jeżeli wyrażenie to ma wartość niezerową ("prawda"), to pętla jest kontynuowana.

W języku C do dyspozycji są trzy instrukcje pętli: `while`, `do` oraz `for`. Instrukcja `while` służy do budowania pętli, w których wyrażenie sterujące jest sprawdzane *przed* wykonaniem ciała pętli. Instrukcja `do` wykonuje ciało pętli i dopiero *potem* oblicza wyrażenie sterujące. Instrukcja `for` jest odpowiednia do pętli, których iteracja opiera się na zmiennej licznikowej (**licznik pętli**).

8.6.1. Pętla `For`

Pętla `for` powtarza egzekucję bloku instrukcji tak długo, aż `warunek` osiągnie wartość logiczną `FALSE`.

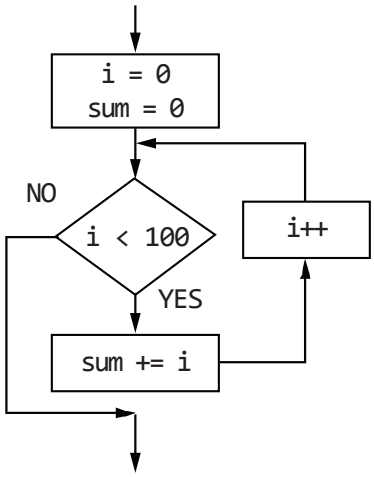
```
for (inicjalizacja_licznika; warunek; zmiana_licznika){
        // ciało pętli
}
```

1. Jako pierwsza wykonywana jest instrukcja `inicjalizacja_licznika`, która nadaje wartość początkową zmiennej licznikowej.

2. Następnie sprawdzany jest warunek, jeżeli jego wartość logiczna jest `TRUE`, to wykonywany jest blok instrukcji objętych nawiasami. Warunek jest sprawdzany przed wykonaniem ciała pętli. W szczególności, jeżeli już przy pierwszym sprawdzeniu warunek nie jest spełniony, ciało pętli nigdy nie będzie wykonane.
3. Po zakończeniu egzekucji ciała pętli, wykonywana jest `zmiana_licznika` i program przechodzi do punktu 2.

W ramach przykładu napiszemy program obliczający sumę stu pierwszych liczb naturalnych (od 0 do 99). Rejestr `r0` będzie licznikiem pętli, a `r1` to wyliczona suma. Instrukcje w wersji z ustawianiem flag `MOVS` i `ADDS` są 16-bitowe. Ich stosowanie (zamiast 32-bitowych bez modyfikacji falg) skraca kod.

```
C Program
int i;
int sum = 0;

for(i = 0; i < 100; i++){
   sum += i;
}
```



Program C	Program asemblerowy

`int i;`
`int sum = 0;`
`for(i = 0; i < 100, i++){`
`  sum += i;`
`}`
	`  ; licznik pętli`
	`  MOVS r1,#0  ; sum = 0`
	`  MOVS r0,#0  ; i = 0`
	`  B  check  ; skocz do warunku`
	`loop  ADD  r1,r1,r0  ; sum += k`
	`  ADDS r0,r0,#1  ; i++`
	`check  CMP  r0,#100  ; czy i < 100`
	`  BLT  loop  ; pętla jeśli nie koniec`
	`endloop`

8.6.2. Pętla `While`

Pętla `while` jest ściśle spokrewniona z pętlą `for`, gdyż obie sprawdzają warunek *przed* wykonaniem ciała pętli. Poza kilkoma rzadkimi wyjątkami pętla `for` może zostać zapisana jako równoważna pętla `while`

```
inicjalizacja_licznika
while (warunek) {
    // ciało pętli
    zmiana_licznika
}
```

Zgodnie z tym schematem, program obliczenia sumy stu pierwszych liczb naturalnych zapiszemy następująco

```
int sum = 0; i = 0;
while (i < 100) {
    sum += i;
    i++;
}
```

W wyniku kompilacji uzyskamy identyczny kod asemblerowy jak w przypadku pętli `for`.

8.6.3. Pętla `Do while`

Pętla `do while` powstaje przez przeniesienie warunku na koniec, czyli najpierw wykonujemy instrukcje ciała pętli, a następnie sprawdzamy warunek. To powoduje, że instrukcje ciała będą wykonane co najmniej raz. Przykład obliczenia sumy pierwszych stu liczb naturalnych będzie miał postać

Program C	**Program asemblerowy**
`int i = 0;`	`MOVS r0,#0 ; i = 0`
`int sum = 0;`	`MOVS r1,#0 ; sum = 0`
`do {`	
`  sum += i;`	`loop ADD r1,r1,r0 ; sum += k`
`  i++;`	`ADDS r0,r0,#1 ; i++`
`} while (i < 100);`	`CMP r0,#100 ; czy i < 100`
	`BLT loop ; pętla jeśli nie koniec`
	`endloop`

Jak widać, kod asemblerowy jest identyczny z wyjątkiem tego, że zniknął skok bezwarunkowy `B check`, który zapewniał sprawdzenie warunku przed wykonaniem instrukcji ciała pętli.

8.7. Instrukcja kompilatora `Switch`

Instrukcja `switch` umożliwia wykonanie wielokrotnego wyboru na podstawie przełącznika (*switch expression*). Jeżeli przełącznik ma wartość całkowitą równą jednej z wymienionych przy etykietach `case`, wykonywany jest skok do etykiety o zgodnej wartości liczbowej. Jeżeli wartość przełącznika nie odpowiada żadnej wartości z listy, program skacze do etykiety `default`. Instrukcja `switch` składa się z opcjonalnej etykiety `default`, listy etykiet `case` połączonych z liczbami całkowitymi oraz instrukcji związanych z etykietami.

- Przełącznik musi być liczbą całkowitą.
- Kolejność etykiet (liczb całkowitych związanych z `case`) nie jest istotna, ale wartości nie mogą się powtarzać.
- Program skacze do właściwej etykiety i wykonuje kolejno wszystkie instrukcje począwszy od wybranej etykiety, także kolejne, które mogą być przypisane do innych etykiet. Dlatego, jeżeli chcemy, aby były wykonane tylko instrukcje związane z jedną etykietą, należy na końcu wstawić `break`, co spowoduje przejście do instrukcji następnej, za strukturą `case`.

Napiszemy funkcję, która będzie zamieniała 4-bitową liczbę binarną na hexadecymalny kod ASCII. Zgodnie z kodowaniem ASCII, liczby od 0 do 9 zamienimy dodając wartość `0x30`, natomiast liczby od 10 do 15 zamienimy na znaki od `'A'` do `'F'`. W programie asemblerowym wykorzystamy instrukcję `TBB` *Table Branch Byte*. Format tej instrukcji:

```
TBB [pc,Rn]     ; PC = PC + 4 + 2 * BranchTable[Rn]
```

Bezpośrednio pod instrukcją `TBB` musi być umieszczona tablica bajtów, z których każdy jest przesunięciem i służy do wyliczenia adresu docelowego skoku. Wyliczenie wykonywane jest zgodnie z formułą podaną w formacie instrukcji:

```
adres_docelowy = PC + 4 + 2 * BranchTable[Rn]
```

Jeżeli potrzebne jest przesunięcie większe niż jeden bajt, możemy zastosować instrukcję `TBH` *Table Branch Halfword*, która używa 16-bitowych pół-słów zamiast bajtów. Poniżej jest przykład realizacji asemblerowej z wykorzystaniem instrukcji `TBB`.

Program C

```c
int b;

char bin2Hex(int b) {
char c;
switch(b) {
  case 10: c = 'A';
           break;
  case 11: c = 'B';
           break;
  case 12: c = 'C';
           break;
  case 13: c = 'D';
           break;
  case 14: c = 'E';
           break;
  case 15: c = 'F';
           break;
  default: c = x + 0x30;
  }
  return c;
}
```

Program asemblerowy

```
         EXPORT bin2Hex
         ALIGN
bin2Hex  PROC
         SUB  r1,r0,#10
         CMP  r1,#6
         BCS  default    ; skocz gdy cyfra
         TBB  [pc,r1]
BrTable  DCB  (caseA - BrTable)/2
         DCB  (caseB - BrTable)/2
         DCB  (caseC - BrTable)/2
         DCB  (caseD - BrTable)/2
         DCB  (caseE - BrTable)/2
         DCB  (caseF - BrTable)/2
         ALIGN
caseA    MOVS r0,#'A'
         BX   lr         ; powrót PC = LR
caseB    MOVS r0,#'B'
         BX   lr         ; powrót PC = LR
caseC    MOVS r0,#'C'
         BX   lr         ; powrót PC = LR
caseD    MOVS r0,#'D'
         BX   lr         ; powrót PC = LR
caseE    MOVS r0,#'E'
         BX   lr         ; powrót PC = LR
caseF    MOVS r0,#'F'
         BX   lr         ; powrót PC = LR
default  ADDS r0,r0,#0x30
         UXTB r0,r0      ; rozszerzenie do 32-bitów
         BX   lr         ; powrót PC = LR
         ENDP
```

Rozdział 9

Wyświetlacz LCD 1602

Zestaw uruchomieniowy ZL27ARM wyposażony jest w wyświetlacz ciekłokrystaliczny LCD1602 z wbudowanym sterownikiem Hitachi HD44780. Wyświetlacz ma dwie linie po 16 znaków, podświetlenie i regulowany kontrast. Znaki 5 × 7 punktów z kursorem w formie podkreślenia. W dodatku A zamieszczony jest schemat połączenia wyświetlacza z mikrokontrolerem.

Tabela 9.1: Sygnały wyświetlacza LCD1602

Sygnał	**Port Cortex-M**	**Funkcja**
`RS`	`PC12` (output)	*Register Select*, wybór rejestru: 0: Rejestr instrukcji (gdy zapis) Flaga Busy oraz licznik adresu (gdy odczyt) 1: Rejestr danych (zapis i odczyt)
`R/W`	`PC11` (output)	*Read /Write*, sygnał sterujący kierunkiem transferu: 0: Zapis do LCD1602 (write) 1: Odczyt z LCD1602 do mikrokontrolera (read)
`E`	`PC10` (output)	*Enable* sygnał zapisu/odczytu, aktywne zbocze opadające

`DB7`
`DB6`
`DB5`
`DB4`
	`PC0` (input/output)
	`PC1` (input/output)
	`PC2` (input/output)
	`PC3` (input/output)
		Data Bits: b4 - b7
		trzystanowe wejścia/wyjścia danych,
		w trybie z transferem 4 - bitowym
		bajt jest przesyłany jako dwie połówki (*nibbles*)

`DB3,DB2,` `DB1,DB0`	niepodłączone	mniej znacząca połówka bajtu danych, nieużywana w trybie 4 - bitowym

Rejestry IR i DR. Sterownik HD44780 ma dwa 8-bitowe rejestry: instrukcji IR i danych DR. Do rejestru IR należy wpisywać polecenia takie jak kasowanie wyświetlacza, przesuwanie kursora oraz adresy pamięci wyświetlacza DDRAM *Display Data RAM* i adresy generatora znaków CGRAM *Character Generator RAM*. Możliwa jest tylko operacja zapisu do rejestru instrukcji IR (nie można go czytać).

Rejestr danych DR, dane otrzymane z mikrokontrolera przekazuje do pamięci DDRAM lub CGRAM. Przy transferze w odwrotnym kierunku, dane czytane z DDRAM lub CGRAM trafiają do rejestru DR z którego może je odczytać mikrokontroler. Bajt zapisany przez mikrokontroler do rejestru DR jest automatycznie zapisywany do DDRAM lub CGRAM w wyniku wewnętrznej operacji sterownika HD44780.

Busy Flag (BF). Gdy flaga zajętości *Busy Flag* jest jedynką, sterownik HD44780 wykonuje wewnętrzne operacje i następna instrukcja nie może być do niego wysłana. Gdy RS = 0 i R/W = 1 (Tablica 9.2), flaga *Busy* jest wystawiona na DB7. Następna instrukcja może być wpisana do wyświetlacza dopiero po upewnieniu się, że flaga *Busy* jest wyzerowana.

Address Counter (AC) przydziela adresy obu pamięciom o dostępie swobodnym: DDRAM i CGRAM. Gdy polecenie z adresem jest wpisywane do rejestru IR, adres ten

Tabela 9.2: Sygnały wyboru rejestrów

RS	**R/W**	**Operacja**
0	0	zapis polecenia do rejestru IR, wywoła operację wewnętrzną
0	1	odczyt flagi *Busy* (DB7) i licznika adresu AC (DB6-DB0)
1	0	zapis bajtu do DR i dalej wewnętrznie do DDRAM lub CGRAM
1	1	odczyt bajtu wewnętrznie z DDRAM lub CGRAM do DR i dalej do μP

jest następnie wpisywany do licznika adresów AC. Instrukcja zawiera także informację, której pamięci dotyczy przysłany adres. Po zapisie (odczycie) do DDRAM lub CGRAM, adres jest automatycznie inkrementowany (może być także dekrementowany). Zawartość licznika AC można odczytać razem z flagą *Busy* - patrz tabela 9.2.

Display Data RAM (DDRAM) zawiera kody ASCII wyświetlanych znaków. Wielkość tej pamięci to 80 znaków. W przypadku LCD1602 (dwie linie po 16 znaków) pamięć DDRAM podzielona jest na dwa obszary po 40 znaków. Górna linia to adresy od `0x00` do `0x27`. Dolna linia to adresy od `0x40` do `0x67`. Po wyzerowaniu wyświetlane są początki obu linii:

00	01	02	03	04	05	06	07	08	09	0A	0B	0C	0D	0E	0F
40	41	42	43	44	45	46	47	48	49	4A	4B	0C	4D	4E	4F

Obie linie zawijają się, czyli po przesunięciu wyświetlanych adresów o jedną pozycję w prawo (wyświetlacz jest nieruchomy, przesuwamy wyświetlany obszar pamięci):

27	00	01	02	03	04	05	06	07	08	09	0A	0B	0C	0D	0E
67	40	41	42	43	44	45	46	47	48	49	4A	4B	4C	4D	4E

A teraz przesuniemy o dwie pozycje w lewo

01	02	03	04	05	06	07	08	09	0A	0B	0C	0D	0E	0F	10
41	42	43	44	45	46	47	48	49	4A	4B	0C	4D	4E	4F	50

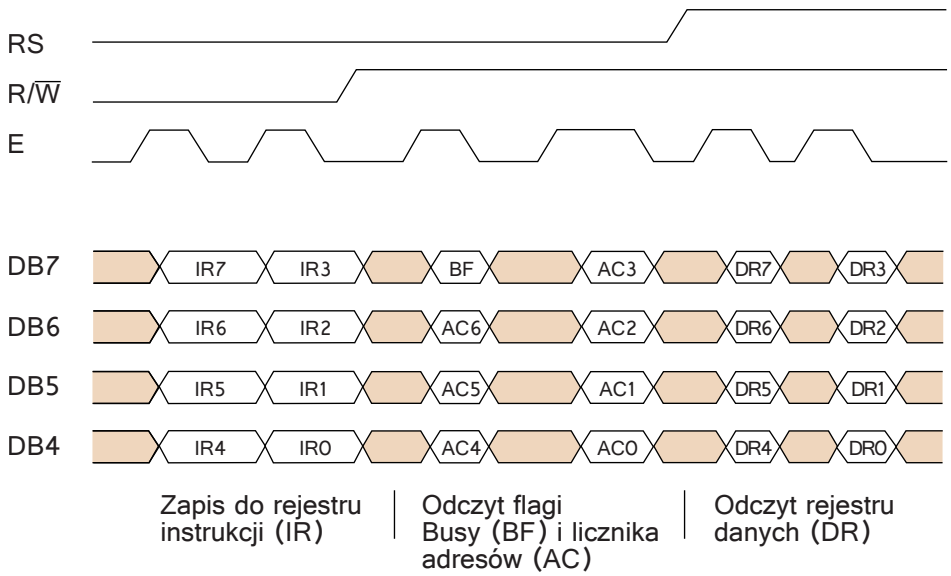
Dzięki temu możemy uzyskać przesuwające się napisy: wpisujemy cały tekst do DDRAM i co określony czas wysyłamy polecenie *Display Shift*.

Character Generator ROM (CGROM). Generator znaków jest wpisany na stałe do sterownika HD44780. W dodatku B pokazany jest generator standardowy (rysunek B.1) oraz jego wersja europejska (rysunek B.2).

Character Generator RAM (CGRAM). Sterownik HD44780 umożliwia zdefiniowanie ośmiu znaków użytkownika w formacie 5 x 8 punktów. Rysunki B.1 i B.2 w skrajnej lewej kolumnie pokazują jakie kody należy wpisać do DDRAM, aby wyświetlić znaki użytkownika. Kody są powielone: każdy znak występuje w dwóch miejscach tablicy.

9.1. Wymiana danych z wyświetlaczem (4 bity danych)

Sterownik wyświetlacza może przesyłać dane w trybie dwóch transferów 4-bitowych albo jednego transferu 8-bitowego. W zestawie uruchomieniowym ZL27ARM zastosowano tryb 4-bitowy.



Rys. 9.1: Przykład transferu danych w trybie 4-bitowym

W trybie 4-bitowym wykorzystywane są tylko cztery linie magistrali danych: od DB4 do DB7, linie DB0 do DB3 są nieaktywne. Przesłanie bajtu wymaga dwóch kolejnych transferów 4-bitowych. Jako pierwsze przesyłane są cztery starsze bity (`b4 - b7`). Młodsze bity (`b0 - b3`) są przesyłane jako drugie. Po dwóch transferach z danymi należy sprawdzić stan flagi *Busy* - to wymaga dwóch odczytów 4-bitowych, jak pokazano na rysunku 9.1. Kolejny transfer jest możliwy dopiero po odczytaniu zerowego stanu flagi *Busy*.

9.2. Instrukcje

Mikrokontroler ma bezpośredni dostęp tylko do dwóch rejestrów sterownika HD44780: do IR *Instruction Register* i do DR *Data Register*. Wewnętrzne operacje sterownika są inicjowane zapisem (odczytem) do tych rejestrów. Najczęściej wykonywane są transfery do wewnętrznej pamięci DDRAM sterownika. Automatyczna inkrementacja (dekrementacja) adresu w AC po każdym zapisie (odczycie) zmniejsza ilość bajtów transferowanych przy zapisie (odczycie) stringu znaków wyświetlanych.

W czasie wykonywania wewnętrznych operacji, żaden bajt oprócz odczytu flagi *Busy* nie może być odczytany (zapisany) ze sterownika. Przed każdym nowym odczytem (zapisem) należy odczekać, aż flaga *Busy* będzie miała stan zerowy. Poniżej opisane są wszystkie instrukcje sterownika HD44780 (w nawiasach czasy wykonania).

Clear Display

RS	R/W	DB7	DB6	DB5	DB4	DB3	DB2	DB1	DB0
0	0	0	0	0	0	0	0	0	1

Wpisuje spacje (`0x20`) do całej pamięci DDRAM, do licznika adresu AC wpisuje zero i likwiduje ewentualne przesunięcie wyświetlanych znaków. Kursor przesuwa na pozycję zerową (pierwsza w pierwszej linii). W instrukcji *Entry Mode* bit I/D (*Increment Mode*) ustawia na 1 i nie zmienia bitu S (kierunek przesuwania).

Return Home (1.52 ms)

RS	R/W	DB7	DB6	DB5	DB4	DB3	DB2	DB1	DB0
0	0	0	0	0	0	0	0	1	*

Do licznika adresu AC wpisuje zero i likwiduje ewentualne przesunięcie wyświetlanych znaków. Nie zmienia zawartości pamięci DDRAM. Kursor przesuwa na pozycję zerową (pierwsza w pierwszej linii). Gwiazdka oznacza, że stan bitu może być dowolny.

Entry Mode Set (37 μs)

RS	R/W	DB7	DB6	DB5	DB4	DB3	DB2	DB1	DB0
0	0	0	0	0	0	0	1	I/D	S

I/D: wpisanie I/D = 1 oznacza *Increment*, I/D = 0 *Decrement*. Gdy I/D = 1 to: wpisanie (odczyt) do pamięci DDRAM inkrementuje adres AC tej pamięci i przesuwa kursor w prawo. Wpisanie (odczyt) do pamięci CGRAM inkrementuje adres AC tej pamięci. Gdy I/D = 0, adresy są dekrementowane, a kursor przesuwany w lewo.

S: S = 1 przesuwa wyświetlane znaki w prawo gdy I/D = 0 i w lewo gdy I/D = 1. Gdy S = 0 wyświetlany tekst nie jest przesuwany. Gdy S = 1, zapis (tylko zapis, odczyt nie) do DDRAM powoduje jednoczesne przesunięcie wyświetlanych znaków i kursora, czyli kursor pozostaje w tym samym miejscu wyświetlacza, ale przesuwa się względem tekstu. Zapis /odczyt CGRAM nie powoduje przesuwania tekstu na wyświetlaczu.

Display On/Off Control (37 μs)

RS	R/W	DB7	DB6	DB5	DB4	DB3	DB2	DB1	DB0
0	0	0	0	0	0	1	D	C	B

D: wyświetlacz jest włączony gdy D = 1 i wyłączony gdy D = 0. Wyłączenie D = 0 zachowuje dane w DDRAM i będą one wyświetlone ponownie po ustawieniu bitu D.

C: kursor jest wyświetlany gdy C = 1 i nie jest wyświetlany gdy C = 0. Wyzerowanie C = 0 powoduje tylko brak wyświetlania kursora i nie zmienia nic w funkcjonowaniu sterownika (np. nie zatrzymuje inkrementacji adresów przy wpisie do DDRAM). Kursor jest wyświetlany w postaci podkreślenia (pięć punktów w ósmej linii znaku 5×7).

B: znak wskazywany przez kursor miga gdy B = 1. Miganie oznacza, że przez około 0.5 sekundy wyświetlany jest znak, a przez kolejne 0.5 sekundy spacja. Bity C = 1 i B = 1 mogą być ustawione jednocześnie: wtedy będzie wyświetlony kursor i będzie migał znak.

Cursor or Display Shift (37 μs)

RS	R/W	DB7	DB6	DB5	DB4	DB3	DB2	DB1	DB0
0	0	0	0	0	1	S/C	R/L	*	*

S/C: S/C = 1 przesuwa tekst na wyświetlaczu, S/C = 0 przesuwa kursor

R/L: R/L = 1 przesuwanie w prawo, R/L = 0 przesuwanie w lewo

To polecenie przesuwa kursor albo wyświetlane znaki o jedną pozycję bez konieczności wpisywania (odczytywania) pamięci DDRAM. Kursor przejdzie na początek drugiej linii, gdy minie czterdziesty znak pierwszej. Przesuwanie wyświetlanych znaków powoduje jednoczesne i równoległe przesuwanie obu linii. Samo przesunięcie nie zmienia zawartości licznika adresów AC.

Function Set (37 μs)

RS	R/W	DB7	DB6	DB5	DB4	DB3	DB2	DB1	DB0
0	0	0	0	1	DL	N	F	*	*

DL ustawia szerokość magistrali danych, DL = 1 to 8 bitów (linie od DB0 do DB7), a DL = 0 to 4 bity (linie od DB4 do DB7). Sterownik budzi się w trybie 8-bitowym.

N ilość lini wyświetlacza. N = 0 jedna i N = 1 dwie linie.

F określa rozmiar znaków, F = 1 to 5 × 10 punktów i F = 0 to 5 × 8 punktów. Tą instrukcję należy wykonać w programie inicjalizacji wyświetlacza, przed wykonaniem innych instrukcji. Po inicjalizacji nie można zmienić ilości linii, ani rozmiaru znaków.

Set CGRAM Address (37 μs)

RS	R/W	DB7	DB6	DB5	DB4	DB3	DB2	DB1	DB0
0	0	0	1	A5	A4	A3	A2	A1	A0

Wpisanie tej instrukcji do sterownika wyświetlacza powoduje ustawienie adresu pamięci generatora znaków użytkownika CGRAM na wartość podaną w kodzie instrukcji.

Set DDRAM Address (37 μs)

RS	R/W	DB7	DB6	DB5	DB4	DB3	DB2	DB1	DB0
0	0	1	A6	A5	A4	A3	A2	A1	A0

Wpisanie tej instrukcji do sterownika wyświetlacza powoduje ustawienie adresu pamięci znaków wyświetlanych DDRAM na wartość podaną w kodzie instrukcji. Jeżeli bit N = 0 (jedna linia), adres musi być w zakresie od `0x00` do `0x4F`. Jeżeli bit N = 1 (dwie linie), adres musi być w zakresie od `0x00` do `0x27` dla pierwszej linii oraz od `0x40` do `0x67` dla drugiej linii.

Read Busy Flag and Address (0 μs)

RS	R/W	DB7	DB6	DB5	DB4	DB3	DB2	DB1	DB0
0	1	BF	A6	A5	A4	A3	A2	A1	A0

Polecenie odczytania rejestru instrukcji IR (poprzednio były same zapisy do IR). Odczytywany jest stan flagi *Busy* (DB7) oraz licznik adresu którejś z pamięci: CGRAM albo DDRAM. Który adres będzie odczytany zależy od tego, z którą pamięcią była wymiana

danych przed odczytem adresu. Jeżeli BF = 1 to sterownik HD44780 jest w trakcie wewnętrznych operacji i jedyną dozwoloną czynnością są kolejne odczyty *Read Busy Flag and Address*.

Write Data to CG or DDRAM (37 μs)

RS	R/W	DB7	DB6	DB5	DB4	DB3	DB2	DB1	DB0
1	0	D7	D6	D5	D4	D3	D2	D1	D0

Zapisz bajt do pamięci CGRAM abo do DDRAM. Zapis będzie wykonany do pamięci, której adres był ostatnio zapisany. Adres pamięci do której nastąpił zapis będzie inkrementowany /dektementowany w zależności od stanu bitu I/D w instrukcji *Entry Mode*. Ponadto, jeżeli bit S w tej samej instrukcji jest ustawiony i zapis jest do DDRAM, zostanie przesunięta pamięć znaków do wyświetlenia.

Read Data from CG or DDRAM (37 μs)

RS	R/W	DB7	DB6	DB5	DB4	DB3	DB2	DB1	DB0
1	1	D7	D6	D5	D4	D3	D2	D1	D0

Odczytaj bajt z pamięci CGRAM abo z DDRAM. Odczyt będzie wykonany z pamięci, której adres był ostatnio zapisany. Adres pamięci z której nastąpił odczyt będzie inkrementowany /dektementowany w zależności od stanu bitu I/D w instrukcji *Entry Mode*. Odczyt nie powoduje przesuwania znaków do wyświetlenia bez względu na stan bitu S.

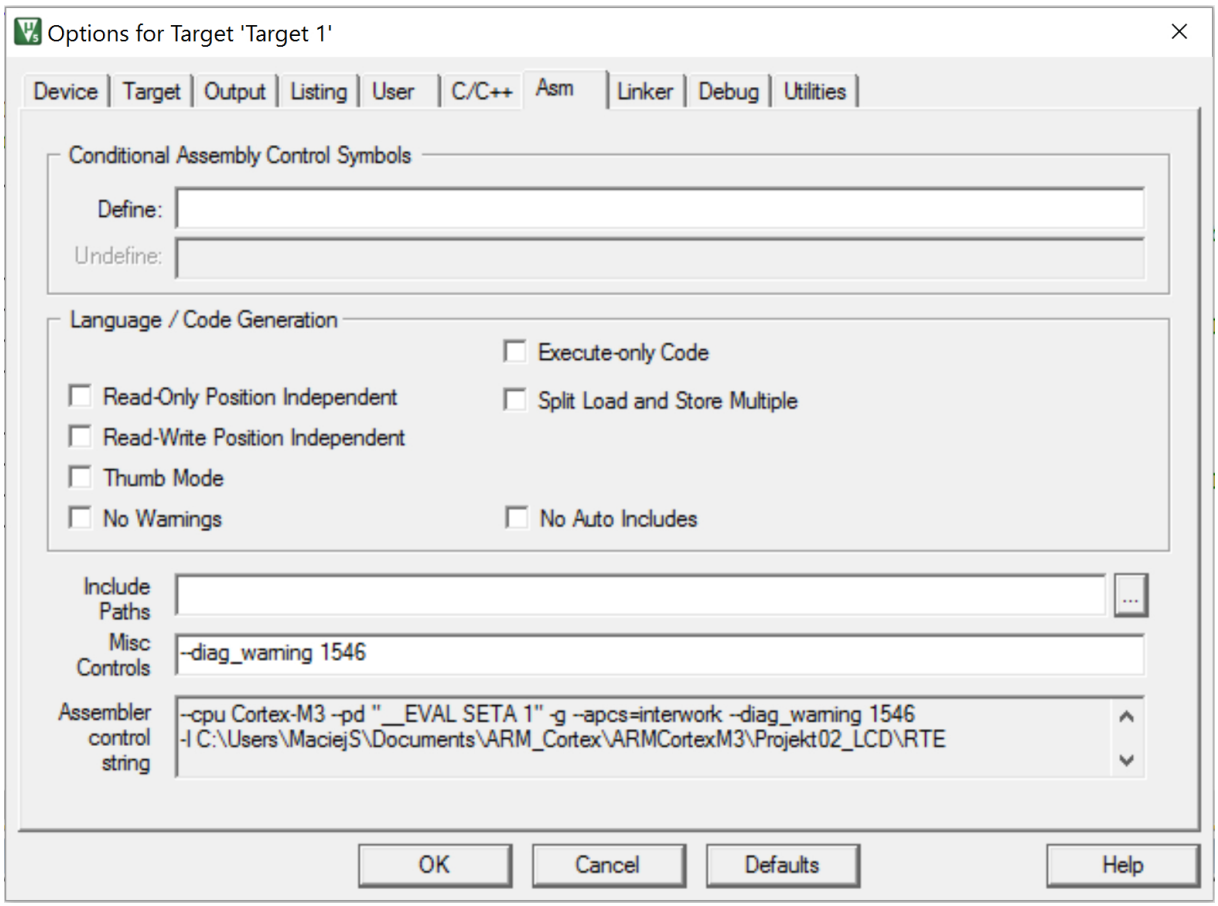
9.3. Projekt02_LCD

Uruchomimy wyświetlacz LCD 1602, który jest wyposażeniem opcjonalnym zestawu ZL27ARM. W głównym katalogu *ARM1CortexM3-ZL27ARM* zakładamy podkatalog *Drivers*, a w nim kolejny: *LCD1602*, w którym tworzymy trzy nowe (puste) pliki

lcd_1602.h – nagłówkowy
lcd_1602asm.s – kod asemblerowy
lcd_1602.c – źródłowy C

W głównym katalogu *ARM1CortexM3-ZL27ARM* kopiujemy ostatni projekt i zmieniamy nazwę kopii na *Projekt02_LCD*. Postępujemy analogicznie jak opisano w 4.11.1 (kasujemy pliki, zmieniamy nazwy, otwieramy *projekt02* w μVision, zmieniamy opcje projektu - trzeba pamiętać o zmianie ścieżki dostępu).

Zaczniemy od dołączenia plików z obsługą wyświetlacza do projektu. W oknie *Project* prawym klawiszem myszki klikamy na *Target 1* i wybieramy *Add Group...* Nazwę nowej grupy zmieniamy na *Drivers*. Prawym klawiszem myszki klikamy na *Drivers* i wybieramy *Add Existing Files to Group 'Drivers'* ... Dodajemy oba (.c i .s) pliki z katalogu *\ARM1CortexM3-ZL27ARM\Drivers\LCD1602*. Prawym klawiszem myszki klikamy na *Target 1* i wybieramy *Options fot Target 'Target1'*... W opcjach projektu, w zakładce *C/C++* musimy dodać ścieżkę dostępu do obu tych plików, inaczej nie będą widoczne w projekcie.



Rys. 9.2: Opcje asemblera

W tym projekcie będziemy łączyć programy asemblerowe z napisanymi w języku C. Kompilator C wymaga, aby stos był wyrównany do ośmiu bajtów. To oznacza, że zawsze trzeba odkładać na stos parzystą ilość rejestrów. Pisząc programy asemblerowe należy o tym pamiętać. Istnieje możliwość włączenia generacji komunikatów ostrzegających. Należy otworzyć opcje projektu i w zakładce asemblera *Asm* w oknie *Misc Controls* wpisujemy polecenie (rysunek 9.2)

–diag_warning 1546

Odłączamy moduł wyświetlacza LCD1602 od płyty ZL27ARM. Będziemy używać oscyloskopu gdyż musimy spełnić katalogowe wymagania czasowe sygnałów. Spełnienie tych wymagań jest bardzo ważne, gdyż chcemy zbudować niezawodny regulator. Najgorszy możliwy przypadek jest taki, że wymagania czasowe nie są spełnione, a mimo to układ działa pozornie poprawnie. W przyszłości, zmiana temperatury otoczenia lub wpływ innych czynników środowiskowych może spowodować brak komunikacji z wyświetlaczem i na przykład zawieszenie się regulatora z katastrofalnymi skutkami dla regulowanego procesu. Wymagania czasowe sygnałów operacji zapisu do wyświetlacza LCD1602 podane są na rysunku 9.3, a operacji odczytu, na rysunku 9.4. Zauważmy, że w obu przypadkach czas narastania /opadania zbocza nie może przekroczyć 25ns. Wartość tą musimy porównać z czasami portu GPIOx podanymi w tabeli 4.5. W przypadku maksymalnej częstotliwości przełączania 10 MHz czas zbocza (przy obciążeniu pojemnościowym 50 pF) jest równy 25 ns. Pojemności ścieżek na tak małej płytce nie przekraczają 10 pF, czyli w rzeczywistości zbocza będą krótsze niż 25 ns. Dlatego możemy zastosować częstotliwość przełączania portu GPIOC 10 MHz.

W pliku *lcd_1602asm.s* umieścimy funkcje asemblerowe obsługi wyświetlacza. Będą to podprogramy najbardziej "sprzętowe", czyli te, które bezpośrednio generują sygnały sterujące transmisją. Za pomocą oscyloskopu będziemy tak dobierali opóźnienia programowe, aby spełnić wymagania czasowe zawarte w katalogu producenta. Będą to podstawowe funkcje takie jak zapis znaku do wyświetlacza, czy oczekiwanie na stan zerowy flagi *Busy*. Dopiero po sprawdzeniu, że wszystkie sygnały są poprawne, dołączymy moduł wyświetlacza do płyty. Kolejny etap, to napisanie i uruchomienie programowej inicjalizacji wyświetlacza i zaprogramowanie jego sterownika odpowiednio do panelu LCD (dwie linie i znaki 5 × 7). Inicjalizację zakończymy wpisaniem ośmiu polskich znaków (małe litery ą, ć, ę, ł, ń, ó, ś, ż) do pamięci CGRAM sterownika HD44780. Po uruchomieniu funkcji asemblerowych napiszemy bardziej zaawansowane programy w kodzie źródłowym C. Prototypy funkcji asemblerowych i funkcji C umieścimy w jednym wspólnym pliku nagłówkowym.

```
;----------------- Definicje portów GPIOC -----------------
GPIOC_base  EQU 0x40011000   ; port C
GPIOx_CRL   EQU 0x00         ; port Configuration Register Low
GPIOx_CRH   EQU 0x04         ; port Configuration Register High
GPIOx_IDR   EQU 0x08         ; port Iput Data Register
GPIOx_ODR   EQU 0x0C         ; port Output Data Register
GPIOx_BSRR  EQU 0x10         ; port Bit Set/Reset Register
GPIOx_BRR   EQU 0x14         ; port Bit Reset Register

;---------------- Definicje pinów LCD1602 -----------------
LCD_RS      EQU 0x1000       ; 1:DATA, 0:INSTRUCTION REGISTER
LCD_RW      EQU 0x0800       ; 1:READ, 0:WRITE
LCD_E       EQU 0x0400       ; 1:ENABLE
LCD_b4b7    EQU 0x000F       ; bity danych b4-b5-b6-b7

            AREA lcd1602asm, CODE, READONLY
;------------------------ delay_us ------------------------
; opóźnienie programowe
; void delay(unsigned int counter);
; counter =      38 <-> 5 us (mikrosekund)
;         =      78 <-> 10 us
;         =     784 <-> 100 us
;         =  32'800 <-> 4.1 ms
;         = 120'000 <-> 15 ms
;         =     8e6 <-> 1 sekunda
            EXPORT delay_us
            ALIGN
delay_us    PROC
            B      label02
label01     NOP
            NOP
label02     CMP    r0,#0        ; czy zero na starcie ?
            SUB    r0,r0,#1
            BNE    label01
            BX     lr
            ENDP
```

Otwieramy plik *lcd_1602asm.s*. Zaczniemy od zdefiniowania stałych i napiszemy pierwszy ważny program, czyli opóźnienie programowe.

Musimy zaprogramować te linie portu GPIOC, które sterują wyświetlaczem (Tabela 9.1), są to `PC12,11,10,3,2,1,0`. Rejestry sterujące portów GPIO opisane są w dokumencie RM0008 (Tabela 4.2). Konfiguracja linii od `PC0` do `PC7` programowana jest w rejestrze `GPIOx_CRL` *port Configuration Register Low*, linii od `PC8` do `PC15` w rejestrze `GPIOx_CRH` *port Configuration Register High*. Wszystkie linie zaprogramujemy na wyjście *Push−Pull* 10 MHz. Bity CNF: `00`, MODE: `01`. Łącznie daje to 4 bity `0001` czyli `0x1`. Programowanie rejestrów konfigurujących: odczytamy → za pomocą maski wyzerujemy ... → kolejną maską ustawimy odpowiednie bity → na koniec zapiszemy. W ten sposób nie zmienimy konfiguracji pozostałych linii portu GPIOC. Programowanie linii GPIOC umieścimy na początku funkcji inicjującej wyświetlacz.

```
;------------------------ LCD_Initialize ------------------------
; programowa inicjalizacja wyświetlacza LCD1602
; inicjuje też odpowiednie linie portu GPIOC
; void LCD_Initialize(void);
                EXPORT LCD_Initialize
                ALIGN
LCD_Initialize  PROC
                PUSH   {r1-r3,lr}    ; 4 rejestry na stos (PRESERVE8)
                ; zaczynam od zaprogramowania portu GPIOC
                ; 10MHz, output Push-Pull <--> 0x1
                ; bity: CRH <-> 1C, CRL <-> 0F
                LDR    r2,=GPIOC_base    ; port GPIOC
                ; Configuration Register Low
                LDR    r1,=0xFFFF0000    ; maska AND
                LDR    r0,[r2,#GPIOx_CRL]
                AND    r0,r0,r1
                MOVW   r1,#0x1111        ; maska OR
                ORR    r0,r0,r1
                STR    r0,[r2,#GPIOx_CRL]
                ; Configuration Register High
                LDR     r1,=0xFFF000FF   ; maska AND
                LDR    r0,[r2,#GPIOx_CRH]
                AND    r0,r0,r1
                LDR    r1,=0x00011100    ; maska OR
                ORR    r0,r0,r1
                STR    r0,[r2,#GPIOx_CRH]
                ; wyzerujemy sygnały sterujące wyświetlacza
                MOVW   r1,#LCD_RS:OR:LCD_RW:OR:LCD_E:OR:LCD_b4b7
                STR    r1,[r2,#GPIOx_BRR] ; wyzeruj bity
                POP    {r1-r3,lr}
                ENDP

                END
```

Prototypy napisanych funkcji należy umieścić w pliku *lcd_1602.h*. W pliku *main.c* musimy dołączyć plik nagłówkowy

```
#include ".\..\Drivers\LCD1602\lcd_1602.h"
```

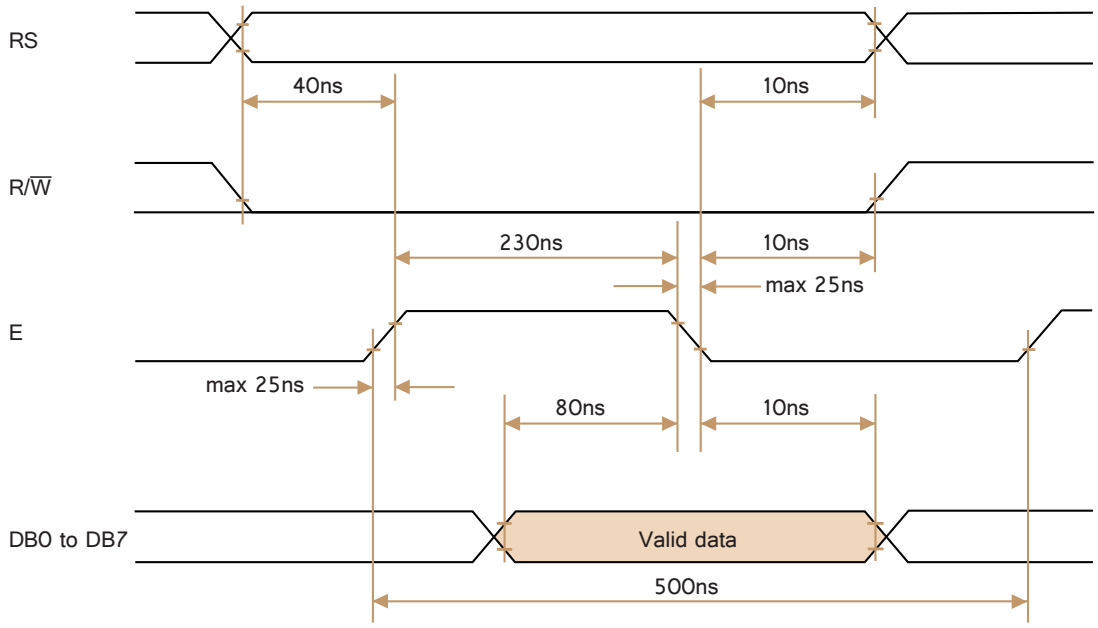
W funcji `main(void)` po konfiguracji RCC i GPIO umieścimy wywołanie funkcji

```
LCD_Initialize();
```

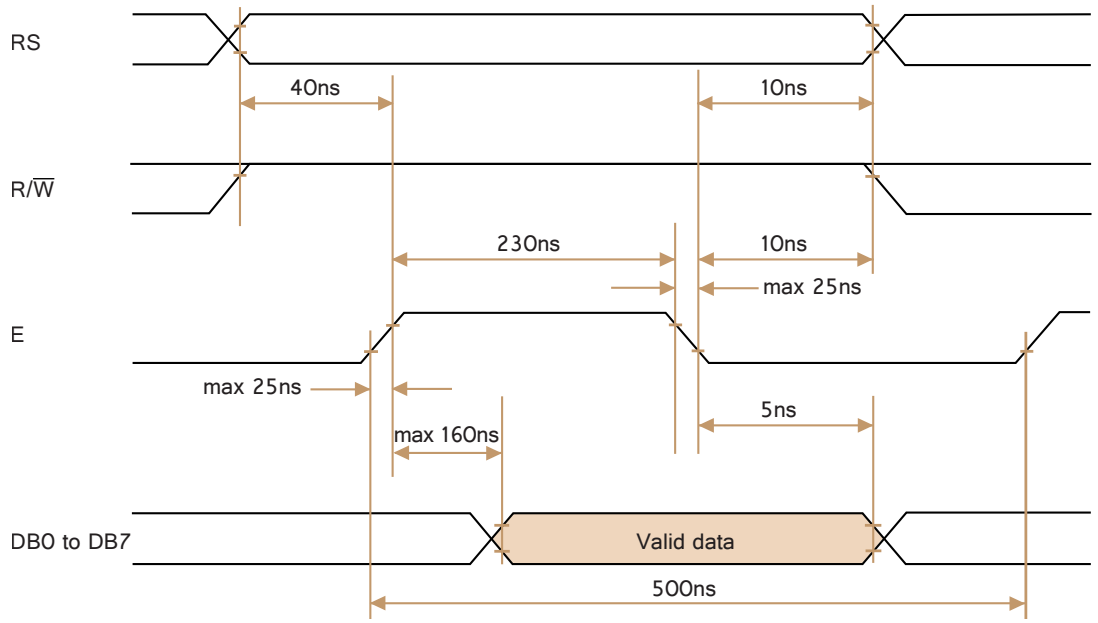
Ostatnią czynnością jest włączenie taktowania portu GPIOC w funkcji inicjalizacji RCC

```
RCC_APB2PeriphClockCmd(RCC_APB2Periph_GPIOC, ENABLE);
```

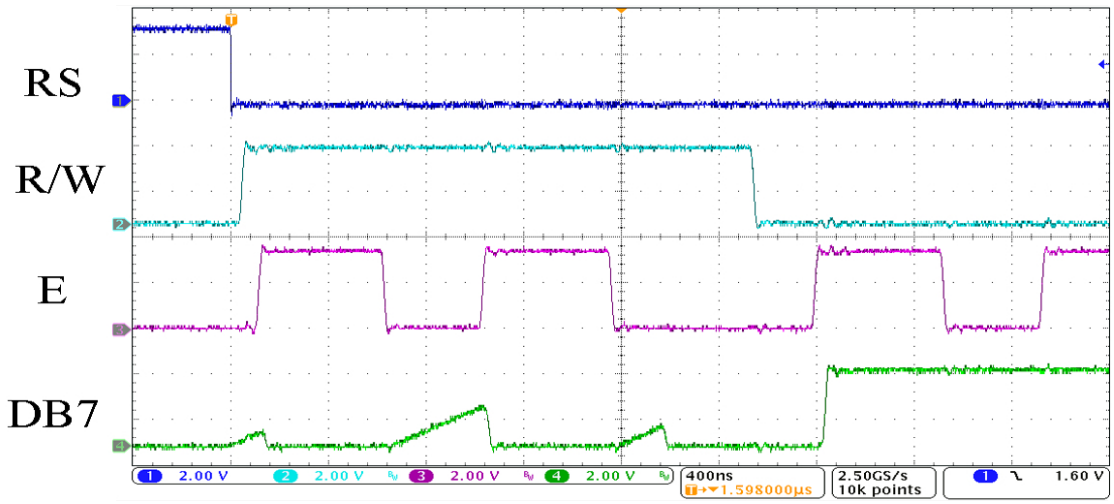
Teraz możemy wykonać test opóźnienia programowego. Należy napisać pętlę nieskończoną `PCx=1` → `delay_us` → `PCx=0` → `delay_us` →. Uzyskamy przebieg prostokątny na wybranej linii portu GPIOC. Częstościomierzem lub oscyloskopem możemy zmierzyć jego częstotliwość. Należy wyznaczyć wartości zmiennej `counter` odpowiadające czasom 15 ms, 4.1 ms oraz 100 μs. Te opóźnienia będą potrzebne w funkcji inicjalizacji wyświetlacza.



Rys. 9.3: Wymagania czasowe w operacji zapisu do LCD1602



Rys. 9.4: Wymagania czasowe w operacji odczytu z LCD1602



Rys. 9.5: Oscylogram zapisu do wyświetlacza

Napiszemy funkcję zapisu połówki bajtu (*nibble*) do wyświetlacza zgodnie z rysunkiem 9.3. Są tu dwa krytyczne czasy. Czas trwania stanu wysokiego sygnału E : minimum 230 ns, oraz czas całego cyklu zapisu: minimum 500 ns. Konieczne będzie odwrócenie kolejności bitów danych z `b3-b2-b1-b0` na `b0-b1-b2-b3`.

```
;--------------------- LCD_PutNibble --------------------
; zapisuje bity odwrócone b3-b2-b1-b0
; void LCD_PutNibble(unsigned char);
; INPUT: r0 = unsigned char
                EXPORT LCD_PutNibble
                ALIGN
LCD_PutNibble   PROC
                PUSH   {r1-r3,lr}          ; 4 rejestry na stos (PRESERVE8)
                LDR    r2,=GPIOC_base      ; port GPIOC
                MOV    r1,#LCD_E           ; bit ENABLE
                STR    r1,[r2,#GPIOx_BSRR] ; ustaw bit ENABLE = 1
                RBIT   r0,r0               ; odwrócenie kolejności bitów
                LSR    r0,r0,#28           ; przesuwa w prawo, wchodzą zera
                STR    r0,[r2,#GPIOx_BSRR] ; ustawia jedynki w danych
                MVN    r0,r0               ; neguje bit po bicie
                LSL    r0,r0,#28           ; przesuwa w lewo
                LSR    r0,r0,#28           ; przesuwa w prawo, wchodzą zera
                STR    r0,[r2,#GPIOx_BRR]  ; zeruje zera w danych
                MOVS   r0,#1
                BL     delay_us            ; opóźnienie minimum 230 ns
                STR    r1,[r2,#GPIOx_BRR]  ; wyzeruj bit ENABLE = 0
                POP    {r1-r3,pc}
                ENDP
```

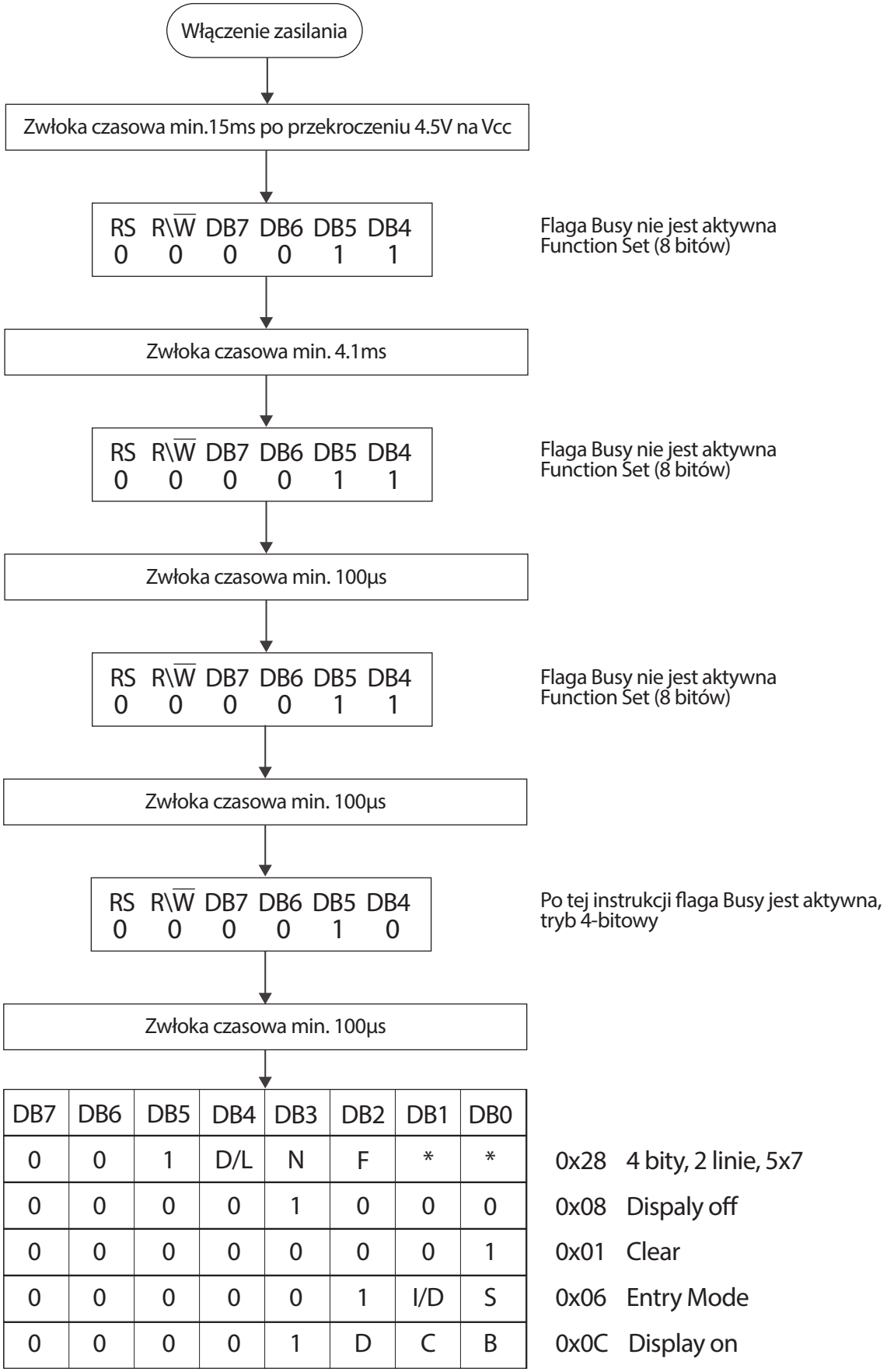
Oscylogram na rysunku 9.5 pokazuje odczyt *Busy Flag* oraz zapis do wyświetlacza. Sygnał ENABLE trwa około 500 ns (>230 ns), cykl odczytu /zapisu ponad 1000 ns. Kolejna ważna funkcja to odczytanie bitu zajętości *Busy Flag*. Wymagania czasowe operacji odczytu pokazane są na rysunku 9.4. podobnie jak poprzednio krytyczne są dwa czasy: minimalna długość trwania sygnału E to 230 ns oraz minimalny czas cyklu 500 ns.

```
;--------------------- LCD_ReadStatus --------------------
; po odczycie musi odwrócić kolejność bitów b3-b2-b1-b0
; unsigned char LCD_ReadStatus(void)
                ALIGN
LCD_ReadStatus  PROC
                PUSH  {r1-r3,lr}            ; 4 rejestry na stos (PRESERVE8)
                LDR   r2,=GPIOC_base        ; port GPIOC
                ; Configuration Register Low
                LDR   r1,=0xFFFF0000        ; maska AND
                LDR   r0,[r2,#GPIOx_CRL]
                AND   r0,r0,r1
                MOVW  r1,#0x4444            ; maska OR
                ORR   r0,r0,r1
                STR   r0,[r2,#GPIOx_CRL]    ; input High-Z
                ; RS = 0, R/W = 1
                MOV   r1,#LDC_RS
                STR   r1,[r2,#GPIOx_BRR]    ; wyzeruj RS = 0
                MOV   r1,#LDC_RW
                STR   r1,[r2,#GPIOx_BSRR]   ; ustaw R/W = 1
                NOP
                NOP                         ; czekamy 40 ns
                MOV   r1,#LDC_E
                STR   r1,[r2,#GPIOx_BSRR]   ; ustaw ENABLE = 1
                MOVS  r0,#1
                BL    delay_us              ; opóźnienie minimum 230 ns
                LDR   r3,[r2,#GPIOx_IDR]    ; port Input Data Register
                AND   r3,r3,#0x0F
                RBIT  r3,r3                 ; odwrócenie kolejności bitów
                LSR   r3,r3,#24             ; przesuwa w prawo
                STR   r1,[r2,#GPIOx_BRR]    ; wyzeruj ENABLE = 0
                MOVS  r0,#1
                BL    delay_us              ; opóźnienie minimum 270 ns
                STR   r1,[r2,#GPIOx_BSRR]   ; ustaw ENABLE = 1
                MOVS  r0,#1
                BL    delay_us              ; opóźnienie minimum 230 ns
                LDR   r0,[r2,#GPIOx_IDR]    ; port Input Data Register
                RBIT  r0,r0                 ; odwrócenie kolejności bitów
                LSR   r0,r0,#28             ; przesuwa w prawo
                ORR   r0,r0,r3              ; r0 = 000000FF
                STR   r1,[r2,#GPIOx_BRR]    ; wyzeruj ENABLE = 0
                ; Configuration Register Low
                LDR   r1,=0xFFFF0000        ; maska AND
                LDR   r3,[r2,#GPIOx_CRL]
                AND   r3,r3,r1
                MOVW  r1,#0x1111            ; maska OR
                ORR   r3,r3,r1
                STR   r3,[r2,#GPIOx_CRL]
                POP   {r1-r3,pc}
                ENDP
```

Podobnie jak w przypadku programu LCD_PutNibble użyjemy oscyloskopu, najlepiej czterokanałowego. W pętli nieskończonej będziemy wywoływać funkcję LCD_ReadStatus. Należy sprawdzić czy uzyskane przebiegi spełniają wymagania z rysunku 9.4. Jeżeli tak, to możemy zainstalować moduł wyświetlacza LCD1602. Napiszemy teraz dwa krótkie programy: wysłania komendy i zapisu znaku do wyświetlacza.

```
;--------------------- LCD_SendCommand --------------------
; void LCD_SendCommand(unsigned char);
                    EXPORT LCD_SendCommand
                    ALIGN
LCD_SendCommand     PROC
                    PUSH   {r1-r3,lr}           ; 4 rejestry na stos
                    MOV    r3,r0
label03             BL     LCD_ReadStatus
                    ANDS   r0,r0,#0x80
                    BNE    label03              ; czekamy na BUSY = 0
                    LDR    r2,=GPIOC_base       ; port GPIOC
                    MOV    r1,#LCD_RS:OR:LCD_RW ; bity RS, R/W
                    STR    r1,[r2,#GPIOx_BRR]   ; wyzeruj RS=0, R/W=0
                    LSR    r0,r3,#4             ; starszy nibble
                    BL     LCD_PutNibble
                    MOV    r0,r3
                    BL     LCD_PutNibble
                    POP    {r1-r3,pc}
                    ENDP
;--------------------- LCD_WriteCharacter --------------------
; void LCD_WriteCharacter(unsigned char);
                    EXPORT LCD_WriteCharacter
                    ALIGN
LCD_WriteCharacter  PROC
                    PUSH   {r1-r3,lr}           ; 4 rejestry na stos
                    MOV    r3,r0
label04             BL     LCD_ReadStatus
                    ANDS   r0,r0,#0x80
                    BNE    label04              ; czekamy na BUSY = 0
                    LDR    r2,=GPIOC_base       ; port GPIOC
                    MOV    r1,#LCD_RS
                    STR    r1,[r2,#GPIOx_BSRR]  ; ustaw RS = 1
                    MOV    r1,#LCD_RW
                    STR    r1,[r2,#GPIOx_BRR]   ; wyzeruj R/W = 0
                    LSR    r0,r3,#4             ; starszy nibble
                    BL     LCD_PutNibble
                    MOV    r0,r3
                    BL     LCD_PutNibble
                    POP    {r1-r3,pc}
                    ENDP
```

Jesteśmy gotowi do dokończenia inicjalizacji wyświetlacza. Algorytm inicjalizacji pokazany jest na rysunku 9.6. Dopiszemy inicjalizację do LCD_Initialize. Poniższe instrukcje należy wstawić przed indtrukcją POP {r1-r3,pc}.



DB7	DB6	DB5	DB4	DB3	DB2	DB1	DB0		
0	0	1	D/L	N	F	*	*	0x28	4 bity, 2 linie, 5x7
0	0	0	0	1	0	0	0	0x08	Dispaly off
0	0	0	0	0	0	0	1	0x01	Clear
0	0	0	0	0	1	I/D	S	0x06	Entry Mode
0	0	0	0	1	D	C	B	0x0C	Display on

Rys. 9.6: Algorytm inicjalizacji wyświetlacza LCD1602

```
            ; inicjalizacja wyświetlacza
            LDR     r0,=120000
            BL      delay_us          ; opóźnienie 15 ms
            MOVS    r0,#0x03          ; instrukcja 0x3
            BL      LCD_PutNibble
            MOVW    r0,#32800
            BL      delay_us          ; opóźnienie 4.1 ms
            MOVS    r0,#0x03          ; instrukcja 0x3
            BL      LCD_PutNibble
            MOV     r0,#784
            BL      delay_us          ; opóźnienie 100 us
            MOVS    r0,#0x03          ; instrukcja 0x3
            BL      LCD_PutNibble
            MOV     r0,#784
            BL      delay_us          ; opóźnienie 100 us
            MOVS    r0,#0x02          ; instrukcja 0x2
            BL      LCD_PutNibble
            MOV     r0,#784
            BL      delay_us          ; opóźnienie 100 us
            MOVS    r0,#0x28          ; dwie linie, 5x7
            BL      LCD_SendCommand
            MOVS    r0,#0x08          ; Display off
            BL      LCD_SendCommand
            MOVS    r0,#0x01          ; Clear Display
            BL      LCD_SendCommand
            MOVS    r0,#0x06          ; Entry Mode Set
            BL      LCD_SendCommand
            MOVS    r0,#0x0C          ; Display on
            BL      LCD_SendCommand
```

Kolejne funkcje napiszemy w pliku LCD_1602.c

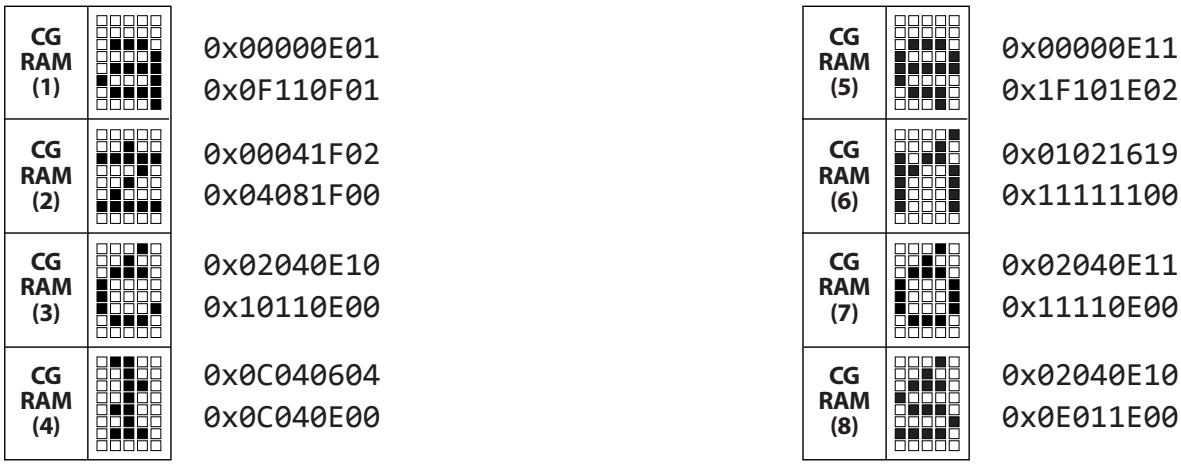
```
//*******************************************************************
#include "lcd_1602.h"
//-------------------------------------------------------------------
void LCD_WriteText(unsigned char * text) {
  while(*text)
    LCD_WriteCharacter(*text++);
}
//-------------------------------------------------------------------
void LCD_GoTo(unsigned char x, unsigned char y) {
  LCD_SendCommand(LCD_DDRAM_SET | (x + (0x40 * y)));
}
//-------------------------------------------------------------------
void LCD_WriteTextXY(unsigned char* text,unsigned char x,unsigned char y){
  LCD_GoTo(x,y);
  while(*text)
    LCD_WriteCharacter(*text++);
}
```

```
//-------------------------------------------------------------------
void LCD_SetUserChar(unsigned char chrNum,unsigned char n,unsigned char*p){
  //chrNum - od którego znaku zacząć (0 - 7)
  //n - ilość znaków do wpisania
  //*p - wskaźnik na tablicę z wzorcem znaków
  LCD_SendCommand(LCD_CGRAM_SET | chrNum * 8);
  n *= 8;
  do
    LCD_WriteCharacter(*p++);
  while (--n);
  LCD_SendCommand(LCD_CLEAR);     // przywraca adresy do DDRAM
}
//-------------------------------------------------------------------
void LCD_WriteBin(unsigned int var, int bitCount) {
  int i;
  for(i = (bitCount - 1); i >= 0; i--) {
    LCD_WriteCharacter((var & (1 << i))?'1':'0');
  }
}
//-------------------------------------------------------------------
```

W pliku *LCD_1602.h* umieszczamy prototypy tych funkcji, które będziemy używać oraz definicje instrukcji wyświetlacza, na przykład:

```
#define LCD_CLEAR        0x01
#define LCD_HOME         0x02
#define LCD_CGRAM_SET    0x40
#define LCD_DDRAM_SET    0x80
```

Możemy teraz, pisząc odpowiedni fragment programu w *main.c*, wyświetlić tekst na ekranie. Kontrast wyświetlacza należy ustawić za pomocą potencjometru CONTRAST obok modułu LCD1602. Możemy także wpisać polskie znaki do pamięci CGRAM.



Rys. 9.7: Polskie znaki

Rozdział 10

Programowanie strukturalne

Paradygmat programowania (*programming paradigm*) jest to wzorzec programowania, który definiuje sposób patrzenia programisty na przepływ sterowania (*control flow*) i wykonywanie programu. Na przykład w programowaniu obiektowym jest on traktowany jako zbiór współpracujących ze sobą obiektów.

Programowanie strukturalne (*structured programming*) opiera się na podziale kodu źródłowego na procedury i hierarchicznie ułożone bloki. Programowanie strukturalne zwiększa przejrzystość kodu i stanowi znaczną poprawę w stosunku do trudnego w utrzymaniu *spaghetti code*, którego nieczytelność wynika z częstego używania instrukcji `goto`. Początki programowania strukturalnego to lata 60 XX wieku. Impulsem do powstania programowania strukturalnego był list Edsgera Dijkstry *Goto statement considered harmful*. W liście tym napisał "Instrukcja *goto* powinna zostać usunięta ze wszystkich wysokopoziomowych języków programowania (to znaczy ze wszystkich poza - prawdopodobnie - czystym kodem maszynowym)". Edsger Dijkstra jest także twórcą algorytmu znajdowania najkrótszych ścieżek w grafie oraz problemu ucztujących filozofów.

Zgodnie z twierdzeniem o programowaniu strukturalnym, każdy program może być zbudowany z trzech podstawowych **struktur kontrolnych**

- **Sekwencja** – wykonanie ciągu kolejnych instrukcji.
- **Wybór** – rozgałęzienie programowe. W zależności od wartości logicznej wyrażenia (predykatu) wykonywana jest odpowiednia instrukcja, np. `if-else, switch`
- **Iteracja** – wykonanie określonego bloku instrukcji jest powtarzane dopóki wartość wyrażenia logicznego jest TRUE, na przykład `for, while, do-while`

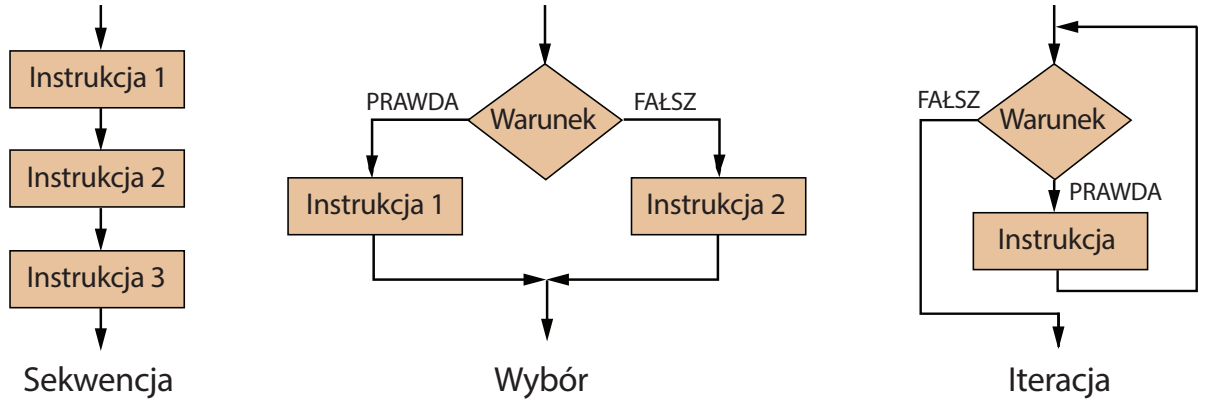
Zgodnie z definicją struktura kontrolna ma jedno wejście oraz jedno wyjście i może być traktowana jak jedna instrukcja. Struktury kontrolne mogą być zagnieżdżone (wbudowane), na przykład w strukturze sekwencji może być umieszczona struktura wyboru, wewnątrz której jest struktura iteracji.

Podprogramy (funkcje) pozwalają na wydzielenie grupy instrukcji, które są wykonywane wielokrotnie z różnymi **parametrami wywołania**.

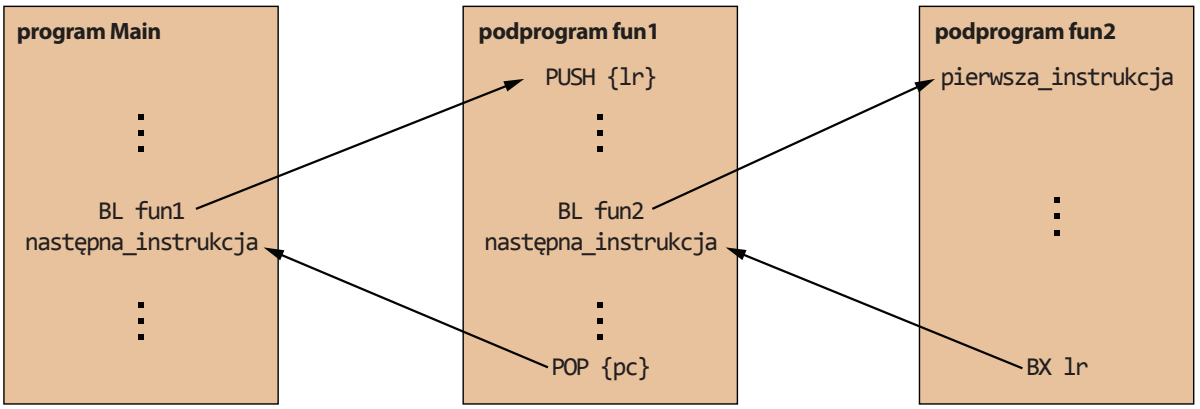
Bloki są to sekwencje instrukcji. Umożliwiają budowanie struktur kontrolnych. Bloki w języku C są ograniczone nawiasami klamrowymi (instrukcja blokowa).

10.1. Podprogramy

Podprogram nie powinien niszczyć (nadpisywać) zawartości rejestrów. Dlatego należy przesłać na stos (PUSH) rejestry używane w podprogramie. Przed instrukcją powrotu do programu głównego (wywołującego) należy pobrać ze stosu (POP) te wartości. Istnieje protokół (standard) przekazywania wartości parametrów do podprogramu i zwracania wartości przetworzonej z podprogamu do programu głównego. Ten protokół to *ARM*



Rys. 10.1: Struktury kontrolne



Rys. 10.2: Wywołania podprogramów

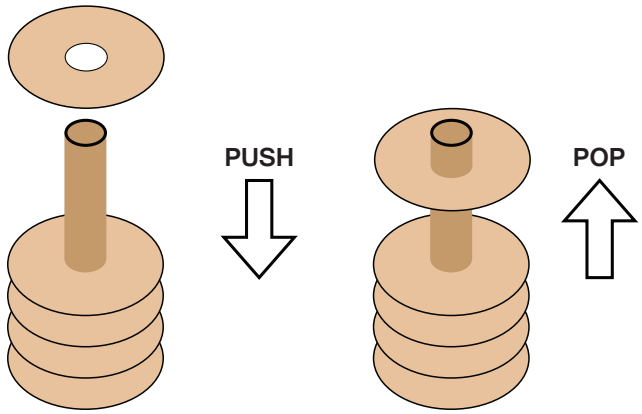
Embedded Application Binary Interface (EABI). Spełnienie wymogów EABI pozwoli na łączenie kodu asemblerowego z kompilatorem języka C.

Program asemblerowy wywołuje podprogramy za pomocą instrukcji `BL` *Branch and Link*. Instrukcja `BL` wykonuje dwie operacje:

1. Zapisuje do rejestru *Link Register* LR adres instrukcji następnej, za instrukcją wywołania `BL`. Ponieważ instrukcja `BL` jest 32-bitowa (= 4 bajty), to adres wpisany do `LR` jest równy PC + 4. Jest to **adres powrotu**.
2. Do licznika programu PC wpisuje adres pierwszej instrukcji podprogramu.

Są dwa sposoby powrotu z podprogramu

1. Jeżeli wywołany podprogram nie wywołuje innych podprogramów, to adres powrotu zapisany w `LR` pozostaje prawidłowy i powrót wykona instrukcja `BX lr` *Branch and Exchange*. Instrukcja `BX lr` przepisuje adres z rejestru `LR` do licznika programu `PC`.
2. Jeżeli podprogram wywołuje kolejny podprogram to drugie wywołanie spowoduje nadpisanie rejestru `LR`. W takim przypadku na początku podprogramu należy przesłać adres powrotu na stos `PUSH lr`. Ostatnią instrukcją podprogramu będzie `POP pc`. Nastąpi pobranie adresu powrotu ze stosu i wpisanie go do PC.



Rys. 10.3: Stos jest kolejką LIFO

10.2. Stos

Stos jest kolejką LIFO *Last-In-First-Out*. Ze stosem są związane dwie fundamentalne operacje: PUSH i POP. PUSH odkłada element na wierzch stosu, a POP pobiera ostatnio odłożony element, jak pokazano na rysunku 10.3. W zależności od kierunku narastania stos może być

- **Descending Stack** – schodzący w dół. Początek stosu to wysokie adresy, każda operacja PUSH zmniejsza wskaźnik stosu o 4. Tak jest w większości mikroprocesorów, także w ARM Cortex.
- **Ascending Stack** – wchodzący pod górę: początek to niskie adresy, każdy PUSH zwiększa SP o 4.

Ponadto stos może być "pełny" lub "pusty"

- **Full Stack** – gdy SP wskazuje na ostatnio zapisany element. PUSH najpierw zmienia SP, a później zapisuje na stos.Tak jest w ARM Cortex.
- **Empty Stack** – gdy SP wskazuje na pierwsze wolne miejsce. PUSH najpierw zapisuje na stos, a później zmienia SP.

Procesor ARM Cortex-M realizuje stos *full descending*.

10.3. Ochrona rejestrów w podprogramie

Zgodnie z wymaganiami protokołu EABI (*ARM Embedded Application Binary Interface*) podprogram nie może niszczyć (nadpisywać) zawartości rejestrów `r4 - r11` oraz `r13 (SP)`. Jeżeli te rejestry są używane w podprogramie, muszą na początku być odłożone na stos, a przed powrotem zdjęte, czyli muszą zostać odtworzone ich zawartości sprzed wywołania. Ten proces jest nazywany ochroną środowiska wywołującego (*preserving runtime environment*).

Natomiast podprogram nie musi chronić zawartości rejestrów roboczych `r0 - r3` (*scratch registers*) oraz rejestru `r12` (*Intra-Procedure-Call-register IP*). Jeżeli program główny ma istotne dane w tych rejestrach, to musi sam odłożyć je na stos przed wywołaniem podprogramu. Tabela 10.1 pokazuje przeznaczenie rejestrów.

Tabela 10.1: Przeznaczenie rejestrów procesora Cortex-M

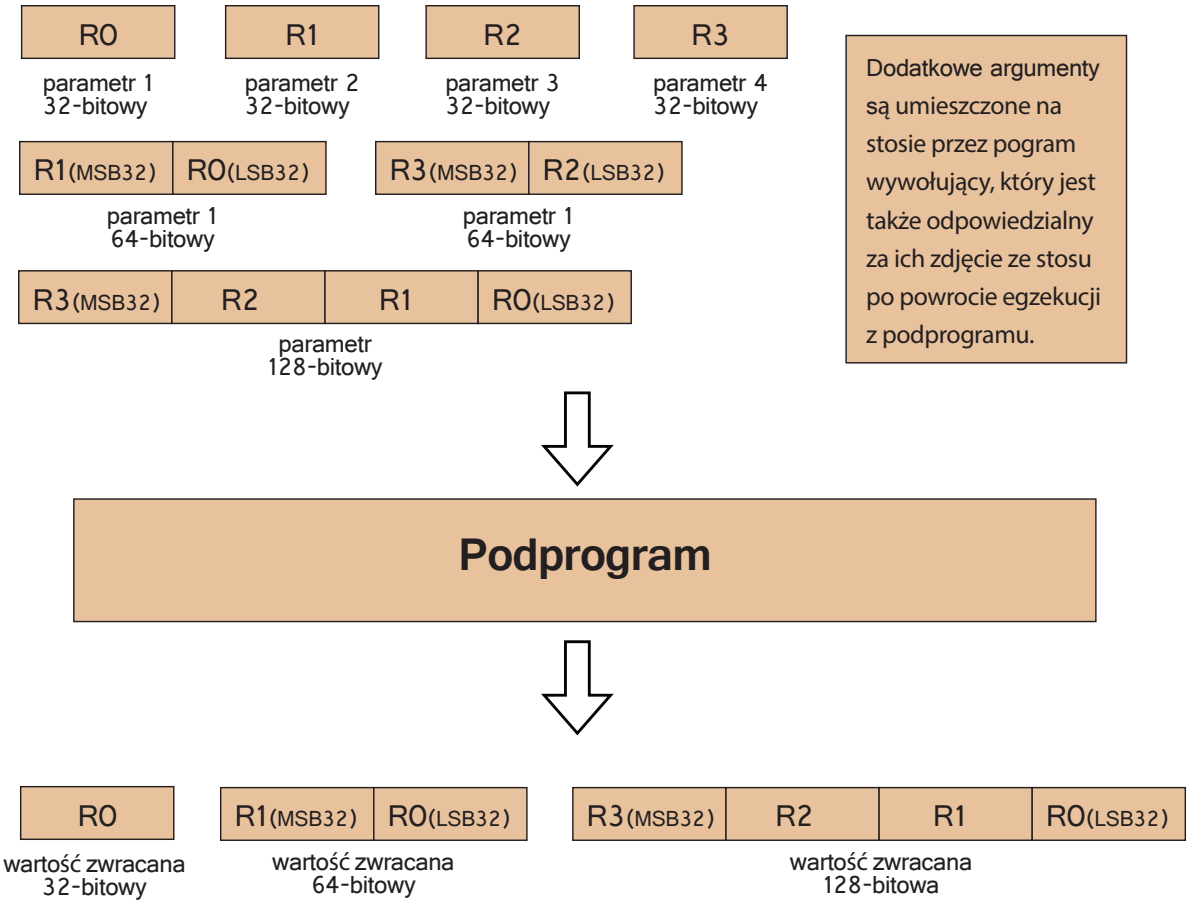
Rejestr	Przeznaczenie	Ochrona	Opis
r0	parametr 1	NIE	rejestry r0 - r3 przekazują parametry do podprogramu oraz wartości zwracane do programu głównego
r1	parametr 2	NIE	
r2	parametr 3	NIE	
r3	parametr 4	NIE	
r4	zmienna lokalna V1	TAK	rejestry r4 - r8 ogólnego przeznaczenia do przechowywania zmiennych lokalnych
r5	zmienna lokalna V2	TAK	
r6	zmienna lokalna V3	TAK	
r7	zmienna lokalna V4	TAK	
r8	zmienna lokalna V5	TAK	
r9	zależy od platformy	TAK	przeznaczenie zależy od platformy programowej
r10	zmienna lokalna V7	TAK	rejestry r10 - r11 ogólnego przeznaczenia do przechowywania zmiennych lokalnych
r11	zmienna lokalna V8	TAK	
r12	Inter-Procedure	NIE	może być zmieniony przy dalekim wywołaniu
r13	Stack Pointer	TAK	musi być zachowany po powrocie z podprogramu
r14	Link Register	NIE	może być zmieniony przez podprogram
r15	Program Counter	−−	w PC musi być adres powrotu

Podprogramy muszą spełnić następujące wymagania

- Oprócz rejestrów r0 - r3 podprogram nie musi chronić rejestru r12, który może być zmieniony przy dalekim wywołaniu oraz rejestru r14 (LR) (*Link Register*) - jeśli tylko powrót jest do właściwego adresu, na przykład za pomocą mechanizmu PUSH {lr} ...POP {pc}.
- Przeznaczenie rejestru r9 zależy od platformy programowej (środowiska IDE).
- Jeżeli program wywołuje podprogram i adres docelowy nie mieści się w zasięgu instrukcji BL, to asembler zamieni instrukcję BL etykieta na BLX r12 *Branch indirect with Link*, czyli w r12 będzie pełny, 32-bitowy adres początku wywoływanego podprogramu.

Instrukcje PUSH i POP muszą zawierać listę rejestrów. Kolejność na tej liście nie jest istotna. Procesor umieszcza rejestry na stosie w kolejności ich symboli tak, że rejestr o największym symbolu będzie umieszczony najgłębiej. Na przykład

Instrukcje PUSH i POP z listą rejestrów	Równoważny kod asemblerowy
PUSH {r4,r2,r6,lr} ; kolejność rejestrów na liście nie jest ; istotna, pierwszy na stos zostanie ; odłożony rejestr r14 (LR), następnie ; kolejno r6, r4 i r2	; kod równoważny PUSH {lr} PUSH {r6} PUSH {r4} PUSH {r2}
POP {r6,r2,lr,r4} ; kolejność rejestrów na liście nie jest ; istotna, pierwszy ze stosu zostanie zdjęty ; rejestr r2, następnie kolejno r4, r6 ; i na końcu r14 (LR)	; kod równoważny POP {r2} POP {r4} POP {r6} POP {lr}



Rys. 10.4: Wymiana danych pomiędzy programem wywołującym i podprogramem

10.4. Przekazywanie parametrów przez rejestry

Programy asemblerowe mogą być wywoływane przez funkcje języka C oraz przez programy niezależnie napisane przez innych programistów. Dlatego muszą spełniać wymagania protokołu ARM EABI także w zakresie wymiany danych z podprogramem. Protokół przekazywania parametrów do podprogramu jest opisany poniżej

- Gdy program główny przekazuje od jednego do czterech parametrów 32-bitowych, wykorzystywane są rejestry `r0 - r4` w kolejności: pierwszy parametr na liście wywołania w rejestrze `r0`, drugi w `r1`, trzeci w `r2` i czwarty w `r3`.
- Gdy program główny przekazuje od jednego do dwóch parametrów 64-bitowych (jak `long long` czy `double`), pierwszy parametr przesyłany jest w rejestrach `r0 - r1` (starsze słowo w `r1`), drugi w rejestrach `r2 - r3` (starsze słowo w `r3`).
- Program główny przekazuje jeden parametr 128-bitowy w rejestrach `r0 - r3` (najstarsze słowo w `r3`).
- Jeżeli parametrów do przesłania jest więcej, to pierwsze na liście przesyłane są w rejestrach `r0 - r3`, a kolejne (które nie zmieściły się w rejestrach) przesyłane są poprzez stos.

Protokół przekazywania wyników z podprogramu do programu głównego jest następujący

- Gdy podprogram zwraca jedną wartość 32 - bitową, to jest ona umieszczana w rejestrze `r0`.

- Wynik 64 - bitowy jest przekazywany w rejestrach `r0 - r1`.
- Wynik 128 - bitowy w rejestrach `r0 - r3`.

Rysunek 10.4 pokazuje wymianę danych z podprogramem za pomocą rejestrów.

Do podprogramu może być przekazana **wartość** parametru, albo **wskaźnik**, czyli adres miejsca w pamięci, gdzie jest przechowywany parametr. W pierwszym przypadku parametr jest kopiowany z pamięci do rejestru i podprogram jest wywoływany z tą kopią. Zmiany zmiennej wykonane w podprogramie nie mają wpływu na samą zmienną umieszczoną w pamięci. Jeżeli podprogram wykonuje operacje na stringu, czy tablicy danych, do podprogramu przekazywany jest wskaźnik, czyli adres początku stringu /tablicy. Podprogram działa wtedy nie na kopii, tylko na oryginalnych danych.

10.5. Projekt03_asm

10.5.1. Bit parzystości - przekazywanie wartości

W transmisji szeregowej z każdym bajtem może być przesyłany bit parzystości, który służy do kontroli poprawności transmisji. Jeżeli jest wybrana opcja *Odd*, to transmitowany bajt wraz z bitem parzystości mają łącznie parzystą ilość jedynek, gdy opcja *Even*, to nieparzystą. Napiszemy podprogram , który będzie zwracał jedynkę, gdy w bajcie jest nieparzysta ilość jedynek i zero gdy parzysta. Kopia bajtu, który ma zostać wytransmitowany, będzie przekazana do podprogramu w rejestrze `r0`. Wynik zostanie zwrócony także w `r0`.

Utworzymy nowy projekt `Projekt03_asm`. Dołączymy do niego dwa pliki: *CodeAsm.s* i *CodeAsm.h*. W pliku *CodeAsm.s* umieścimy podprogram, a w *AsmCode.h*. jego prototyp.

```
            AREA AsmCode, CODE, READONLY
;----------------- parityBit -----------------
;INPUT:  r0 = bajt (8 bitów)
;OUTPUT: r0 = 0 gdy parzysta ilość jedynek
;        r0 = 1 gdy nieparzysta ilość jedynek
; bit parzystości w transmisji szeregowej
            EXPORT parityBit
            ALIGN
parityBit   PROC
            MOVS   r1,#0            ; r1 = licznik jedynek
            MOVS   r2,#8            ; r2 = licznik pętli
            MOV    r3,r1            ; r3 = 0 do dodawania Carry
label01     LSRS   r0,r0,#1         ; przesuń bit b0 do Carry
            ADC    r1,r3            ; r1 = r1 + 0 + Carry
            SUBS   r2,r2,#1         ; dekrementuj licznik pętli
            BNE    label01          ; skocz gdy Z != 0
            MOVS   r0,#0
            LSRS   r1,r1,#1         ; bit b0 licznika jedynek do C
            BCC    label02          ; skocz, gdy C = 0
            ADC    r0,r0            ; r0 = 0 + 0 + 1
label02     BX     lr
            ENDP
```

W pliku *main.c* napiszemy program, który będzie binarnie wyświetlał liczby od `0x00` do `0xFF` i z każdą liczbą odpowiadający jej bit parzystości.

```
for(k=0; k < 256; k++) {
  LCD_SendCommand(LCD_HOME);
  LCD_WriteBin(k, 8);
  parity = parityBit(k);     // bit parzystości
  sprintf((char *)str2disp,"%2i",parity);
  LCD_WriteText(str2disp);
  delay_us(8000000);
}
```

10.5.2. Wyszukanie największej liczby - przekazywanie wskaźnika

Kolejny przykład to wyszukanie liczby największej w tablicy liczb całkowitych. Do podprogramu przekażemy wskaźnik na początek tablicy. Podprogram zwróci indeks liczby, czyli jej położenie (licząc od zera).

```
;---------------- maxLocation ----------------
;INPUT:  r0 = adres początku tablicy
;        r1 = długość tablicy
;OUTPUT: r0 = indeks największej liczby
            EXPORT maxLocation
            ALIGN
maxLocation PROC
            PUSH   {r3-r5,lr}            ; parzysta ilość rejestrów
            MOV    r2,r0                 ; adres początku tablicy
            LDR    r4,[r2]               ; r4 = 'maksymalna liczba'
            MOVS   r0,#0                 ; r0 = indeks 'maksymalnej'
            MOVS   r3,#1                 ; r3 = licznik pętli
            B      label05
label03     LDR    r5,[r2,r3,LSL #2]     ; r5 = kolejna liczba
            CMP    r5,r4                 ; czy kolejna większa od max
            BLE    label04               ; skocz, gdy nie
            LDR    r4,[r2,r3,LSL,#2]     ; kolejna jest największa
            MOV    r0,r3                 ; r0 = indeks największej
label04     ADDS   r3,r3,#1              ; i++ inkrementuj licznik pętli
label05     CMP    r3,r1                 ; czy koniec tablicy
            BCC    label03               ; skocz gdy nie
            POP    {r3-r5,pc}
            ENDP
```

Rozdział 11

Kontroler przerwań NVIC, SysTick i RTC

11.1. Przerwania

Przerwania (*Interrupts*) łączą mechanizm sprzętowy (*hardware*) i programowy (*software*) w celu zmuszenia procesora do przerwania bieżącej pracy i do wykonania określonego bloku instrukcji nazywanego programem obsługi *Interrupt Service Routine* (ISR). Program ISR jest odpowiedzią na wystąpienie zdarzenia zewnętrznego, które wymaga natychmiastowej reakcji systemu. Po zakończeniu ISR procesor kontynuuje wykonywanie bieżącego programu. Obsługa przerwania jest niewidoczna dla programu głównego. Przerwania mogą być generowane także przez sam procesor (*system exceptions*), a w mikrokontrolerach dodatkowo przez wewnętrzne układy peryferyjne.

Przerwania umożliwiają wykonywanie wielu zadań jednocześnie. W określonej chwili czasowej, procesor może wykonywać tylko jeden program. Ale okresowe przerwania pozwalają na współbieżną pracę np. dwóch procesów (praca z podziałem czasu). W systemie może funkcjonować wiele przerwań. Możliwe są różne scenariusze

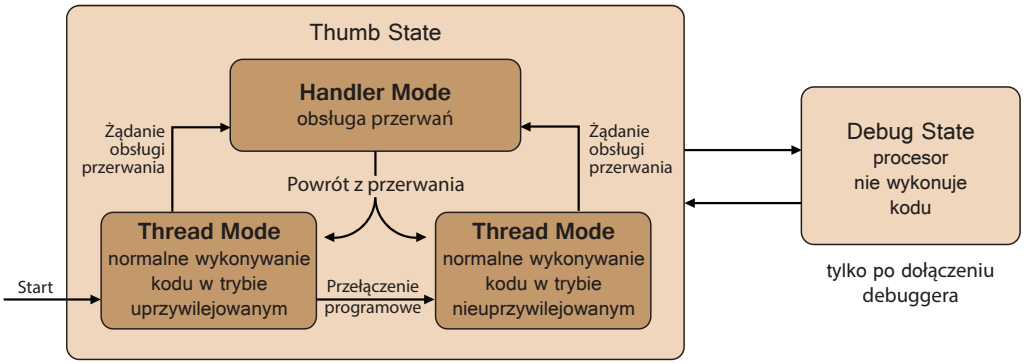
- Obsługa sekwencyjna – kolejne przerwanie (zgłoszone podczas obsługi) obsługiwane jest po zakończeniu bieżącego.
- Obsługa zagnieżdżona – po zgłoszeniu nowego przerwania obsługa bieżącego jest zawieszana i kontynuowana po obsłużeniu przerwania nowo zgłoszonego.
- Obsługa priorytetowa – zawieszenie obsługi bieżącego przerwania następuje tylko wówczas, gdy nowo zgłoszone ma wyższy priorytet, w przeciwnym razie obsługa następuje po obsłużeniu wszystkich przerwań o wyższych priorytetach.

Przeciwieństwem przerwań jest *polling*, czyli odpytywanie, gdy procesor ustawicznie sprawdza stan urządzeń zewnętrznych w celu określenia, czy któreś nie wymaga obsługi. Wykorzystując system przerwań procesor może realizować inne zadania, obsługując urządzenia zewnętrzne tylko w przypadku zaistnienia takiej konieczności.

Przykładem jest sytuacja, gdy czekamy na ważny telefon. Stosując odpytywanie podnosimy słuchawkę co 10 sekund w celu sprawdzenia czy na linii jest oczekiwana osoba. Stosując przerwania, możemy czytać książkę o systemach mikroprocesorowych oczekując na dzwonek telefonu. Ten dzwonek jest zgłoszeniem przerwania, które obsłużymy podnosząc słuchawkę.

11.2. Tryby i stany pracy procesora

Po RESET procesor budzi się w stanie wykonywania instrukcji z zestawu Thumb-2 (*Thumb State*), w trybie normalnego wykonywania kodu (*Thread Mode, thread = nitka*) i na poziomie uprzywilejowanego dostępu do zasobów (*Privileged Assess Level*). Aktywny



Rys. 11.1: Tryby i stany pracy procesora

jest wskaźnik stosu MSP (*Main Stack Pointer*) i to do niego wpisywana jest zawartość z adresu pamięci `0x00000000`. Jeżeli procesor jest na poziomie uprzywilejowanym to może zmieniać bity w rejestrze CONTROL. Wpisanie jedynki do bitu `SPSEL` tego rejestru przełącza wskaźnik stosu na PSP (*Process Stack Pointer*). UWAGA: wskaźnik PSP nie jest automatycznie inicjowany żadną wartością, programista musi zrobić to sam. Wpisanie jedynki do bitu `nPRIV` rejestru CONTROL zmienia poziom na nie-uprzywilejowany. Na tym poziomie nie można zmieniać zawartości rejestru CONTROL, czyli nie można przełączyć procesora na poziom uprzywilejowany ani zmienić wskaźnika stosu na MSP.

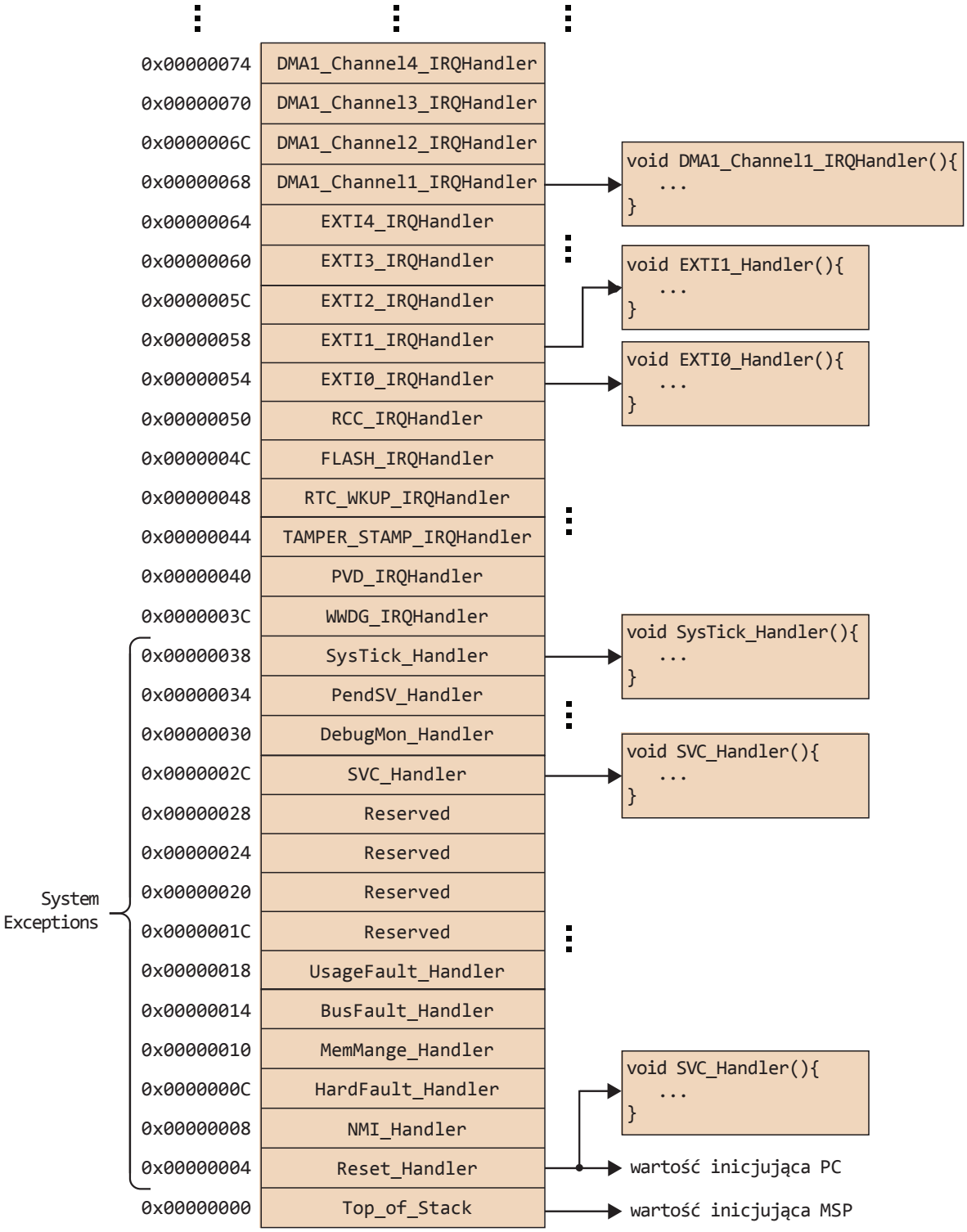
W czasie przejścia do obsługi przerwania procesor zmienia tryb pracy na *Handler mode*. W tym trybie używany jest główny wskaźnik stosu MSP i procesor pracuje na poziomie uprzywilejowanym. Programowy powrót do poziomu uprzywilejowanego jest więc możliwy jeżeli wykorzystamy mechanizm przerwań: będąc w obsłudze sztucznie możemy tak zmienić zawartości odpowiednich rejestrów, aby procesor po powrocie przeszedł do poziomu uprzywilejowanego.

Zewnętrzny programator może zmienić stan procesora na *Debug State*, w którym procesor przerywa wykonywanie instrukcji. UWAGA: większość rejestrów układu zarządzania przerwaniami NVIC może być zmieniana tylko z poziomu uprzywilejowanego.

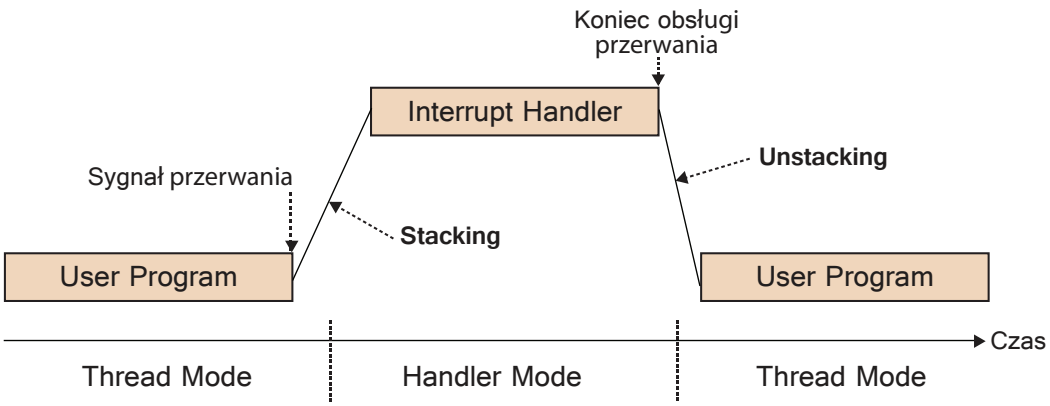
11.3. Program obsługi przerwania ISR

Interrupt Service Routine (ISR). Po załączeniu, lub sygnale RESET, procesor ARM Cortex pobiera dwa słowa z adresów `0x00000000` i `0x00000004`. Pierwsze (z adresu `0x00000000`) jest wpisywane do wskaźnika stosu *Main Stack Pointer* MSP, drugie (z adresu `0x00000004`) jest wpisywane do licznika programu PC. Słowo pamiętane pod adresem `0x00000004`) jest adresem programu *reset_handler()*, który konfiguruje obszary pamięci RAM i wywołuje program użytkownika *main()*. Po wpisaniu adresu do PC, procesor rozpoczyna wykonywanie programu od tego adresu.

Pierwsze słowo 32-bitowe jest zawartością wskaźnika stosu, drugie to adres początku programu. Kolejne stanowią tablicę wszystkich adresów obsług przerwań oraz wyjątków. Tablica adresów jest nazywana tablicą wektorów przerwań (*interrupt vector table*), a taki sposób organizacji to przerwania wektorowe. Procesor Cortex-M używa układu *Nested Vectored Interrupt Controller* (NVIC) do zarządzania przerwaniami. NVIC umożliwia aplikacjom na zezwalanie /zabranianie obsługi określonych przerwań oraz na nadawanie im priorytetów. Procesor obsługuje przerwania zgodnie z ich priorytetami (obsługa priorytetowa). Procesor przerywa bieżącą obsługę, jeżeli wystąpi nowe przerwanie o wyższym



Rys. 11.2: Tablica wektorów przerwań



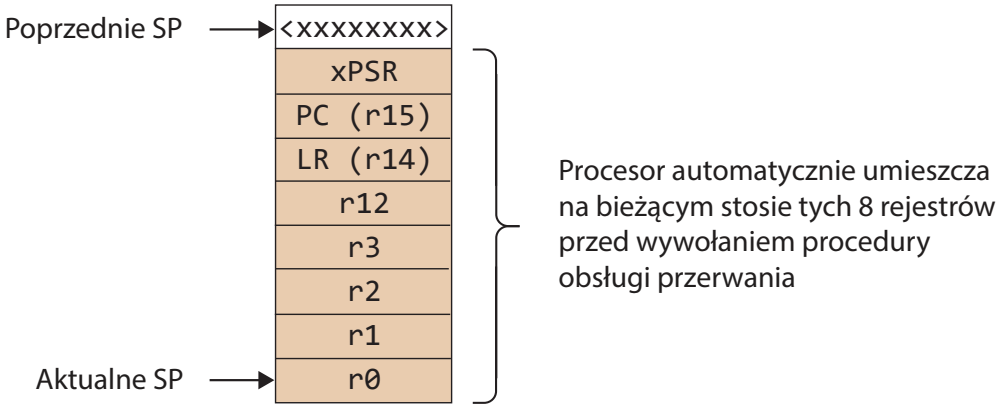
Rys. 11.3: Ochrona (*Stacking*) i odtworzenie (*Unstacking*) środowiska programowego w trakcie obsługi przerwania

priorytecie. Przerwana obsługa zostanie wznowiona, po zakończeniu obsługi o wyższym priorytecie. **Wyższa wartość numeru priorytetu oznacza niższy priorytet**. Przerwanie *Reset_Handler* ma najwyższy priorytet oznaczony jako −3.

W tablicy wektorów przerwań (rysunek 11.2), pierwszych piętnaście dotyczy obsługi różnych anomalii w działaniu systemu: są to wyjątki (*system exceptions*). Pozostałe (od szesnastego) to adresy obsług przerwań zewnętrznych (*external interrupts*).

Bezpośrednio przed rozpoczęciem wykonywania programu obsługi przerwania procesor automatycznie chroni na stosie środowisko przerywanego programu. Jest to *Interrupt Stacking*. Bezpośrednio przed powrotem to środowisko jest odtwarzane ze stosu: *Interrupt Unstacking*.

- **Interrupt Stacking**. Po przyjęciu przerwania, a przed wykonaniem pierwszej instrukcji programu obsługi, procesor wysyła na stos osiem rejestrów: cztery najmłodsze (`r0, r1, r2, r3`) oraz cztery bezpośrednio związane ze stanem procesora (`r12, LR, PSR, PC`), rysunek 11.4. Używa do tego stosu bieżącego, czyli MSP lub PSP w zależności od tego który jest używany przez przerwany program. Do `LR` zapisuje informację o bieżącym trybie pracy i przełącza się w *Handler Mode*, poziom uprzywilejowany i wskaźnik stosu MSP.
- **Interrupt Unstacking**. Po zakończeniu obsługi przerwania, automatycznie zdejmowane ze stosu są rejestry, które były tam umieszczone w fazie *stacking*.



Rys. 11.4: Rozmieszczenie rejestrów chronionych na stosie w operacji *Stacking*

Program obsługi przerwania wraca przez wykonanie instrukcji `BL lr`, jednak znaczenie rejestru `LR` w tym przypadku jest inne niż przy powrocie z normalnego podprogramu.

- `LR` w przypadku normalnego podprogramu zawiera adres powrotu, czyli adres instrukcji bezpośrednio za instrukcją wywołania. Zawartość `LR` jest kopiowana do `PC` i sterowanie wraca do programu wywołującego.
- W przypadku obsługi przerwania adres powrotu jest umieszczony na stosie, natomiast `LR` zawiera dodatkowe informacje o trybie pracy procesora w momencie przyjęcia przerwania.
  `LR = 0xFFFFFFF1` powrót do obsługi przerwania (*Handler Mode*) o niższym priorytecie, środowisko jest odtwarzane z wykorzystaniem wskaźnika MSP
  `LR = 0xFFFFFFF9` powrót do normalnego programu (*Thread Mode*), środowisko jest odtwarzane z wykorzystaniem wskaźnika MSP (*Main Stack*)
  `LR = 0xFFFFFFFD` powrót do normalnego programu (*Thread Mode*), środowisko jest odtwarzane z wykorzystaniem wskaźnika PSP (*Process Stack*)

11.4. Układ zarządzania przerwaniami NVIC

Układ *Nested Vectored Interrupt Controller* NVIC jest zintegrowany z rdzeniem procesora (rysunek 11.5) i zarządza wszystkimi przerwaniami. Udostępnia trzy podstawowe funkcjonalności

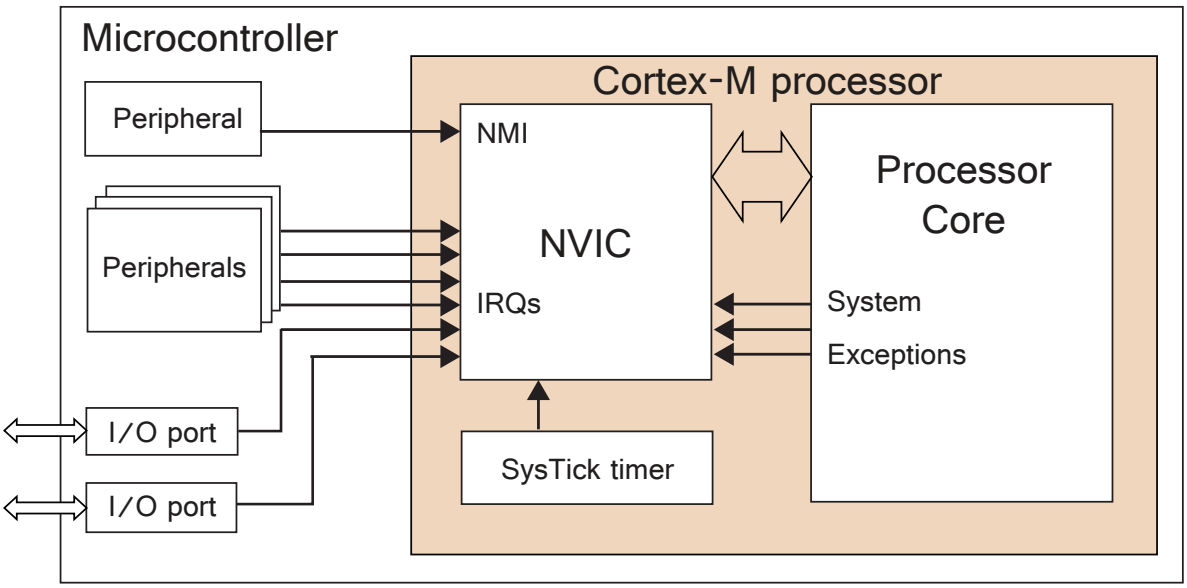
1. Zezwala (*Enable*) i zabrania (*Disable*) na obsługę przerwań - indywidualnie i grupowo.
2. Pozwala określić priorytet wywłaszczenia (*preemption*) i podpriorytet każdego przerwania.
3. Ustawia i zeruje bity stanu każdego przerwania indywidualnie.

Architektura ARMv7-M przewiduje obsługę aż do 256 przerwań. Jednak w procesorze STM32F103VB zrealizowano $60 + 16 = 76$, czyli przerwań zewnętrznych jest 60. Każde przerwanie ma pięć bitów sterujących:

Bit sterujący ...	**... w rejestrze (32-bitowym)**
Enable bit	Interrupt set enable register (`ISER`)
Disable bit	Interrupt clear enable register (`ICER`)
Pending bit	Interrupt set pending register (`ISPR`)
Un-pending bit	Interrupt clear pending register (`ICPR`)
Active bit	Interrupt active bit register (`IABR`)

Architektura ARMv7-M definiuje osiem rejestrów dla każdego bitu sterującego, na przykład 'Enable bit' jest w rejestrach `ISER0, ISER1, ..., ISER7`. Producent mikrokontrolera może zaimplementować mniejszą ilość przerwań. W procesorze STM32F103VB przerwań zewnętrznych jest 60, czyli na każdy bit potrzebne są trzy rejestry, np. `ISER0, ISER1, ISER3`.

- **Zezwalanie i zabranianie przerwań** Zapisanie jedynki do określonej pozycji rejestru `ISERx` powoduje, że odpowiadające tej pozycji przerwanie zostanie zezwolone. Rejestr można odczytać: każda jedynka oznacza zezwolenie na obsługę odpowiadającego jej przerwania. Wykaz numerów przerwań podany jest w dokumencie RM0008. Można zabronić obsługi przerwania wpisując jedynkę do odpowiedniej pozycji rejestru `ICERx`.



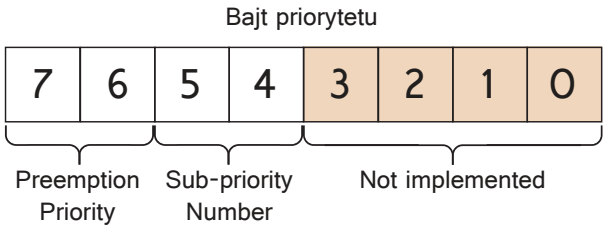
Rys. 11.5: Źródła przerwań w ARM Cortex-M

- **Przerwanie oczekujące (*Pending*)**. Jeżeli zostanie zgłoszone przerwanie, to mikrokontroler ustawia odpowiedni bit w rejestrze `ISPRx`, odczytanie tych rejestrów pozwala ustalić wszystkie przerwania oczekujące i będące w trakcie obsługi. Bit oczekiwania można ustawić programowo, wpisując jedynkę do odpowiedniej pozycji rejestru `ISPRx`. Wpisanie jedynki do odpowiedniej pozycji w rejestrze `ICPRx` kasuje oczekiwanie na obsługę przerwania. UWAGA: skasowanie bitu oczekiwania jest obowiązkiem programisty, należy wpisać jedynkę do właściwej pozycji rejestru `ICPRx` przed powrotem z obsługi przerwania.
- **Przerwanie aktywne**. Każdy zapalony bit w rejestrze `IABRx` oznacza, że odpowiadające mu przerwanie jest w trakcie obsługi. Rejestr `IABRx` jest tylko do odczytu, próba zapisu nie będzie miała żadnego skutku.

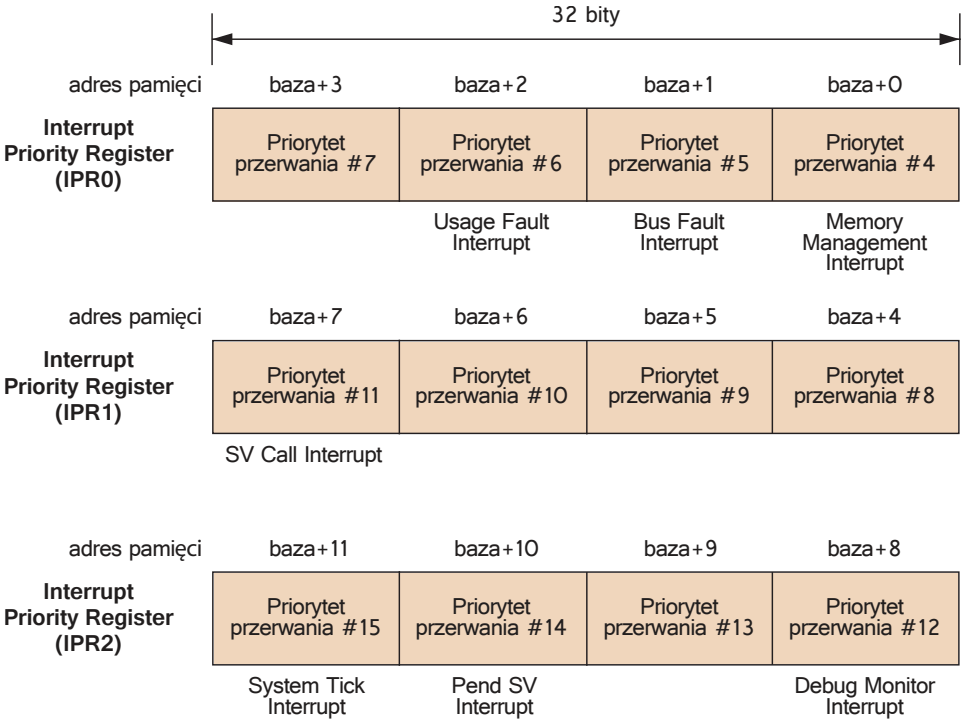
11.4.1. Priorytety przerwań

Priorytety określają kolejność obsługi przerwań. Każde przerwanie ma przydzielony rejestr priorytetu *Interrupt Priority Register* IP, o długości 8 bitów. Rejestr IP możemy podzielić na dwa pola. Pierwsze pole to *preemption priority*, czyli priorytet wywłaszczenia, drugie to *sub-priority number*, czyli pod-priorytet w obrębie priorytetu wywłaszczenia. Pamiętajmy, że **niższy numer to wyższy priorytet**. Przerwania od wewnętrznych peryferiali i przerwania zewnętrzne mają zawsze dodatnie priorytety. Natomiast trzy przerwania od rdzenia procesora mają priorytety ujemne (czyli najsilniejsze) i nie może to być zmienione programowo. Te trzy przerwania to: `Reset` (-3), `NMI` (-2) oraz `HardFault` (-1). Gdy jest wiele oczekujących przerwań, pierwsze będzie obsłużone to o najniższym numerze, czyli najwyższym priorytecie.

Wywłaszczanie jest stosowane, gdy w systemie istnieją szczególnie ważne przerwania, których obsługa nie może czekać. Jeżeli wystąpi zgłoszenie przerwania o wyższym priorytecie wywłaszczenia od obecnie obsługiwanego, to bieżąca obsługa zostanie przerwana (wywłaszczona) i procesor przejdzie do obsługi nowo zgłoszonego przerwania. Oczywiście po zakończeniu, będzie kontynuowana obsługa wywłaszczonego przerwania. Jeżeli nowe



Rys. 11.6: Bajt priorytetu w mikrokontrolerze STM32F103



Rys. 11.7: Rejestry priorytetów wyjątków systemowych

przerwanie ma równy (lub niższy) priorytet wywłaszczenia, to bieżąca obsługa nie będzie przerwana. Pod-priorytety służą do określenia kolejności obsługi gdy nastąpi zgłoszenie kilku przerwań o tym samym priorytecie wywłaszczenia. W skrajnym przypadku, gdy jest kilka przerwań o identycznych priorytetach (numerach w rejestrach IP), pierwsze jest obsługiwane to o najmłodszym adresie w tablicy wektorów (rys 11.2).

Architektura ARMv7-M przewiduje 8-bitowe rejestry priorytetu, jednak w procesorze STM32F103 cztery najmłodsze bity nie są wykorzystywane, więc na określenie priorytetu są przewidziane 4 bity. Te cztery bity możemy dowolnie podzielić na dwa pola, mamy więc pięć możliwości: $0 + 4, 1 + 3, 2 + 2, 3 + 1, 4 + 0$. Podziału dokonuje się wpisując odpowiednie bity do pola `PRIGROUP` w rejestrze *Application interrupt and reset control register* `SCB_AIRCR`. Szczegółowy opis jest w dokumencie PM0056. Przykład podziału $2 + 2$ rejestru priorytetu pokazany jest na rysunku 11.6. Procesor STM32F103VB ma 21 rejestrów *Interrupt priority registers* `IPRx`, pierwsze trzy pokazane są na rysunku 11.7.

11.4.2. Funkcje CMSIS do obsługi NVIC

Do programowania struktury przerwań wykorzystamy funkcje CMSIS. Są one zawarte w dwóch plikach, które już wcześniej dołączyliśmy do projektu. Te pliki to *misc.c* oraz

core_cm3.c. Definicje nazw poszczególnych przerwań są w pliku *stm32f10x.h*. Niektóre funkcje obsługi NVIC:

- Inicjalizacja NVIC zgodnie z danymi w strukturze:
  `void NVIC_Init(NVIC_InitTypeDef* NVIC_InitStruct)`

- Podział bajtu priorytetu na pola *preemption* i *sub-priority*:
  `void NVIC_PriorityGroupConfig(uint32_t NVIC_PriorityGroup)`

- Nadanie priorytetu obsługi określonemu przerwaniu :
  `void NVIC_SetPriority(IRQn_Type IRQn, uint32_t priority)`

- Zezwolenie na obsługę określonego przerwania:
  `void NVIC_EnableIRQ(IRQn_Type IRQn)`

- Zabronienie obsługi określonego przerwania:
  `void NVIC_DisableIRQ(IRQn_Type IRQn)`

- Zgaś bit oczekiwania na obsługę (trzeba wykonać na końcu obsługi):
  `void NVIC_ClearPendingIRQ(IRQn_Type IRQn)`

11.5. Zegar systemowy SysTick

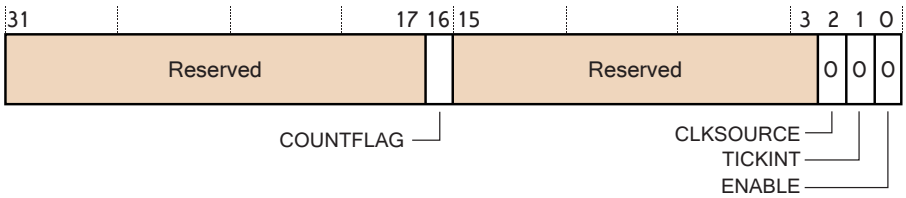
Zegar systemowy `SysTick` jest prostym licznikiem 24-bitowym przeznaczonym do odmierzania stałych interwałów czasowych. Najczęściej jest używany do generowania okresowo powtarzanego przerwania systemowego. Licznik zlicza od `N-1` do zera i generuje przerwanie, gdy osiągnie zero. Następnie ładuje wartość startową `N-1` z rejestru *Reload Value Register* i cykl jest powtarzany. `SysTick` nie przestaje odliczać także wtedy, gdy procesor jest zatrzymany, na przykład w czasie debugowania.

`SysTick` jest szczególnie użyteczny w systemach czasu rzeczywistego RTOS *Real Time Operating Systems*. Gdy jednocześnie jest wykonywanych kilka procesów, to procesor przydziela czas maszynowy poszczególnym procesom zgodnie z pewną założoną strategią. Na przykład algorytm karuzelowy (*round robin*) – najprostszy algorytm szeregowania procesów w systemie operacyjnym – przydziela przedziały czasu maszynowego kolejno wszystkim procesom, nie uwzględniając żadnych priorytetów. Jednak najczęściej stosowany jest algorytm EDF *Earliest Deadline First*, który do szeregowania zadań stosuje kolejkę priorytetową. Zawsze, gdy w systemie pojawi się niezajęty proces, z kolejki zostanie pobrane zadanie o najwyższym priorytecie, czyli najbliższe swego *deadline*. Optymalność algorytmu polega na tym, że jeśli EDF nie może uszeregować danego zbioru zadań, to żaden algorytm z dynamicznym przydziałem priorytetów nie znajdzie wykonalnego szeregowania oraz w sytuacji gdy jakikolwiek algorytm z dynamicznym przydziałem priorytetów potrafi uszeregować dany zbiór zadań, to również potrafi to EDF.

Ponieważ `SysTick` jest przewidziany do odmierzania czasu systemowego, jego rejestry są dostępne tylko w trybie uprzywilejowanym. Te rejestry to (producenci podają różne nazwy rejestrów, nazwy podane poniżej są zgodne z dokumentem PM0056):

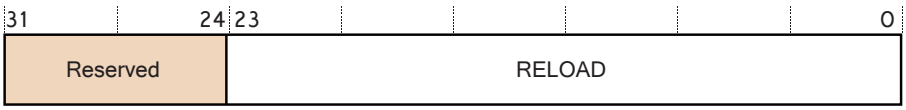
- `STK_CTRL` *SysTick Control and Status Register*

- STK_LOAD *SysTick Reload Value Register*
- STK_VAL *SysTick Current Value Register*
- STK_CALIB *SysTick Calibration Value Register*



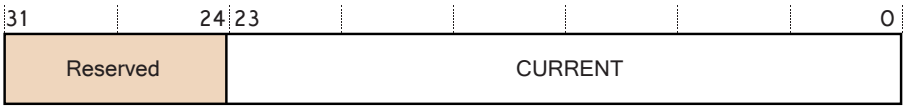
Rys. 11.8: SysTick Control and Status Register

- COUNTFLAG – czytany zwraca 1, gdy licznik przeszedł przez zero w okresie od ostatniego odczytu
- CLKSOURCE – wybiera źródło taktowania: 0 → AHB/8, 1 → AHB,
- TICKINT – zezwolenie na wystawianie przerwania przez SysTick : 0 → przerwanie zabronione, 1 → przerwanie zezwolone
- ENABLE – gdy bit ENABLE jest ustawiony, licznik jest ładowany wartością RELOAD i odlicza do zera. Jeżeli jednocześnie TICKINT = 1, to jest wystawiane przerwanie SysTick. Gdy bit ENABLE jest wyzerowany, licznik nie zlicza.



Rys. 11.9: SysTick Reload Value Register

Wartość RELOAD może być dowolną liczbą z zakresu od 0x00000000 do 0x00FFFFFF. Zero może być wpisane, ale nie wywoła przerwania, gdyż przerwanie jest generowane w czasie przejścia od wartości 1 do 0. Żeby wygenerować odstęp czasu równy dokładnie N cyklom zegara należy wpisać liczbę o jeden mniejszą, czyli $N - 1$, gdyż stan zerowy licznika trwa przez pełny cykl zegara. Dopiero w następnym takcie licznik ma wartość pobraną z rejestru STK_LOAD.



Rys. 11.10: SysTick Current Value Register

Bieżąca wartość licznika SysTick.

11.6. Projekt04_SysTick

W głównym katalogu *ARM1CortexM3-ZL27ARM* kopiujemy ostatni projekt i zmieniamy nazwę kopii na *Projekt04_SysTick*. Postępujemy analogicznie jak opisano w 4.11.1

(kasujemy pliki, zmieniamy nazwy, otwieramy *projekt04* w μVision, zmieniamy opcje projektu - trzeba pamiętać o zmianie ścieżki dostępu).

W tym projekcie uruchomimy przerwania systemowe `SysTick`. W obsłudze przerwań będziemy naprzemiennie włączać i wyłączać diodę LED8. Otwieramy plik *main.c*

```
/*****************************************************
 * ZL27ARM: Projekt04_SysTick                        *
 *****************************************************/
#include "stm32f10x.h"
#include <stdbool.h>    // true, false
#include <stdio.h>
#include ".\..\Drivers\LCD1602\lcd_1602.h"
int  RCC_Config(void);
void GPIO_Config(void);
void NVIC_Config(void);
int  SysTick_Config_Div8(unsigned int Ticks);

int main(void) {
   unsigned char str2lcd[] = "LED7: p\5tla main \nLED8: Systick\0";
   RCC_Config();
   GPIO_Config();
   NVIC_Config();
   GPIO_ResetBits(GPIOB, 0xFF00u);    // zgasimy wszystkie diody LED
   LCD_Initialize();                  // inicjalizacja wyświetlacza
   LCD_SendCommand(LCD_DISPLAY_ONOFF | LCD_DISPLAY_ON | LCD_CURSOR_ON;
   SysTick_Config_Div8(9000000ul);    // inicjalizacja i załączenie SysTick
   LCD_WriteText((unsigned char *)str2lcd);
   while(1) {
       GPIO_WriteBit(GPIOB, GPIO_Pin_14, Bit_SET);
       delay_us(4000000);
       GPIO_WriteBit(GPIOB, GPIO_Pin_14, Bit_RESET);
       delay_us(4000000);
   }
}
// ----------------- konfiguracja RCC: sygnały taktujące ----------------
int RCC_Config(void) {
   // konfiguracja RCC bez zmian, należy tylko umieścić:
   RCC_APB2PeriphClockCmd(RCC_APB2Periph_GPIOB, ENABLE); // LEDy
   RCC_APB2PeriphClockCmd(RCC_APB2Periph_GPIOC, ENABLE); // LCD1602
}
// ---------------------- konfiguracja portów GPIO ----------------------
void GPIO_Config(void) { -bez zmian - }


// --------------- konfiguracja kontrolera przerwań NVIC ----------------
void NVIC_Config(void) {
   NVIC_SetVectorTable(NVIC_VectTab_FLASH, 0x0);
}
```

```
// -------------- konfiguracja licznika systemowego SysTick -------------
int SysTick_Config_Div8(unsigned int Ticks) {
   unsigned int ctrl;
   if (Ticks > SysTick_LOAD_RELOAD_Msk) return 1; //błąd gdy Ticks>0xFFFFFF
   SysTick->LOAD = (Ticks & SysTick_LOAD_RELOAD_Msk) - 1; // LOAD = Ticks-1
   NVIC_SetPriority(SysTick_IRQn, 0);    // najwyższy priorytet przewania
   SysTick->VAL = 0;                    // aktualna wartość licznika
   ctrl= SysTick_CTRL_TICKINT_Msk | SysTick_CTRL_ENABLE_Msk;  // ustaw bity
   ctrl &= SysTick_CLKSource_HCLK_Div8;  // podział częstotliwości AHB/8
   SysTick->CTRL = ctrl;                // start licznika SysTick
   return 0;
}
```

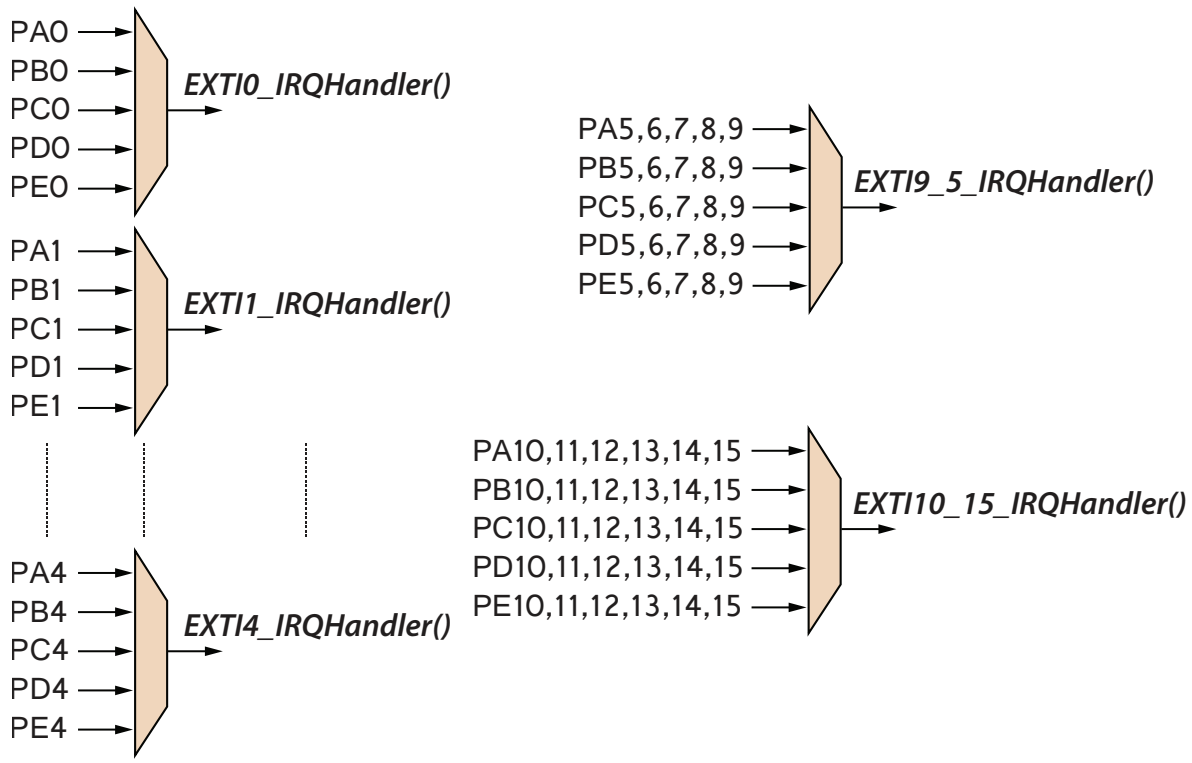
Podczas programowania `SysTick` należy podać początkową wartość licznika. Układ `RCC` zainicjowaliśmy tak, że procesor jest taktowany częstotliwością 72 MHz. Przerwanie ma być zgłaszane co jedną sekundę. Zgodnie z rysunkiem 4.15 `SysTick` jest taktowany z `AHB` częstotliwością 72 MHz. Zastosujemy podział tej częstotliwości przez osiem (`CLKSOURCE = 0`), czyli 72 / 8 = 9 MHz. Do rejestru `LOAD` musimy więc wpisać 9'000'000 - 1 = 8'999'999. Pozostaje teraz tylko napisanie obsługi przerwania zgłaszanego przez `SysTick`. W pliku *stm32f10x_it.c*, w okolicy linii 134, musimy wpisać

```
/**
   * @brief    This function handles SysTick Handler.
   * @param    None
   * @retval   None
   */
void SysTick_Handler(void) {
   uint8_t b;
   b = 1 - GPIO_ReadOutputDataBit(GPIOB, GPIO_Pin_15);
   GPIO_WriteBit(GPIOB, GPIO_Pin_15, (BitAction) b);
}
```

Po uruchomieniu, na wyświetlaczu będzie wyświetlony komunikat, dioda LED8 będzie migać z częstotliwością dokładnie 0.5 Hz (sekunda świecenia i sekunda przerwy), dioda LED7 będzie migać około dwa razy szybciej.

11.7. Przerwania zewnętrzne

Kolejnym naszym zadaniem jest dołączenie klawiatury. Wykorzystamy do tego celu przerwania zewnętrzne. Zewnętrzne źródła przerwań są obsługiwane przez układ `EXTI` *External interrupt /event controller*, który może zgłosić przerwanie (*interrupt*) lub zdarzenie (*event*). Zdarzenie może wywołać pewną reakcję, na przykład wybudzić procesor ze stanu uśpienia, ale nie powoduje wywołania programu obsługi. Niektóre układy mogą generować kilka zdarzeń, ale tylko jedno przerwanie. Na przykład w mikrokontrolerze STM32F103 są dwa przetworniki analogowo - cyfrowe. Każdy z nich może wygenerować trzy różne zdarzenia (*End of conversion, End of injected conversion, Analog watchdog event*), ale tylko jedno wspólne przerwanie *ADCinterrupt*. Program obsługi przerwania musi sprawdzić, które zdarzenie go wywołało.



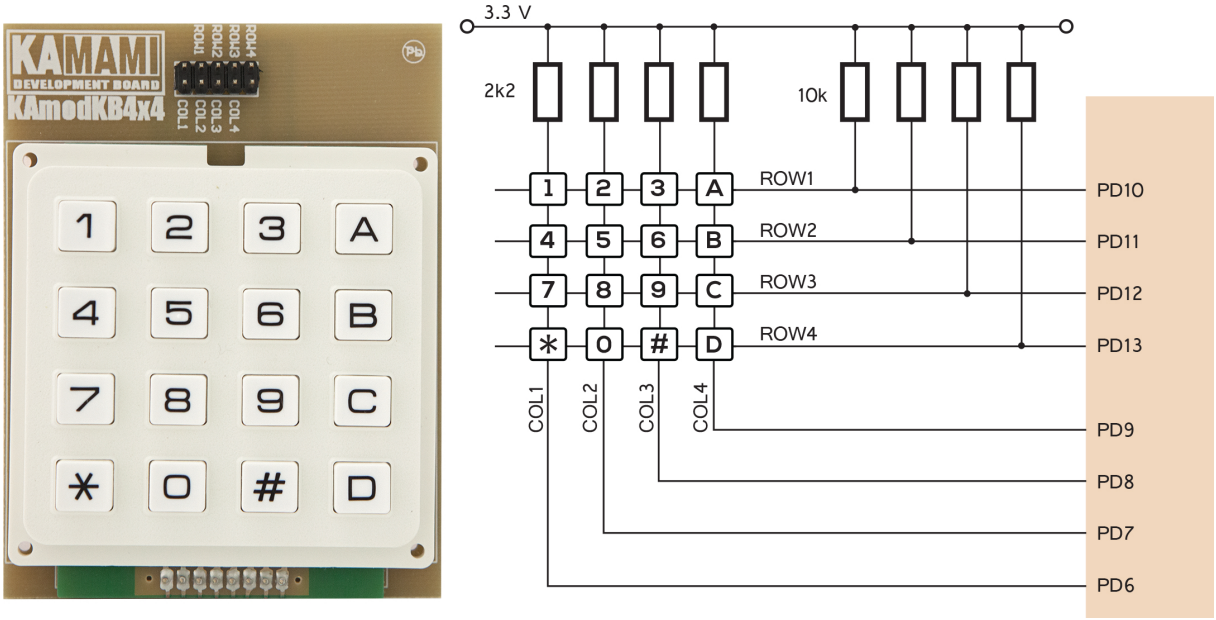
Rys. 11.11: Logiczne powiązania linii portów GPIOx i zgłaszanych przerwań

Moduł `EXTI` zawiera układy wykrywania zboczy. Każda linia może zostać indywidualnie zaprogramowana na zgłaszanie zdarzenia albo przerwania oraz na wykrywanie zbocza narastającego /opadającego lub obu. Każda linia może także być indywidualnie zamaskowana (wyłączona). Rejestr *pending* zawiera informację o liniach oczekujących na obsługę przerwania.

Przypisanie linii do zgłaszanych przerwań pokazuje rysunek 11.11. Linie od 0 do 4 poszczególnych portów zgłaszają, każda osobne przerwanie, ale wszystkie linie o tym samym numerze (na przykład `PA0, PB0, PC0, PD0, PE0`) zgłaszają to samo przerwanie (`EXTI0_IRQ`). Linie od 5 do 9 zgłaszają wspólne przerwanie `EXTI9_5_IRQ`. Podobnie linie od 10 do 15 zgłaszają wspólne przerwanie `EXTI10_15_IRQ`.

Każdej linni odpowiada jeden bit w rejestrach konfigurujących układu `EXTI`:

- `EXTI_IMR` *Interrupt mask register* – wpisanie jedynki zezwala, wpisanie zera zabrania na wystawianie przerwania.
- `EXTI_EMR` *Event mask register* – wpisanie jedynki zezwala, wpisanie zera zabrania na zgłaszanie zdarzenia.
- `EXTI_RTSR` *Rising trigger selection register* – wpisanie jedynki powoduje, że aktywne (wykrywane) jest zbocze narastające, zero wyłącza wykrywanie zbocza narastającego.
- `EXTI_FTSR` *Falling trigger selection register* – wpisanie jedynki powoduje, że aktywne jest zbocze opadające, zero wyłącza. Można wpisać jedynki do obu rejestrów, wtedy będą wykrywane oba zbocza.
- `EXTI_SWIER` *Software interrupt event register* – wpisanie jedynki (gdy bit był wyzerowany) powoduje ustawienie bitu *pending* w rejestrze `EXTI_PR`. Jeżeli dana linia nie jest zamaskowana, spowoduje to zgłoszenie przerwania.

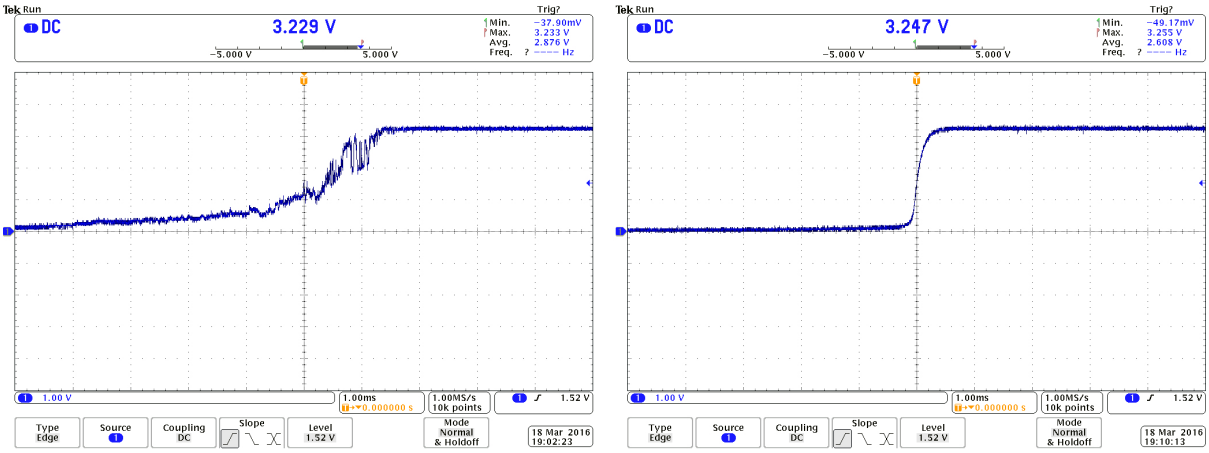


Rys. 11.12: Klawiatura matrycowa 4 × 4

- `EXTI_PR` *Pending register* – jeżeli bit odpowiadający określonej linii jest ustawiony, to znaczy, że linia ta zgłosiła przerwanie. Wpisanie jedynki kasuje oczekiwanie. Bit *pending* nie jest kasowany automatycznie, musi to zrobić programista przed zakończeniem programu obsługi.

11.8. Klawiatura matrycowa 4 × 4

Do płyty ZL27ARM dołączymy klawiaturę KAmodKB4x4, która ma 16 klawiszy ułożonych w matrycę 4 × 4 (rysunek 11.12). Układ klawiszy w postaci matrycy pozwala zredukować ilość linii portu I/O potrzebnych do odczytania stanu klawiatury. Wiersze ROWx dołączymy do linii portu GPIOD zgodnie z rysunkiem 11.12, czyli `ROW1` ↔ `PD10`, `ROW2` ↔ `PD11`, `ROW3` ↔ `PD12`, `ROW4` ↔ `PD13`. Linie te skonfigurujemy jako wyjścia. Kolumny COLy dołączymy (z opornikiem podciągającym 2.2 kΩ) do linii portu GPIOD: `COL1` ↔ `PD6`, `COL2` ↔ `PD7`, `COL3` ↔ `PD8`, `COL4` ↔ `PD9` i linie te skonfigurujemy jako wejścia wysokiej impedancji. W stanie oczekiwania na klawisz, wszystkie wyjścia ROWx, sterujące wiersze będą w stanie niskim, natomiast zewnętrzne oporniki 2.2 kΩ wymuszą stan wysoki wejść COLy. Naciśnięcie klawisza oznacza połączenie wiersza ROWx z kolumną COLy, co spowoduje wystąpienie zbocza opadającego na wejściu. Wykorzystamy przerwanie `EXTI9_5_IRQ`. W układzie `EXTI` załączymy rozpoznawanie zbocza opadającego, które sygnalizuje początek wciśnięcia klawisza. W obsłudze przerwania, wiersze ustawimy w stan wysoki i kolejno każdy wiersz będziemy załączać w stan niski. Po przełączeniu odczytamy wejścia `PD6 - PD9`, jeżeli na którymś wystąpi zero, to naciśnięty klawisz jest na przecięciu niskiego stanu wiersza i zera odczytanego z kolumny. W ten sposób trzeba zbadać kolejne wiersze aż do natrafienia na wciśnięty klawisz. Do zapamiętania ciągu wciśniętych klawiszy wykorzystamy bufor w pamięci o długości 32 znaków (długość obu linii wyświetlacza). Do tego bufora będziemy wpisywać kody ASCII kolejno wciskanych klawiszy.

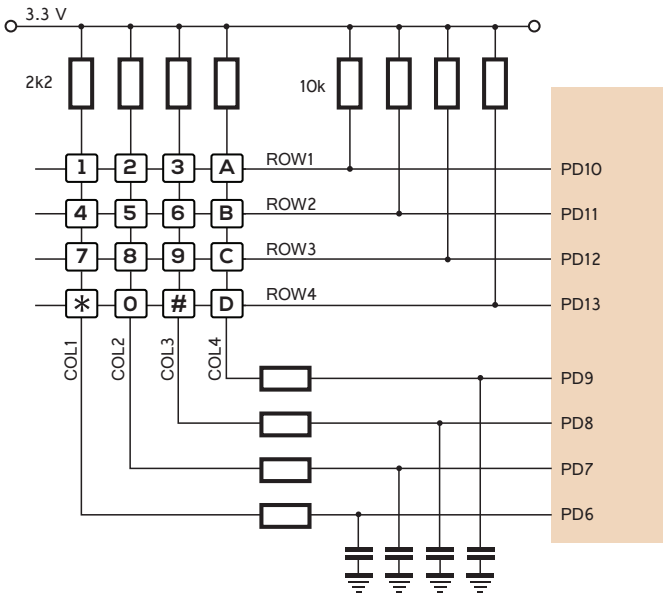


Rys. 11.13: Odbicia styków klawiatury

Jednoczesne wciśnięcie dwóch klawiszy z tej samej kolumny prowadzi do zwarcia wyjść sterujących wierszami. Na przykład jednoczesne wciśnięcie `1` i `4` zwiera wyjście `PD10` z `PD11`. Problem ten rozwiążemy konfigurując `PD10 - 13` jako wyjścia z otwartym drenem (*Open-Drain*). Ustalenie właściwych poziomów sygnałów zapewnią oporniki podciągające 10 kΩ.

Poważnym problemem przy odczytywaniu klawiatury są stany nieustalone. Są dwa takie momenty: pierwszy na początku (wciskanie) i drugi na końcu (zwalnianie). Oscylogramy napięcia na wejściu mikrokontrolera pokazane są na rysunku 11.13. Lewa strona rysunku to moment zwalniania klawisza w układzie podłączenia klawiatury jak na rysunku 11.12. Widać wyraźne oscylacje. Zjawisko to jest znane jako **odbicia styków**. Prostym zabiegiem sprzętowym jest filtr dolnoprzepustowy RC (układ całkujący). Na rysunku 11.15 przedstawiony jest schemat 11.12 uzupełniony o filtry RC z opornikami 220 Ω i kondesatorami 47 nF. Stała czasowa filtru jest równa $\tau = 220 * 47 * 10^{-9} = 1.034 * 10^{-5} \approx 10\mu s$. Prawa strona rysunku 11.13 pokazuje wejście `PD` po zastosowaniu filtrów RC. Kolejny problem z którym musimy się zmierzyć, to przesłuchy. Takie bezpośrednie dołączenie klawiatury do mikrokontrolera, bez układów przeciw-zakłóceniowych, za pomocą nieekranowanych przewodów, powoduje powstawanie przesłuchów pomiędzy liniami. Przełączenie wyjścia ze stanu wysokiego do niskiego (i odwrotnie) powoduje zaindukowanie impulsów w liniach sąsiednich. Pewną poprawę uzyskamy programując wyjścia `GPIO` na 2 MHz (rysunek 4.28), ale zjawiska przesłuchów w ten sposób nie zlikwidujemy. Oba problemy rozwiążemy programowo, wstawiając opóźnienie po każdej zmianie stanu wyjścia (wiersza). To opóźnienie musi być większe od czasu trwania najdłuższego stanu nieustalonego. Ze względu na konieczność stosowania opóźnień, klawiatura otrzyma najniższy priorytet przerwań.

Rys. 11.14: Klawiatura numeryczna



Rys. 11.15: Schemat dołączenia klawiatury z filtrami RC

11.9. Projekt05_keypad

W głównym katalogu *ARM1CortexM3-ZL27ARM* kopiujemy ostatni projekt i zmieniamy nazwę kopii na *Projekt05_keypad*. Postępujemy analogicznie jak opisano w 4.11.1 (kasujemy pliki, zmieniamy nazwy, otwieramy *projekt05* w μVision, zmieniamy opcje projektu - trzeba pamiętać o zmianie ścieżki dostępu).

Do projektu musimy dołączyć plik *stm32f10x_exti.c*. Klikamy prawym klawiszem myszki na podkatalogu *StdPeriphDrv* i wybieramy opcję *Add Existing Files to Group...* Dodajemy plik z katalogu *Libraries\STM32F10x_StdPeriph_Driver\src*.

W podkatalogu *Drivers* założymy nowy katalog *KAmodKB4x4*, obok katalogu *LCD1602*. W tym nowym katalogu utworzymy dwa pliki *KB_4x4.h* oraz *KB_4x4.c*. Do pliku nagłówkowego wpisujemy:

```
/****************************** KB_4x4.h ******************************
#include "stm32f10x.h"
#include ".\..\Drivers\LCD1602\lcd_1602.h"
extern unsigned char key_buffer[];    // tablica znaków z klawiatury
extern volatile unsigned int key_pointer;    // wskaźnik na tą tablicę
char KB2char (void);    // identyfikuje klawisz, kod ASCII -> key_buffer
```

Musimy dołączyć plik *lcd_1602.h* gdyż tam jest funkcja `delay_us()` która będzie potrzebna przy dekodowaniu klawiszy. Deklarujemy także dwie zmienne globalne (które zdefiniujemy w pliku *main.c*). Pierwsza to tablica na kody ASCII wciskanych klawiszy, druga zmienna to wskaźnik na pierwsze wolne miejsce w tej tablicy. Do pliku *KB_4x4.c* wpisujemy

```
/****************************** KB_4x4.c ******************************
#include ".\..\Drivers\KAmodKB4x4\KB_4x4.h"
const unsigned char KBkody[] = {'1','2','3','+','4','5','6','-','7','8',
                                                    '9','*','.','0',0x0D,'/',};
```

```
// odczytuje stan klawiatury, identyfikuje wciśnięty i zwraca jego kod
char KB2char (void) {
   unsigned int GPIO_Pin_row, GPIO_Pin_col, i, j;
   GPIO_SetBits(GPIOD, GPIO_Pin_10|GPIO_Pin_11|GPIO_Pin_12|GPIO_Pin_13);
   GPIO_Pin_row = GPIO_Pin_10;
   for (i=0; i<4; i++) {
       GPIO_ResetBits(GPIOD, GPIO_Pin_row);
       delay_us(32800);    // 4.1ms na ustalenie poziomów sygnałów
       GPIO_Pin_col = GPIO_Pin_6;
       for (j=0; j<4; j++) {
           if (GPIO_ReadInputDataBit(GPIOD, GPIO_Pin_col) == 0) {
               GPIO_ResetBits(GPIOD, GPIO_Pin_10|GPIO_Pin_11);
               GPIO_ResetBits(GPIOD, GPIO_Pin_12|GPIO_Pin_13);
               return KBkody[4*i+j];
           }
           GPIO_Pin_col = GPIO_Pin_col << 1;
       }
       GPIO_SetBits(GPIOD, GPIO_Pin_row);
       GPIO_Pin_row = GPIO_Pin_row << 1;
   }
   GPIO_ResetBits(GPIOD, GPIO_Pin_10|GPIO_Pin_11|GPIO_Pin_12|GPIO_Pin_13);
   return 0;
}
```

Będzie nam potrzebna klawiatura numeryczna, dlatego kody zostały przydzielone klawiszom jak na rysunku 11.14

Zmodyfikujemy plik *stm32f10x_it.c*. Zaczniemy od dołączenia pliku nagłówkowego

```
/* Includes ------------------------------------------------------------------*/
#include "stm32f10x_it.h"
#include ".\..\Drivers\KAmodKB4x4\KB_4x4.h"
```

Następnie wpiszemy program obsługi przerwania – w okolicach linii 414

```
/******************************************************************************/
void EXTI9_5_IRQHandler (void) {
   char c;
   c = KB2char();
   if (c != 0) {
       if (key_pointer > 31)
           key_pointer = 0;
       key_buffer[key_pointer] = c;
       key_pointer++;
   }
   // wyzeruj flagi oczekiwania na obsługę przerwania
   EXTI_ClearITPendingBit(EXTI_line6|EXTI_line7|EXTI_line8|EXTI_line9);
}
```

Teraz możemy zmodyfikować plik *main.c*. Musimy dołączyć plik nagłówkowy klawiatury. Poza ciałem funkcji `main()` zdefiniujemy zmienne globalne: obszar na znaki z klawiatury

wpisywane w obsłudze przerwania ze wskaźnikiem oraz dwa obszary, które wykorzystamy w następnym projekcie.

```
/*****************************************************
 * ZL27ARM: Projekt05_klawiatura                     *
 *****************************************************/
#include ".\..\Drivers\KAmodKB4x4\KB_4x4.h"

// zmienne globalne ----------------------------------
unsigned char key_buffer[32] = {0}; // bufor na znaki z obsługi przerwania
volatile unsigned int key_pointer = 0;    // wskaźnik do bufora
unsigned char txt2LCD[32] = {0};          // bufor funkcji sprintf
unsigned char KB2mem[32] = {0};           // bufor funkcji sscanf
```

W funkcji `main()` wypełnimy spacjami bufor `key_buffer` i w pętli nieskończonej będziemy wyświetlać jego zawarość, z odświeżaniem co sekundę. Każdy znak wpisany w obsłudze przerwania `EXTI9_5_IRQHandler()` zostanie wyświetlony po czasie nie dłuższym niż jedna sekunda.

```
int main(void) {
   unsigned int i;
   // konfiguracja systemu: bez zmian ----------------

   for (i=0; i<32; i++)
       key_buffer[i] = ' ';
   while (1) {
       LCD_GoTo(0,0);
       for (i=0; i<16; i++)
           LCD_WriteCharacter(key_buffer[i]);
       }
       LCD_GoTo(0,1);
       for (i=16; i<32; i++)
           LCD_WriteCharacter(key_buffer[i]);
       }
       GPIO_WriteBit(GPIOB, GPIO_Pin_14, Bit_SET);
       delay_us(4000000);
       GPIO_WriteBit(GPIOB, GPIO_Pin_14, Bit_RESET);
       delay_us(4000000);
   }
}
// ----------------- konfiguracja RCC: sygnały taktujące ----------------
int RCC_Config(void) {
   // konfiguracja RCC bez zmian, należy tylko umieścić:
   RCC_APB2PeriphClockCmd(RCC_APB2Periph_AFIO, ENABLE);  // AFIO
   RCC_APB2PeriphClockCmd(RCC_APB2Periph_GPIOB, ENABLE); // LEDy
   RCC_APB2PeriphClockCmd(RCC_APB2Periph_GPIOC, ENABLE); // LCD1602
   RCC_APB2PeriphClockCmd(RCC_APB2Periph_GPIOD, ENABLE); // klawiatura
}
```

W inicjalizacji `RCC` należy załączyć taktowanie układu AFIO (*Alternate-Function I/O*) oraz dodatkowo GPIOD (*General-Purpose I/O*), port D. W inicjalizacji `GPIO` musimy skonfigurować port D, linie `10-13` jako wyjścia *open-drain*, linie `6-9` jako wejścia *floating*.

```
// ------------ konfiguracja portów GPIO: należy dodać ---------------
GPIO_InitStructure.GPIO_Pin =GPIO_Pin_10|GPIO_Pin_11|GPIO_Pin_12|GPIO_Pin_13);
GPIO_InitStructure.GPIO_Speed = GPIO_Speed_2MHz;
GPIO_InitStructure.GPIO_Mode = GPIO_Mode_Out_OD;      // wyjście open-drain
GPIO_Init(GPIOD, &GPIO_InitStructure);
GPIO_InitStructure.GPIO_Pin =GPIO_Pin_6|GPIO_Pin_7|GPIO_Pin_8|GPIO_Pin_9);
GPIO_InitStructure.GPIO_Speed = GPIO_Speed_2MHz;
GPIO_InitStructure.GPIO_Mode = GPIO_Mode_IN_FLOATING; // wejście floating
GPIO_Init(GPIOD, &GPIO_InitStructure);
GPIO_ResetBits(GPIOD, GPIO_Pin_10|GPIO_Pin_11|GPIO_Pin_12|GPIO_Pin_13);
GPIO_AFIODeInit();
```

Najbardziej rozbudowana jest inicjalizacja układu `NVIC`. Wykonamy w niej konfigurację przerwań `EXTI` od linii od 6 do 9 oraz przerwania `EXTI9_5_IRQHandler()`

```
// --------------- konfiguracja kontrolera przerwań NVIC ----------------
void NVIC_Config(void) {
   NVIC_InitTypeDef NVIC_InitStructure;
   EXTI_InitTypeDef EXTI_InitStructure;
   NVIC_SetVectorTable(NVIC_VectTab_FLASH, 0x0); // wektory we FLASH
   NVIC_PriorityGroupConfig(NVIC_PriorityGroup_1); //grupowanie przerwań 2+2
   // linia 6 = COL1 -----------------------------------------------------
   GPIO_EXTILineConfig(GPIO_PortSourceGPIOD, GPIO_PinSource6); // źródło
   EXTI_InitStructure.EXTI_Line = EXTI_Line6; // wybór linii
   EXTI_InitStructure.EXTI_Mode = EXTI_Mode_Interrupt; // przerwanie
   EXTI_InitStructure.EXTI_Trigger = EXTI_Trigger_Falling;//zbocze opadające
   EXTI_InitStructure.EXTI_LineCmd = ENABLE;
   EXTI_Init(&EXTI_InitStructure);
   // analogicznie: linia 7 = COL2, linia 8 = COL3, linia 9 = COL4 --------

   // Wybór konfigurowanego IRQ ------------------------------------------
   NVIC_InitStructure.NVIC_IRQChannel = EXTI9_5_IRQn;     // wybór IRQ
   NVIC_InitStructure.NVIC_IRQChannelPreemptionPriority = 3;//prioryt.grupowy
   NVIC_InitStructure.NVIC_IRQChannelSubPriority = 3;    // podpriorytet
   NVIC_InitStructure.NVIC_IRQChannelCmd = ENABLE;  // włączenie obsługi IRQ
   NVIC_Init(&NVIC_InitStructure);
}
```

11.10. Projekt06_scanf

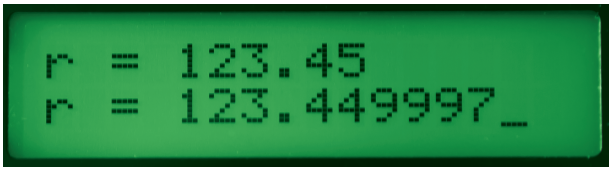
W głównym katalogu *ARM1CortexM3-ZL27ARM* kopiujemy ostatni projekt i zmieniamy nazwę na *Projekt06_scanf*. Naszym celem będzie uruchomienie funkcji standardowych `printf` i `scanf`. Zaczniemy od uzupełnienia pliku *KB_4x4.c*

```
// pobiera znak z bufora 'key_buffer', dosuwa znaki i wpisuje NUL na końcu
// wysyła echo na wyświetlacz
unsigned char get_char(void) {
   int i;
   unsigned char c;
   while (key_buffer[0] == 0)    // czekamy na klawisz
       ;
   c = key_buffer[0];
   NVIC_DisableIRQ(EXTI9_5_IRQn);//zabrania na przerwania EXTI od klawiszy
   for(i=0; i<31; i++)
       key_buffer[i] = key_buffer[i+1];
   key_pointer--;
   key_buffer[31]=0;
   NVIC_EnableIRQ(EXTI9_5_IRQn); // zezwala na przerwania EXTI od klawiszy
   if(c != 0x0D)
       LCD_WriteCharacter(c);    // echo
   return c;
}
// pobiera znaki i wpisuje do bufora 'KB2mem' --------------------------
void get_str(void) {
   int i;
   unsigned char c;
   for(i=0; i<31; i++)
       KB2mem[i] = ' ';
   for(i=0; i<31; i++) {
       c = get_char();
       if(c == 0x0D)
           break;
       else
           KB2mem[i] = c;
   }
}
// wyświetla komunikat, pobiera znaki i wpisuje do bufora 'KB2mem' ------
void LCD_getString(unsigned char * text) {
   LCD_WriteText((unsigned char * text);
   get_str();
}
```

W pliku nagłówkowym musimy umieścić prototypy dopisanych funkcji. Teraz możemy wykonać test działania funkcji `printf` i `scanf`. Będziemy stosować wersje z przedrostkiem `s`. Funkcja `sscanf` pobiera znaki ze wskazanego bufora pamięci. Najpierw wywołamy funkcję `LCD_getString()`, która umieści string pobrany z klawiatury (z echem na wyświetlaczu) w buforze pamięci, a następnie `sscanf()` ze wskaźnikiem na ten bufor

```
float r;

   LCD_getString((unsigned char *)"r = ");
   sscanf((const char *) KB2mem, "%f", &r);

   LCD_GoTo(0,1);
   sprintf((char *)txt2LCD,"r = %f",r);
   LCD_WriteText(txt2LCD);
```

Funkcja `sprintf` tworzy string zgodnie z podanym ciągiem formatującym i umieszcza go w buforze pamięci. Funkcja `LCD_WriteText()` przepisuje zawartość tego bufora do wyświetlacza. Przykład działania programu:



11.11. Zegar czasu rzeczywistego RTC

Zegar czasu rzeczywistego RTC (*Real Time Clock*) jest wydzielonym układem wewnątrz mikrokontrolera. Charakteryzuje go bardzo mały pobór energii, dzięki czemu może pracować z dołączonej zewnętrznej baterii także wtedy, gdy cały mikrokontroler jest wyłączony.

11.11.1. Czas UNIXowy

Czas UNIXowy jest to system reprezentacji czasu jako liczby sekund, które upłynęły od godziny 00:00:00 (północ) 1 stycznia 1970 roku według czasu UTC (*Coordinated Universal Time*), czyli od chwili zwanej początkiem epoki UNIXa, dlatego w literaturze angielskojęzycznej jest to ***UNIX Epoch Time***. Historycznie, ilość sekund jest 32-bitową liczbą ze znakiem, czyli najdawniejszym czasem, który może być przedstawiony jest -2^{31}, a to oznacza 20:45:52 UTC, 13 grudnia 1901. Patrząc w przyszłość wartością graniczną jest $2^{31}-1$, czyli 03:14:07 UTC 19 stycznia 2038. Wprawdzie coraz więcej urządzeń korzysta z 64-bitowej wersji czasu UNIX, pozwalającej na poprawne działanie aż do 15:30:08 UTC 4 grudnia 292277026596 to standard 32-bitowy jest nadal stosowany, prowadząc do "problemu roku 2038"..

Jako że w przeciwieństwie do GMT (Greenwich Mean Time), UTC jest oparty na zegarze atomowym, a prędkość obrotowa ziemi ulega zmianom, to w celu zapewnienia dokładności przejścia Słońca nad południkiem zerowym o godz. 12:00 UTC nie gorszej niż 0.9 sekundy, co pewien czas, do UTC dodawana jest *sekunda przestępna* (zakłada się również możliwość odjęcia sekundy przestępnej, jednak dotychczas taka konieczność nie zdarzyła się). Operację tę przeprowadza IERS (International Earth Rotation Service). Według standardu UNIX Epoch Time, różnica czasu między kolejnymi dobami wynosi zawsze 86400 sekund. Jeżeli wystąpi sekunda przestępna, to jedna wartość czasu UNIX może reprezentować dwie różne sekundy w czasie rzeczywistym (gdy sekunda przestępna jest dodawana i doba trwa dłużej) bądź może reprezentować sekundę która nigdy nie

miała miejsca (gdy sekunda przestępna jest odejmowana i doba trwa krócej). Jako że występowanie sekund przestępnych jest nieregularne i zależne od decycji komisji, zegary nie synchronizujące czasu z zewnętrznymi wzorcami ignorują sekundy przestępne. Będzie to powodowało przesunięcia pomiędzy zegarami które korzystają z synchronizacji sieciowej i tymi które nie mają takiej możliwości. Wprawdzie skala tych przesunięć wynosi maksymalnie 2s w ciągu roku, ale należy o tym pamiętać jeśli wykorzystujemy czas UNIX do dokładnego określania czasu zdarzenia. Od roku 1970 do końca 2015 dodano łącznie 26 sekund przestępnych. Polska znajduje się w strefach czasowych:

- UTC + 01:00 – czas środkowoeuropejski (CET) – w okresie zimowym, oraz
- UTC + 02:00 – czas środkowoeuropejski letni (CEST) – w okresie letnim

11.11.2. Konfiguracja zegara RTC

Pin V_{BAT} procesora Cortex-M pozwala na dołączenie baterii, która po wyłączeniu napięcia zasilającego V_{DD} zasila układ zegara RTC oraz pewien zbiór rejestrów oznaczonych jako *Backup Registers* (BKP). Rejestry te nie są kasowane po załączeniu zasilania i zachowują swoje wartości sprzed wyłączenia V_{DD}.

Układ RTC zawiera zestaw stale zliczających rejestrów, które mogą być wykorzystane jako zegar i kalendarz. Rejestry można zapisywać w celu ustawienia czasu i daty. Rejestr RCC_BDCR (*Backup domain control register*) należy do BKP i konfiguruje źródło sygnału taktującego RTC oraz zezwala na jego pracę. Po sygnale RESET dostęp do rejestrów BKP i rejestrów układu RTC jest zabroniony. W celu umożliwienia dostępu należy

- załączyć taktowanie układów *Power control* (PWR) i *Backup Registers* (BKP) przez ustawienie bitów `PWREN` i `BKPEN` w rejestrze RCC_APB1ENR,
- ustawić bit `DBP` w rejestrze *Power Control Register* (PWR_CR), co zezwoli na dostęp do rejestrów BKP i RTC.

Zapis do rejestrów zegara RTC jest możliwy tylko w trybie *Configuration Mode*. W tym celu należy ustawić bit `CNF` w rejestrze RTC_CRL. Ponadto zapis do któregokolwiek rejestru RTC jest możliwy tylko gdy zakończona została poprzednia operacja. Do wykrycia tego stanu służy bit `RTOFF` w rejestrze RTC_CR. Nowy zapis jest możliwy, gdy bit `RTOFF` jest ustawiony.

```
// ------------------- konfigurowanie RTC -------------------
void RTC_Config(void) {
   RCC -> APB1ENR |= RCC_APB1Periph_PWR;  // Power interface clock enable
   RCC -> APB1ENR |= RCC_APB1Periph_BKP;  // Backup interface clock enable
   PWR -> CR |= PWR_CR_DBP;       // Disable backup domain write protection
   // Select LSE clock
   RCC -> CSR &=  RCC_CSR_LSION;         // LSI (w RCC) off
   RCC -> BDCR &=  RCC_BDCR_LSEBYP;      // LSE oscillator Bypass = off
   RCC -> BDCR |= RCC_BDCR_RTCSEL_LSE;  // LSE oscillator clock used
   RCC -> BDCR |= RCC_BDCR_LSEON;  // External Low Speed oscillator enable
   while((RCC -> BDCR & RCC_BDCR_LSERDY) == 0)
       ;                           // czekamy na gotowość LSE
   RCC -> BDCR |= RCC_BDCR_RTCEN; // RTC clock enable
   RTC_WaitForSynchro();     // Oczekiwanie na synchronizację
   RTC_WaitForLastTask();    // Oczekiwanie aż rejestry RTC zostaną zapisane
```

```
   RTC_SetPrescaler(32767); // RTCCLK/RTC_PR = (32.768 KHz)/(32767+1)
   RTC_WaitForLastTask();
   RTC_ITConfig(RTC_IT_SEC, ENABLE);  // Włącz przerwanie "sekundnika" RTC

   RTC_WaitForLastTask();
   RCC_ClearFlag();
}
```

11.12. Projekt07_RTC

W głównym katalogu *ARM1CortexM3-ZL27ARM* kopiujemy ostatni projekt i zmieniamy nazwę na *Projekt07_RTC*. Uruchomimy zegar RTC, a w obsłudze przerwania sekundowego od RTC będziemy wyświetlać aktualny czas.

```
/*******************************************************************
* Function Name : RTC_IRQHandler
* Description   : This function handles RTC global interrupt request.
*******************************************************************/
extern unsigned int RTC_run;    // zezwolenie na wyświetlanie

void RTC_IRQHandler(void) {
   volatile unsigned int stanRTC;
   static unsigned char stanRTCTekst[17] = {0};
   int godziny, minuty, sekundy;
   if (RTC_GetITStatus(RTC_IT_SEC) != RESET) {
       RTC_WaitForLastTask();    // czekaj na możliwość zapisu do RTC
       RTC_ClearITPendingBit(RTC_IT_SEC); // zgaś bit pending
       stanRTC = RTC_GetCounter();
       stanRTC %= 24 * 3600;          // ograniczenie do 24 godzin
       sekundy = stanRTC % 60;        // reszta z dzielenia: sekundy
       minuty = stanRTC / 60;         // dzielenie całkowite: minuty
       godziny = minuty / 60;         // dzielenie całkowite: godziny
       minuty = minuty % 60;          // reszta z dzielenia: minuty
       sprintf((char *)stanRTCTekst,"%2i:%02i:%02i",godziny,minuty,sekundy);
       if(RTC_run)
           LCD_WriteTextXY(stanRTCTekst, 8, 0);
   }
}
```

W pliku *main.c* musimy zadeklarować zmienną zezwalającą na wyświetlanie czasu oraz włączyć przerwania od RTC.

```
unsigned int RTC_run = 1;    // zezwala na wyświetlanie czasu
// Włącz przerwanie od RTC ---------------------------------------------
NVIC_InitStructure.NVIC_IRQChannel = RTC_IRQn;
NVIC_InitStructure.NVIC_IRQChannelPreemptionPriority = 1;
NVIC_InitStructure.NVIC_IRQChannelSubPriority = 0;
NVIC_InitStructure.NVIC_IRQChannelCmd = ENABLE;
NVIC_Init(&NVIC_InitStructure);
```

Można także dopisać wczytywanie czasu GG:MM:SS z klawiatury i ustawianie zegara.

Rozdział 12

Liczniki i sterowanie PWM

W tym rozdziale zajmiemy się licznikami *Timers*. Mikrokontroler STM32F103 zawiera cztery liczniki. Najbardziej rozbudowany jest *Advanced-control timer* `TIM1`. Pozostałe trzy to *General-purpose timers*: `TIM2, TIM3` oraz `TIM4`. Wykorzystując te liczniki będziemy sterować diodami LED, wykonamy odczyt położenia drążka w nadajniku RC (*Radio Control*), będziemy sterować silnikami prądu stałego, zbudujemy zdalnie sterowany samochód terenowy z napędem na sześć kół, a na koniec dołączymy do niego ultradźwiękowy pomiar odległości od przeszkody. Wszystkie te zadania można sprowadzić do dwóch czynności: generowanie sygnałów cyfrowych o określonym okresie i wypełnieniu (sygnał PWM *Pulse-Width Modulation*) oraz pomiar czasu trwania stanu wysokiego (niskiego) sygnału cyfrowego.

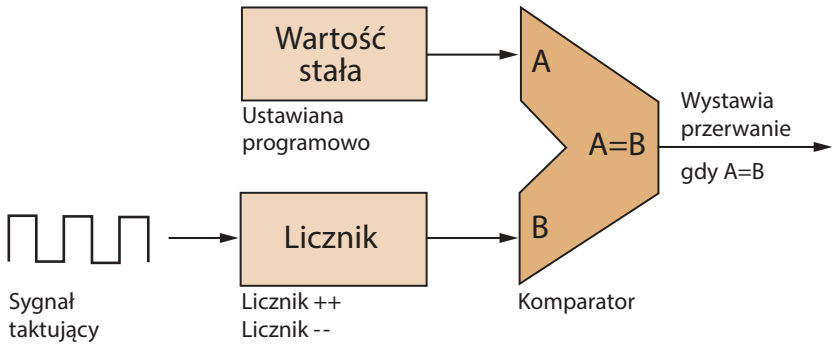
12.1. Budowa licznika i tryby zliczania

Licznik jest układem, który inkrementuje bądź dekrementuje swoją wartość z każdym taktem sygnału wejściowego. Częstotliwość sygnału taktującego można zmieniać programowo. Gdy licznik pracuje w trybie odliczania impulsów wejściowych (jak pokazano na rysunku 12.1), to może zgłosić przerwanie po osiągnięciu zadanej ich liczby. W trybie pomiaru czasu trwania stanu wysokiego sygnału, stan licznika jest zapamiętywany w momencie wystąpienia określonego zbocza sygnału wejściowego i wystawiane jest przerwanie. Różnica dwóch kolejnych stanów licznika pomnożona przez okres sygnału taktującego to czas pomiędzy kolejnymi zboczami. Jeżeli wykrywane są oba zbocza (narastające i opadające), uzyskujemy czas trwania stanu wysokiego (niskiego), gdy tylko jedno zbocze uzyskujemy okres sygnału wejściowego.

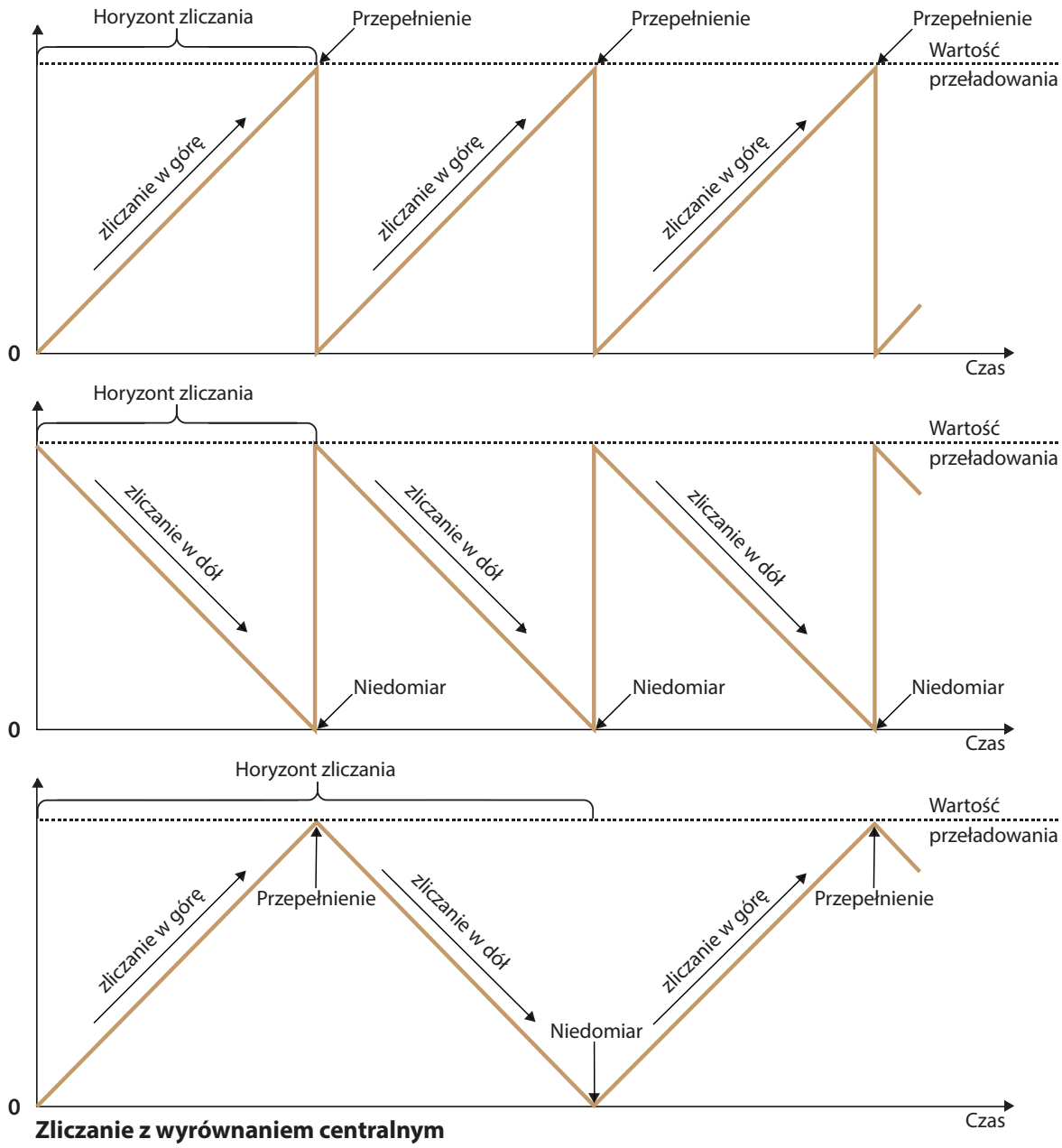
Licznik ma trzy tryby zliczania: w górę (*up-counting*), w dół (*down-counting*) oraz z wyrównaniem centralnym (*center-aligned counting*), jak pokazano na rysunku 12.2.

- w trybie *up-counting* licznik startuje od zera, zlicza do zapamiętanej wartości, wtedy jest zerowany i cykl się powtarza. Ta wartość jest pamiętana w rejestrze *auto-reload register* `ARR`. Na przykład, gdy `ARR = 4`, to wartości licznika zmieniają się następująco: `0, 1, 2, 3, 4, 0, 1, 2, 3, 4, 0, ...` i tak dalej.
- w trybie *down-counting* licznik startuje od zapamiętanej wartości i jest dekrementowany aż do osiągnięcia zera. Wtedy jest przeładowywany wartością pamiętaną w `ARR` i cykl się powtarza. Na przykład, gdy `ARR = 4`, to wartości licznika zmieniają się następująco: `4, 3, 2, 1, 0, 4, 3, 2, 1, 0, 4, ...`
- w trybie *center-aligned counting* wykonywane jest zliczanie w górę i w dół na przemian. Na przykład, gdy `ARR = 4`, to wartości licznika zmieniają się następująco: `0, 1, 2, 3, 4, 3, 2, 1, 0, 1, 2, ...`

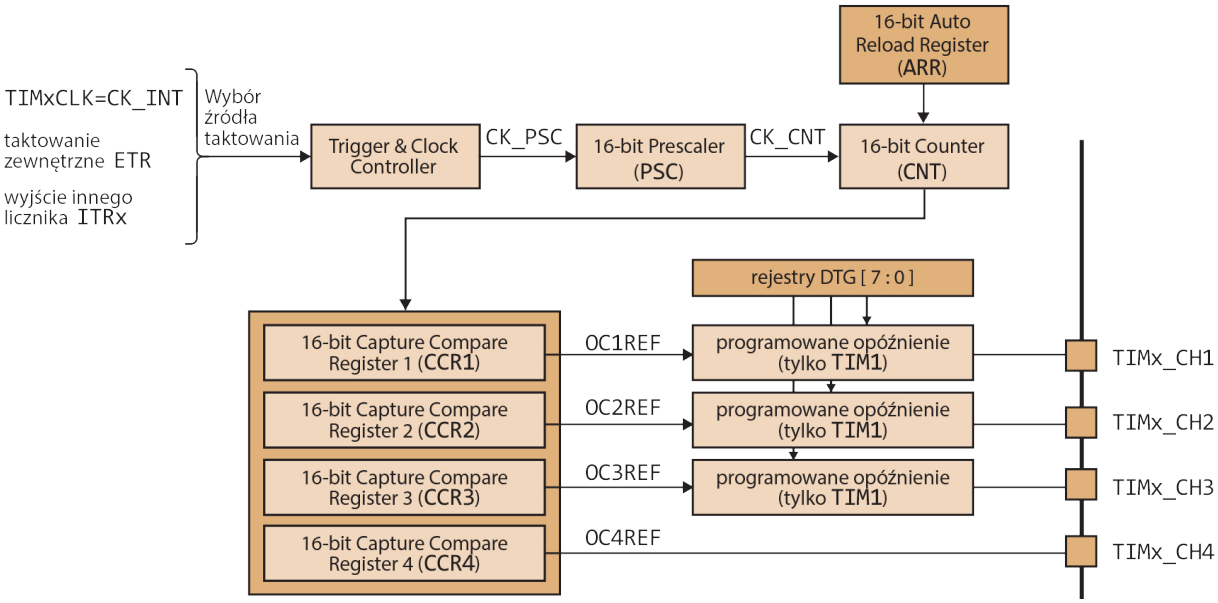
Zawartość licznika tworzy przebieg piłokształtny, rysunek 12.2. Odmierzony odcinek czasu zależy od częstotliwości taktowania oraz od zawartości rejestru `ARR`. Rozdzielczość licznika,



Rys. 12.1: Konfiguracja licznika stosowanego do odmierzania czasu



Rys. 12.2: Trzy tryby zliczania



Rys. 12.3: Konfiguracja licznika w trybie odmierzania czasu

to okres sygnału taktującego, horyzont czasowy obejmowany przez licznik to pojemność licznika pomnożona przez rozdzielczość. Licznik 16-bitowy taktowany sygnałem 1MHz, może odmierzyć czasy od 1μs do 65.536ms z krokiem (rozdzielczością) 1μs.

W trakcie odliczania mogą wystąpić dwa zdarzenia: przepełnienie (*overflow*) i niedomiar (*underflow*).

- w trybie zliczania w górę, gdy licznik doliczy do wartości pamiętanej w rejestrze `ARR`, w następnym takcie następuje przepełnienie i licznik jest zerowany
- w trybie zliczania w dół, gdy licznik doliczy do zera, w następnym takcie następuje niedomiar i licznik jest ładowany zawartością rejestru `ARR`.
- w trybie z wyrównaniem centralnym przepełnienie występuje na przemian z niedomiarem.

Jeżeli licznik jest wykorzystywany do pomiaru czasu pomiędzy zboczami sygnału wejściowego, to program obsługi przerwania musi rozpoznać czy pomiędzy tymi zboczami wystąpiło przepełnienie, aby właściwie zinterpretować stany licznika zapamiętane w czasie każdego z tych zboczy. Można to wykonać przez odczyt bitu `UIF` *Update Interrupt Flag* w rejestrze statusu `TIMx_SR`.

12.2. Odmierzanie odcinków czasu

Rysunek 12.3 przedstawia schemat konfiguracji licznika pracującego w trybie odmierzania odcinków czasu. Licznik *Counter* `CNT` ma długość 16 bitów. Jest on przeładowywany zawartością rejestru *Auto Reload Register* `ARR`, w zależności od trybu zliczania, jak było opisane wcześniej. Licznik może być taktowany z różnych źródeł:

- `CK_INT` *Internal Clock* jest to sygnał `TIMxCLK` pochodzący z układu RCC. Licznik `TIM1` otrzymuje swój sygnał `TIM1CLK` z linii APB2 (72MHz), natomiast liczniki `TIM2, TIM3, TIM4` mają wspólny sygnał taktujący `TIMxCLK` z linii APB1 (36MHz). Częstotliwość tą można podwoić, wtedy wszystkie liczniki będą taktowane częstotliwością 72MHz.

- ETR sygnał zewnętrzny z pinu procesora TIMxETR.
- ITRx wyjście innego licznika, umożliwia synchronizację liczników oraz ich kaskadowe łączenie.

Częstotliwość taktowania może zostać podzielona w 16-bitowym preskalerze PSC, zgodnie z regułą

$$f_{CK_CNT} = \frac{f_{CK_PSC}}{Prescaler - 1}$$

W każdym takcie, rejestr CNT jest porównywany jednocześnie z czterema rejestrami CCRx *Capture/Compare Register*. Wyjścia tych rejestrów są oznaczone jako OCxREF *Output Reference Signal*, czyli wyjście rejestru CCR1 to OC1REF, rejestru CCR2 to OC2REF i tak dalej. W licznikach *general* (TIM2/3/4) sygnały OCxREF są dołączone do pinów TIMx_CH1, TIMx_CH2, TIMx_CH3, TIMx_CH4 procesora. W liczniku *advanced* TIM1, pomiędzy wyjścia OCxREF a piny procesora włączone są układy, które umożliwiają dodanie opóźnień. Funkcja każdego wyjścia OCxREF może zostać określona przez ustalenie wartości bitów pola OCxM w rejestrach TIMx_CCMR zgodnie z tabelą

Output Compare Mode (OCM)	**wyjście OCxREF rejestru CCRx**
000	zamrożone (nie jest zmieniane)
001	wysokie gdy CNT = CCR
010	niskie gdy CNT = CCR
011	zmiana stanu (*toggle*) gdy CNT = CCR
100	wymuszenie stanu niskiego (zawsze niskie)
101	wymuszenie stanu wysokiego (zawsze wysokie)
110	*Pulse Width Modulation*, PWM tryb 1
111	*Pulse Width Modulation*, PWM tryb 2

W PWM tryb 1, gdy ustawione jest zliczanie do góry, wyjście rejestru CCRx jest aktywne, gdy CNT < CCR i nie–aktywne, gdy CNT > CCR. Przy zliczaniu w dół (dekrementacji) jest odwrotnie. PWM tryb 2, wyjście jest zanegowane w stosunku do trybu 1. Bity CCxP i CCxE w rejestrze TIMx_CCER decydują o polaryzacji wyjścia: CCxP = 1 aktywne wysokie, CCxP = 0 aktywne niskie. CCxE = 0 wyjście wyłączone, CCxE = 1 wyjście aktywne. W procesorze STM32F103 'x' przyjmuje wartości 1, 2, 3 oraz 4.

12.3. Projekt08_toggleLED

Wykorzystamy licznik TIM4 do zapalania/gaszenia diod LED1 i LED2. Nie będziemy używać do tego celu przerwań, wystarczy odpowiednio skonfigurować licznik TIM4. W płycie ZL27ARM (Tabela 12.1):

- pin 95 procesora (PB8), funkcja alternatywna TIM4_CH3 – dołączony jest do LED1,
- pin 96 procesora (PB9), funkcja alternatywna TIM4_CH4 – dołączony jest do LED2.

Jak zawsze, kopiujemy ostatni projekt i zmieniamy jego nazwę. W pliku *main.c*, na początku umieszczamy prototyp funkcji inicjalizacji

```
void TIM_Config(void);
```

Tabela 12.1: Sygnały dołączone do liczników w płycie ZL27ARM

pin	po RESET	funkcja alternatywna	sygnał w płycie ZL27ARM
23	PA0-WKUP	TIM2_CH1	przycisk SW0
24	PA1	TIM2_CH2	przycisk SW1
25	PA2	TIM2_CH3	przycisk SW2
26	PA3	TIM2_CH4	przycisk SW3
31	PA6	TIM3_CH1	MISO złącze karty SD1
32	PA7	TIM3_CH2	MOSI złącze karty SD1
35	PB0	TIM3_CH3	
36	PB1	TIM3_CH4	
51	PB12	TIM1_BKIN	LED5
52	PB13	TIM1_CH1N	LED6
53	PB14	TIM1_CH2N	LED7
54	PB15	TIM1_CH3N	LED8
67	PA8	TIM1_CH1	
68	PA9	TIM1_CH2	T1IN/ST3232 złącze Con3/RS232
69	PA10	TIM1_CH3	R1OUT/ST3232 złącze Con3/RS232
70	PA11	TIM1_CH4	DM Con2/USB
71	PA12	TIM1_ETR	DP Con2/USB
83	PD2	TIM3_ETR	przełącznik JP9, pin 1
92	PB6	TIM4_CH1	
93	PB7	TIM4_CH2	
95	PB8	TIM4_CH3	LED1, RxD/L9616 złącze Con4/CAN
96	PB9	TIM4_CH4	LED2, TxD/L9616 złącze Con4/CAN
97	PE0	TIM4_ETR	

Do procedury inicjalizacji dodajemy wywołanie funkcji

```
TIM_Config();
```

a samą funkcję umieścimy na końcu pliku. Musimy zmienić inicjalizację portu GPIOB, tak aby załączyć funkcje alternatywne w liniach PB8 i PB9.

```
// port B - diody LED
GPIO_InitStructure.GPIO_Pin = GPIO_Pin_10 | GPIO_Pin_11 | GPIO_Pin_12 |
                                    GPIO_Pin_13 | GPIO_Pin_14 | GPIO_Pin_15;
GPIO_InitStructure.GPIO_Speed = GPIO_Speed_2MHz;
GPIO_InitStructure.GPIO_Mode = GPIO_Mode_Out_PP;     // wyjście push-pull
GPIO_Init(GPIOB, &GPIO_InitStructure);
// port B - wyjścia TIM4
GPIO_InitStructure.GPIO_Pin = GPIO_Pin_8 | GPIO_Pin_9;
GPIO_InitStructure.GPIO_Speed = GPIO_Speed_2MHz;
GPIO_InitStructure.GPIO_Mode = GPIO_Mode_AF_PP;      // Alternate function
GPIO_Init(GPIOB, &GPIO_InitStructure);
```

Do funkcji RCC_Config dodamy włączenie taktowania licznika TIM4

```
RCC_APB1PeriphClockCmd(RCC_APB1Periph_TIM4, ENABLE);//włącz taktowanie TIM4
```

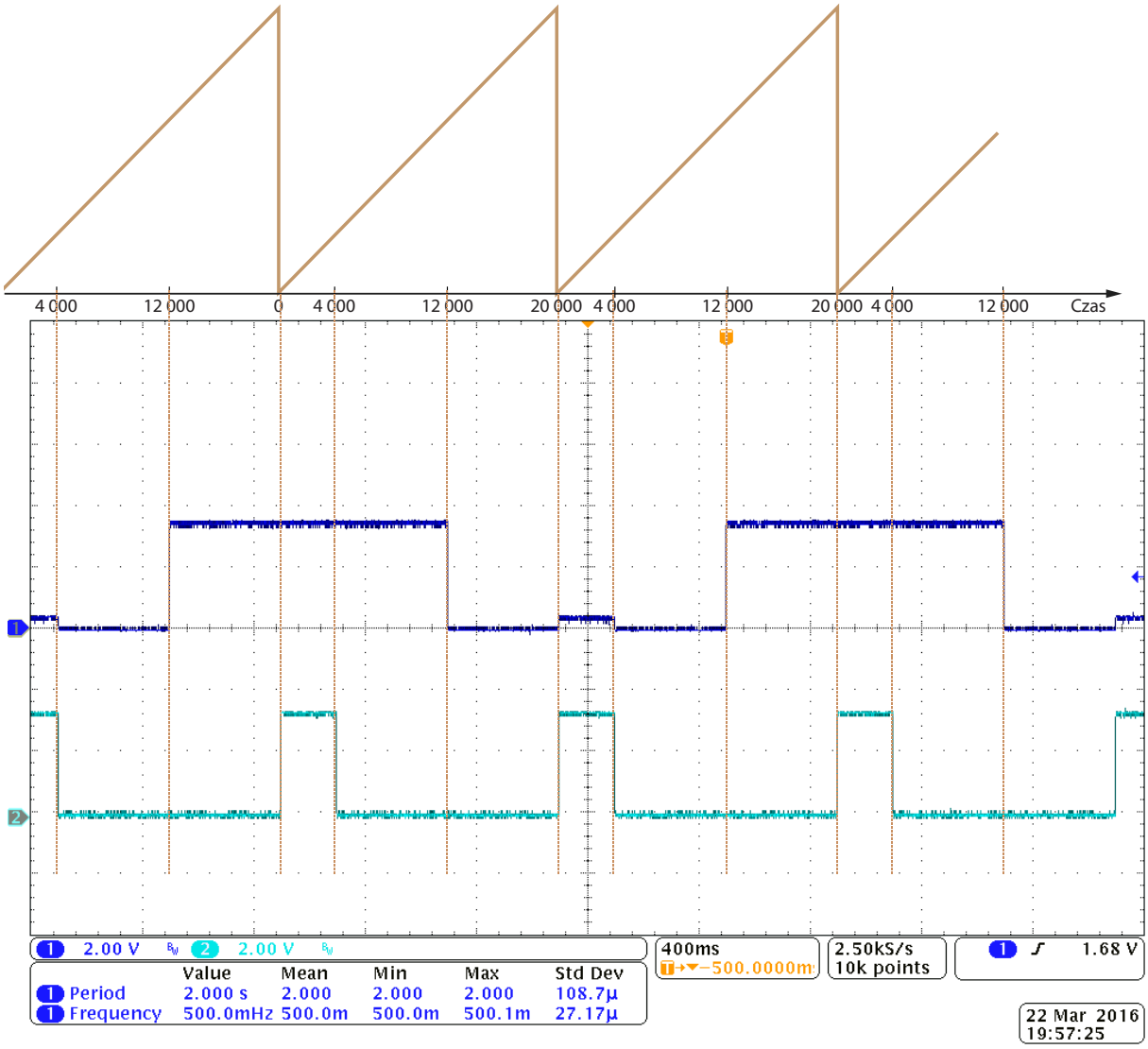
Na końcu pliku umieszczamy funkcję inicjalizacji licznika TIM4.

```
// -------------- konfiguracja licznika TIM4 --------------
void TIM_Config(void) {
   TIM_TimeBaseInitTypeDef TIM_TimeBaseStructure;
   TIM_OCInitTypeDef TIM_OCInitStructure;
   //Konfiguracja licznika 4: ustawienia taktowania i trybu pracy
   TIM_TimeBaseStructure.TIM_Prescaler = 3600-1; // fclk =72MHz/3600 =20kHz
   TIM_TimeBaseStructure.TIM_Period = 20000-1;   //20000 taktów = 1 sekunda
   TIM_TimeBaseStructure.TIM_ClockDivision = TIM_CKD_DIV1; // dead-time
   TIM_TimeBaseStructure.TIM_RepetitionCounter=0;    // licznik powtórzeń
   TIM_TimeBaseStructure.TIM_CounterMode = TIM_CounterMode_Up; //tryb pracy
   TIM_TimeBaseInit(TIM4, &TIM_TimeBaseStructure);    // inicjalizacja
   // Konfiguracja kanału 3
   TIM_OCInitStructure.TIM_OCMode = TIM_OCMode_Toggle; //tryb pracy kanału
   TIM_OCInitStructure.TIM_OutputState = TIM_OutputState_Enable; //włączenie
   TIM_OCInitStructure.TIM_Pulse =12000;  // przesunięcie o 12000 taktów
   TIM_OCInitStructure.TIM_OCPolarity = TIM_OCPolarity_High; //polaryzacja
   TIM_OC3Init(TIM4, &TIM_OCInitStructure); //Inicjalizacja kanalu 3
   TIM_OC3PreloadConfig(TIM4, TIM_OCPreload_Enable); // buforowanie
   //Konfiguracja kanału 4
   TIM_OCInitStructure.TIM_OCMode = TIM_OCMode_PWM1; // tryb pracy kanału
   TIM_OCInitStructure.TIM_OutputState = TIM_OutputState_Enable; //włączenie
   TIM_OCInitStructure.TIM_Pulse =4000; // wypełnienie 20%
   TIM_OCInitStructure.TIM_OCPolarity = TIM_OCPolarity_High; //polaryzacja
   TIM_OC4Init(TIM4, &TIM_OCInitStructure); // inicjalizacja kanału 4
   TIM_OC4PreloadConfig(TIM4, TIM_OCPreload_Enable); // buforowanie
   TIM_Cmd(TIM4, ENABLE);    // włączenie timera TIM4
}
```

Efekt pracy programu widoczny jest na rysunku 12.4. Częstotliwość taktowania licznika jest równa 72 MHz, do preskalera `PSC` wpisujemy wartość 3600-1, czyli częstotliwość taktowania licznika `CK_CNT` jest równa 72 MHz / 3600 = 20 kHz. Do rejestru `ARR` wpisujemy wartość 20'000, czyli horyzont zliczania jest równy jednej sekundzie.

Kanał 4 (dolny wykres na rysunku 12.4) jest zaprogramowany jako PWM1 (tryb 1). Do rejestru `CCR4` wpisaliśmy 4000. Licznik `CNT` zlicza o zera w górę. Na początku `CNT < CCR4`, czyli wyjście `OC4REF` jest jedynką. Po przekroczeniu wartości 4000 przez licznik `CNT`, wyjście to jest zerowane i pozostaje zerem aż do końca horyzontu zliczania. Wypełnienie sygnału jest więc równe 4'000 / 20'000 = 20%.

Kanał 3 (górny wykres na rysunku 12.4) jest zaprogramowany jako `Toggle`. To oznacza, że wyjście `OC3REF` zmienia swój stan na przeciwny zawsze, gdy `CNT = CCR3`. Zaprogramowaliśmy `CCR3` = 12'000, czyli zawsze gdy `CNT` = 12'000, zmieniany jest stan wyjścia `OC3REF`, jak widać na rysunku 12.4. Okres sygnału jest równy podwojonemu horyzontowi zliczania, czyli dwie sekundy, a częstotliwość 0.5 Hz. Takie wartości wyświetlił oscyloskop (dolne okno, rysunek 12.4).



Rys. 12.4: Sygnały na wyjściach `TIM4_CH3` i `TIM4_CH4`

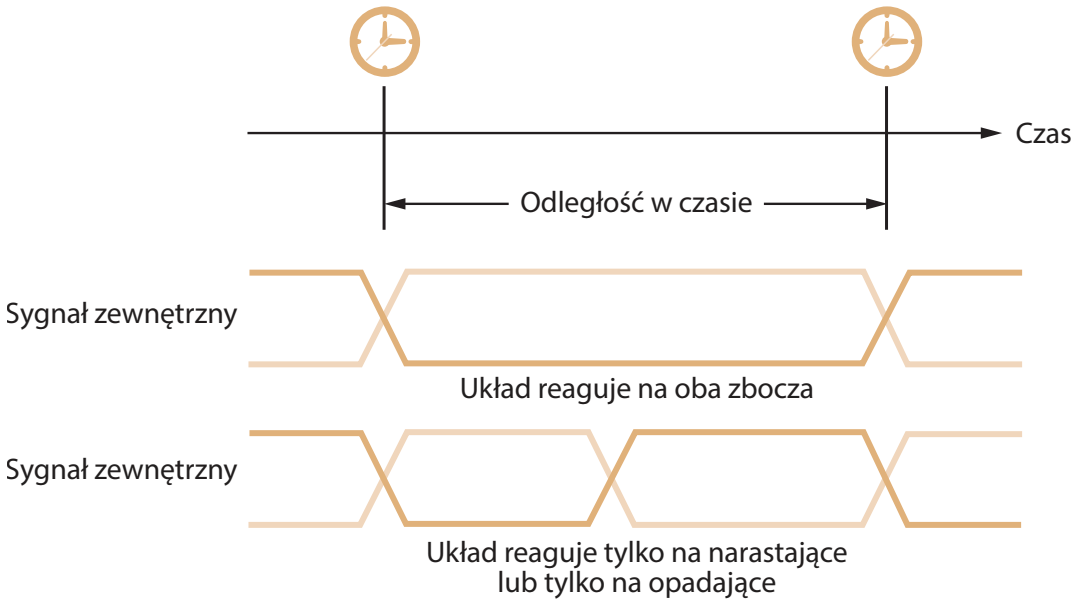
12.4. Pomiar długości impulsu

Naszym celem jest pomiar czasu pomiędzy zboczami sygnału zewnętrznego. Możemy mierzyć czas pomiędzy zboczami (rysunek 12.5):

- narastające−opadające (stan wysoki), opadające−narastające (stan niski),
- tylko narastające (okres sygnału),
- tylko opadające (okres sygnału).

Gdy układ *Edge Detector* wykryje zbocze, na które został zaprogramowany, aktualna zawartość licznika `CNT` zostaje przepisana do rejestru *Capture & Compare Register* `CCR` i zgłaszane jest przerwanie. Jeżeli mierzone są długie odcinki czasu, program musi wykrywać i zliczać momenty przepełnienia licznika. Każdy kanał ma pin wejściowy, filtr, układ wykrywania zbocza i preskaler

- układ wykrywania zbocza (*Edge Detector*) można zaprogramować na wykrywanie tylko zboczy narastających, tylko opadających lub obu,



Rys. 12.5: Pomiar czasu pomiędzy zboczami sygnału.

- preskaler: zliczane będzie każde zbocze, co drugie, co czwarte, co ósme
- filtr określa ilość kolejnych jednakowych stanów sygnału, aby zmiana stanu (wysoki/niski) została przyjęta (N = 2, 4, 8) oraz określa podział częstotliwości f_{DTS} przez 1, 2, 4, 8, 16, 32. Wpisanie zera wyłącza tą funkcję.

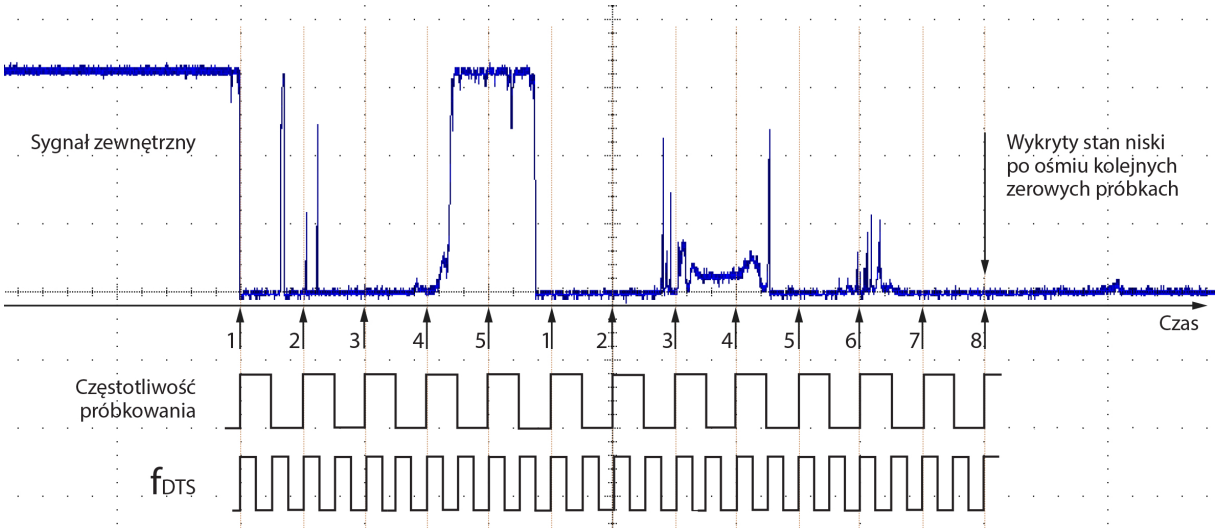
Każdy kanał jest wyposażony w prosty filtr cyfrowy którego zadaniem jest pominięcie stanu nieustalonego sygnału wejściowego. Na przykład, jeżeli odbicia styków trwają pięć okresów zegara, możemy zaprogramować filtr na okres testowania sygnału znacznie dłuższy niż te pięć taktów Jeżeli określona ilość odczytów sygnału (kolejnych próbek) wykaże ten sam stan, to dopiero wtedy układ uzna sygnał za stabilny.

Sygnał *dead-time and sampling clock* f_{DTS}, taktujący filtr cyfrowy powstaje przez podział wewnętrznego zegara systemowego *Internal Clock* `CK_INT`. Współczynnik podziału zależy od stanu bitów `9:8` rejestru `TIM_CR1`. Jest to pole `CKD[1:0]`.

`CKD[1:0] = 00:` f_{DTS} `= CK_INT`
`CKD[1:0] = 01:` f_{DTS} `= CK_INT / 2`
`CKD[1:0] = 10:` f_{DTS} `= CK_INT / 4`
`CKD[1:0] = 11:` Reserved

Możemy także zmieniać okres próbkowania i wymaganą ilość jednakowych próbek, programując pole bitowe `ICxF` (‘`x`’ to numer kanału) w rejestrze `TIMx_CCMR1(2)`. W trybie *input capture mode*, znaczenie bitów jest następujące:

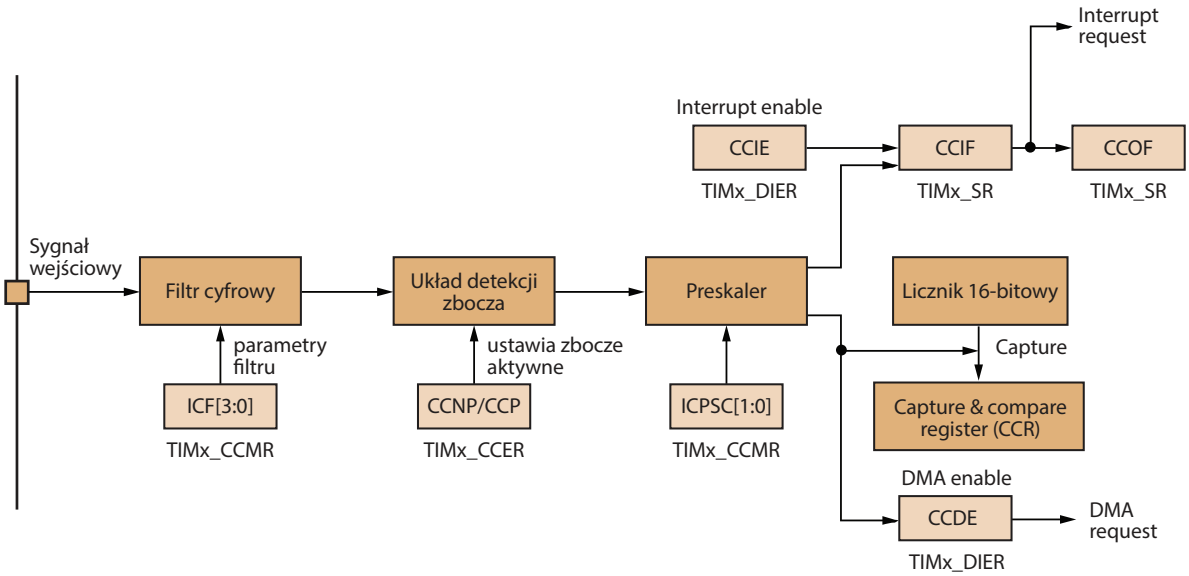
0000:	filtr wyłączony, próbkowanie z f_{DTS}	1000:	$f_{SAMPLING} = f_{DTS}/8$, N=6
0001:	$f_{SAMPLING} = f_{CK_INT}$, N=2	1001:	$f_{SAMPLING} = f_{DTS}/8$, N=8
0010:	$f_{SAMPLING} = f_{CK_INT}$, N=4	1010:	$f_{SAMPLING} = f_{DTS}/16$, N=5
0011:	$f_{SAMPLING} = f_{CK_INT}$, N=8	1011:	$f_{SAMPLING} = f_{DTS}/16$, N=6
0100:	$f_{SAMPLING} = f_{DTS}/2$, N=6	1100:	$f_{SAMPLING} = f_{DTS}/16$, N=8
0101:	$f_{SAMPLING} = f_{DTS}/2$, N=8	1101:	$f_{SAMPLING} = f_{DTS}/32$, N=5
0110:	$f_{SAMPLING} = f_{DTS}/4$, N=6	1110:	$f_{SAMPLING} = f_{DTS}/32$, N=6
0111:	$f_{SAMPLING} = f_{DTS}/4$, N=8	1111:	$f_{SAMPLING} = f_{DTS}/32$, N=8



Rys. 12.6: Przykład filtrowania sygnału wejściowego

Przykład filtrowania sygnału wejściowego pokazany jest na rysunku 12.6. Konfiguracja `ICxF:0101`, czyli podział sygnału f_{DTS} przez dwa i osiem kolejnych jednakowych próbek.

Każdy licznik ma cztery kanały. Konfiguracja kanału służącego do pomiaru długości impulsu pokazana jest na rysunku 12.7. Do pinu procesora (`GPIO_Mode_IN_FLOATING`), którego funkcją alternatywną jest wejście określonego kanału, dołączony jest sygnał, którego parametry chcemy zmierzyć. Za filtrem cyfrowym jest układ wykrywania zbocza, a następnie preskaler (do prostych pomiarów należy go wyłączyć). Wykrycie określonego zbocza powoduje zatrzaśnięcie aktualnego stanu licznika w rejestrze *Capture & Compare Register* `CCR` oraz zgłoszenie przerwania. Dwa kolejne projekty: zliczanie naciśnięć i pomiar sygnału z odbiornika aparatury do zdalnego sterowania modeli są ilustracją stosowania liczników w konfiguracji pomiaru sygnału wejściowego.



Rys. 12.7: Kanał licznika `TIMx` skonfigurowany do pomiaru długości impulsu

12.5. Projekt09_TIM2switch, eliminacja drgań styków

W tym projekcie będziemy wyświetlać ilość naciśnięć przycisku SW0 – jest to także joystick w kierunku N. Do eliminacji drgań styków wykorzystamy filtr cyfrowy w pierwszym kanale licznika TIM2. Każde naciśnięcie będzie sygnalizowane zmianą stanu diody LED5. Jak zwykle zaczniemy od wykonania kopii ostatniego projektu i zmiany jej nazwy. Filtr zaprogramujemy na możliwie najdłuższy okres próbkowania i największą ilość wymaganych jednakowych próbek sygnału. W naszej płycie ZL27ARM zegary ogólnego przeznaczenia TIM2/3/4 są taktowane sygnałem CK_INT = 72 MHz. Programując TIM_CR1 -> CKD = 0x2 (binarnie 10), podzielimy tą częstotliwość przez cztery, czyli filtr cyfrowy będzie taktowany sygnałem f_{DTS} o częstotliwości 18 MHz. Pole bitowe IC1F rejestru TIM2_CCMR1 zaprogramujemy na wartość 0xF (binarnie 1111). Oznacza to podział sygnału f_{DTS} przez 32 i wymóg ośmiu jednakowych kolejnych próbek. Uzyskamy okres próbkowania T_S = 32/18 μs. Osiem jednakowych próbek to horyzont czasowy 8*32/18 = 14 μs. W pliku *main.c* zadeklarujemy zmienną globalną:

```
unsigned int SW0count = 0;     // licznik
```

w funkcji *main()*, po inicjalizacji umieścimy wyświetlanie stanu licznika przyciśnięć

```
while (1) {
   LCD_GoTo(0,0);
   sprintf((char *)txt2LCD,"SW0=%4i",SW0count);
   LCD_WriteText(txt2LCD);
   delay_us(800000);
}
```

Do funkcji RCC_Config dodamy włączenie taktowania licznika TIM2

```
RCC_APB1PeriphClockCmd(RCC_APB1Periph_TIM2, ENABLE);//włącz taktowanie TIM2
```

Do funkcji GPIO_Config() dodamy inicjalizację portu GPIOA. Switche SW0-3 dołączone są do linii GPIOA0 - GPIOA3 (Tablica 12.1).

```
// port A - Przyciski i joystick
GPIO_InitStructure.GPIO_Pin = GPIO_Pin_0|GPIO_Pin_1|GPIO_Pin_2|GPIO_Pin_3;
GPIO_InitStructure.GPIO_Speed = GPIO_Speed_2MHz;
GPIO_InitStructure.GPIO_Mode = GPIO_Mode_IN_FLOATING;
GPIO_Init(GPIOA, &GPIO_InitStructure);
```

W funkcji konfigurującej kontroler przerwań NVIC_Config() musimy dopisać włączenie przerwania od TIM2

```
// Włącz przerwanie od TIM2 ---------------------------------------------
NVIC_InitStructure.NVIC_IRQChannel = TIM2_IRQn;     // TIM2 global Interrupt
NVIC_InitStructure.NVIC_IRQChannelPreemptionPriority = 1;
NVIC_InitStructure.NVIC_IRQChannelSubPriority = 0;
NVIC_InitStructure.NVIC_IRQChannelCmd = ENABLE;
NVIC_Init(&NVIC_InitStructure);
```

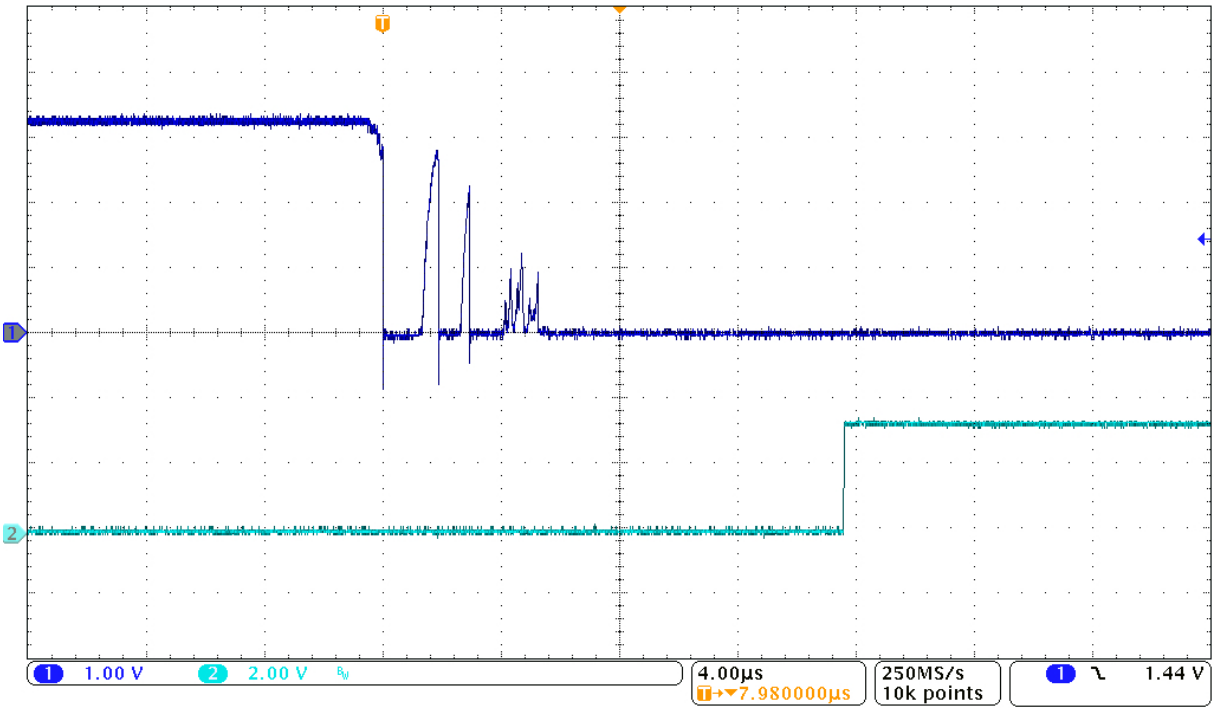
Do funkcji TIM_Config() dodamy inicjalizację licznika TIM2 oraz jego pierwszego kanału

```
//Konfiguracja licznika 2: ustawienia taktowania i trybu pracy licznika 2
TIM_InternalClockConfig(TIM2);                  // zegar systemowy 72 MHz
TIM_TimeBaseStructure.TIM_Prescaler = 7200-1;  // 72MHz / 7200 = 10 kHz
TIM_TimeBaseStructure.TIM_Period = 0xFFFF;     // ARR = MAX
TIM_TimeBaseStructure.TIM_ClockDivision = TIM_CKD_DIV4;  // fDTS = 18 MHz
TIM_TimeBaseStructure.TIM_RepetitionCounter=0; // wyłącz preskaler
TIM_TimeBaseStructure.TIM_CounterMode = TIM_CounterMode_Up;
TIM_TimeBaseInit(TIM2, &TIM_TimeBaseStructure);// inicjalizacja licznika
//Konfiguracja kanalu 1
TIM_ICInitStructure.TIM_Channel = TIM_Channel_1;// numer kanału
TIM_ICInitStructure.TIM_ICPolarity = TIM_ICPolarity_Falling;   // zbocze
TIM_ICInitStructure.TIM_ICSelection = TIM_ICSelection_DirectTI;// wejście
TIM_ICInitStructure.TIM_ICPrescaler = TIM_ICPSC_DIV1;
TIM_ICInitStructure.TIM_ICFilter = 0xF;    // f/32, N=8
TIM_ICInit(TIM2, &TIM_ICInitStructure);
TIM_Cmd(TIM2, ENABLE);                    // włączenie timera TIM2
TIM_ITConfig(TIM2, TIM_IT_CC1, ENABLE);    // włączenie przerwania od CC1
```

Pozostaje napisanie programu obsługi przerwania

```
/*********************************************************************
* Function Name : TIM2_IRQHandler
* Description   : This function handles TIM2 global interrupt request.
* Input         : None
* Output        : None
* Return        : None
*********************************************************************/
extern unsigned int SW0count;    // licznik przyciśnięć
void TIM2_IRQHandlervoid){
   if(TIM_GetITStatus(TIM2, TIM_IT_CC1) != RESET){
      TIM_ClearITPendingBit(TIM2, TIM_IT_CC1);
      SW0count++;
   }
   GPIO_WriteBit(GPIOB, GPIO_Pin_12, (BitAction)(1-GPIO_ReadOutputDataBit(
                                                  GPIOB, GPIO_Pin_12)));
}
```

Teraz możemy uruchomić całość. Każde naciśnięcie SW0 powoduje zwiększenie wyświetlanej wartości licznika. Zmiana stanu diody LED5 pozwoli nam na pomiar, za pomocą oscyloskopu, horyzontu obejmowanego przez filtr cyfrowy. Wynik pomiaru pokazany jest na rysunku 12.8. Widać odbicia styków SW0 (górny sygnał) oraz zmianę stanu diody LED5 (dolny sygnał). Odledłość pomiędzy pierwszym zboczem opadającym SW0, a zboczem narastającym LED5 jest horyzontem działania filtru cyfrowego. Podstawa czasu to 4 μs, odległość pomiędzy zboczami to około 14 μs, co odpowiada ośmiu okresom próbkowania filtru cyfrowego.

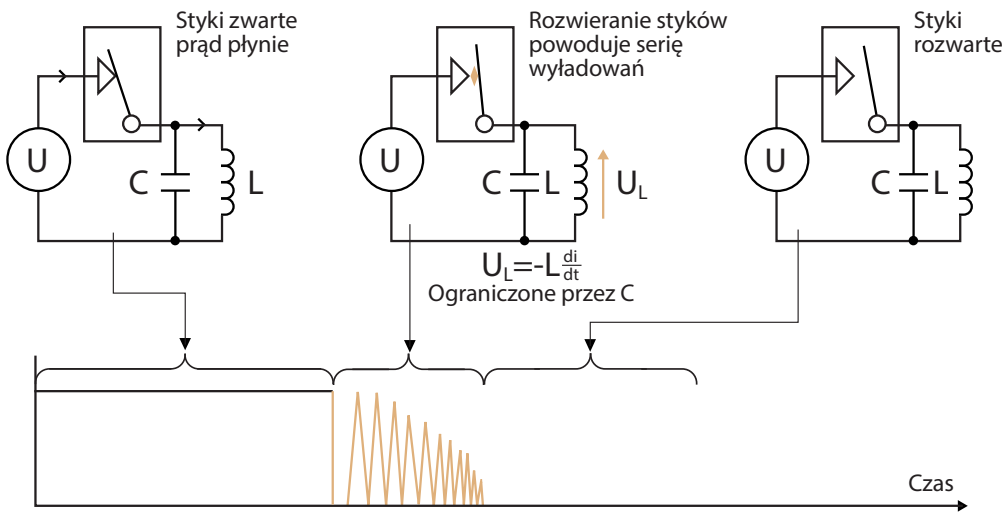


Rys. 12.8: Efekt działania filtru cyfrowego: górny sygnał to styk `SW0`, dolny to `LED5`

Czas trwania stanów nieustalonych zależy od jakości styków oraz indukcyjności i pojemności dołączonych do styków. Każdy układ posiada indukcyjność na którą składają się indukcyjności elementów dyskretnych (cewki, dławiki, transformatory), indukcyjności szczątkowe elementów "nie-indukcyjnych" oraz indukcyjność rozproszona zawarta w ścieżkach obwodów drukowanych, przewodach i wszystkich innych elementach przewodzących prąd. Podobnie jest z pojemnością, która jest sumą pojemności elementów dyskretnych (kondensatory), pojemności szczątkowych pozostałych elementów (na przykład dławików) i pojemności rozproszonej. Mechanizm powstawania zakłóceń wyłączeniowych na stykach pokazany jest na rysunku 12.9. Napięcie zaindukowane na indukcyjności w której nastąpiła zmiana prądu jest równe

$$U_L = -L \cdot \frac{di}{dt} \tag{12.1}$$

Kiedy obwód jest rozłączany, prąd płynący przez przełącznik jest gwałtownie przerywany. W chwili rozłączania styków mamy nieskończoną wartość di/dt. Wynikiem tego jest powstanie impulsu dużego napięcia U_L, które dodaje się do napięcia roboczego na rozwierających się zestykach przełącznika. Pojawia się mała, ale zwiększająca się szczelina powietrzna między stykami, która pod wpływem napięcia ulega przebiciu, co powoduje ponowny przepływ prądu zmniejszający wartość pochodnej di/dt, czyli zmniejszający napięcie U_L i wygaszenie powstałego łuku. Wygaszenie łuku oznacza ponowne przerwanie prądu, zatem znowu pojawi się impuls napięcia powodujący powstanie kolejnego łuku. Amplituda każdego kolejnego łuku jest mniejsza, gdyż następuje stopniowe rozładowywanie energii zgromadzonej w polu manetycznym indukcyjności. Stałe większanie odległości pomiędzy stykami powiązane z malejącą energią łuku powoduje, że po pewnym czasie proces zostaje przerwany. Dopiero w tym momencie można powiedzieć, że obwód został rozłączony. Widocznym efektem jest pojawienie się iskry między zestykami, co w rzeczywistości odpowiada całej serii małych iskier. To zjawisko prowadzi do powstania szybkich



Rys. 12.9: Zakłócenia wyłączeniowe *burst*

zakłóceń wyłączeniowych *burst*, czyli serii impulsów napięcia 5/50 ns (5 ns czas narastania i 50 ns czas opadania). Energia zakłóceń *burst* jest zmniejszana przez pojemność układu. Indukcyjność przeciwstawia się gwałtownym zmianom prądu (generując napięcie U_L). Pojemność przeciwstawia się gwałtownym zmianom napięcia, powodując przepływ prądu

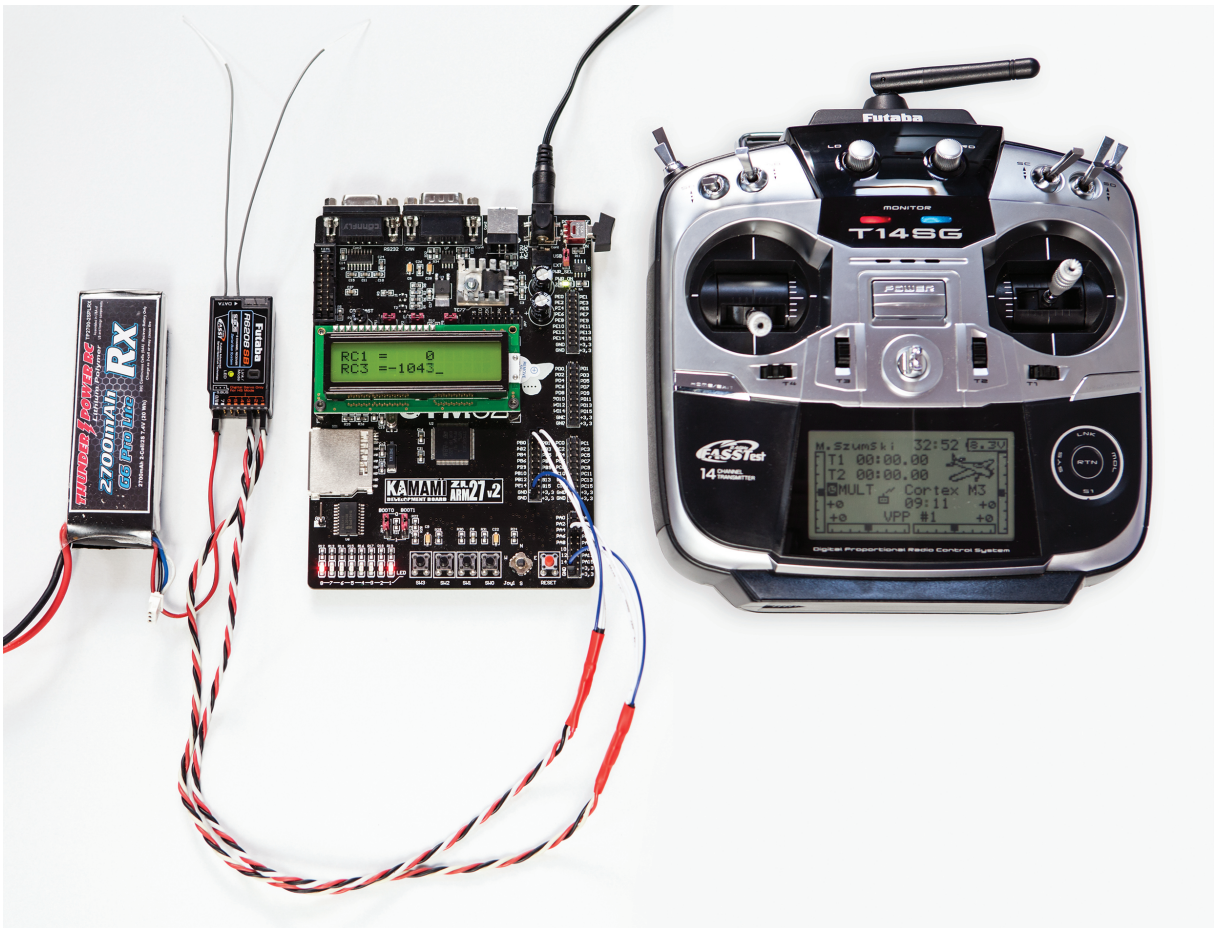
$$I_C = C \cdot \frac{du}{dt}$$

Można powiedzieć, że kondensator "gasi" szpilki napięcia. Zjawisko szybkich zakłóceń wyłączeniowych było wykorzystywane w samochodach z zapłonem iskrowym (silniki benzynowe), gdzie iskra była generowana w układzie: akumulator - cewka zapłonowa - przerywacz - świeca.

Układy elektroniczne należy tak projektować, aby zakłócenia generowane przez styki były możliwie najmniejsze. Opisane stany przejściowe nie tylko utrudniają odczyt położenia styków, ale są źródłem zakłóceń przewodzonych (obecnych w elementach przewodzących) oraz emitowanych w postaci szybkozmiennego pola elektromagnetycznego.

12.6. Projekt10_RadioControl

Nasz kolejny projekt będzie polegał na pomiarze sygnałów z odbiornika aparatury do zdalnego sterowania modeli. Do tego celu wykorzystamy licznik `TIM3`. Nadajnik do zdalnego sterowania modeli jest wyposażony w dwa drążki sterownicze, każdy o dwóch stopniach swobody, rysunek 12.10. Mamy łącznie cztery stopnie swobody, czyli cztery kanały. W odbiorniku kanały są rozdzielone na poszczególne złącza. Do każdego kanału możemy dołączyć jeden serwomechanizm. Dźwignia (orczyk) serwomechanizmu nadąża za ruchem odpowiedniego drążka w nadajniku. Serwomechanizm dołączony jest do odbiornika trzema przewodami: czarny to minus zasilania (GND), czerwony to plus zasilania, biały jest przewodem sygnałowym. Współczesne aparatury wykonywane są w technice cyfrowej 3.3V. Sygnał z odbiornika jest impulsem 3.3V o długości liniowo zależnej od chwilowego położenia drążka w nadajniku i możemy podłączyć go bezpośrednio do wejść mikrokontrolera. Przebiegi oscyloskopowe z czterech kanałów pokazane są na rysunku



Rys. 12.10: Nadajnik RC z odbiornikiem dołączonym do płyty ZL27ARM

12.11. Górny sygnał to jedno skrajne położenie, poniżej drążek jest w neutrum i poniżej drugie skrajne położenie. W oknie u dołu rysunku podane są czasy trwania każdego z tych impulsów

- 1.940 ms - pierwsze skrajne położenie drążka
- 1.520 ms - położenie środkowe (neutrum)
- 1.100 ms - przeciwne skrajne położenie drążka

Będziemy to traktować jako 1.520 ms +/− 420 μs. Cały zakres regulacji nadajnik dzieli na 2048 odcinków. Zachowamy tą rozdzielczość w pomiarze długości trwania impulsu. W przedziale czasu $840 \cdot 10^{-6}$ musimy zmieścić 2048 impulsów, czyli każdy impuls powinien trwać nie dłużej niż

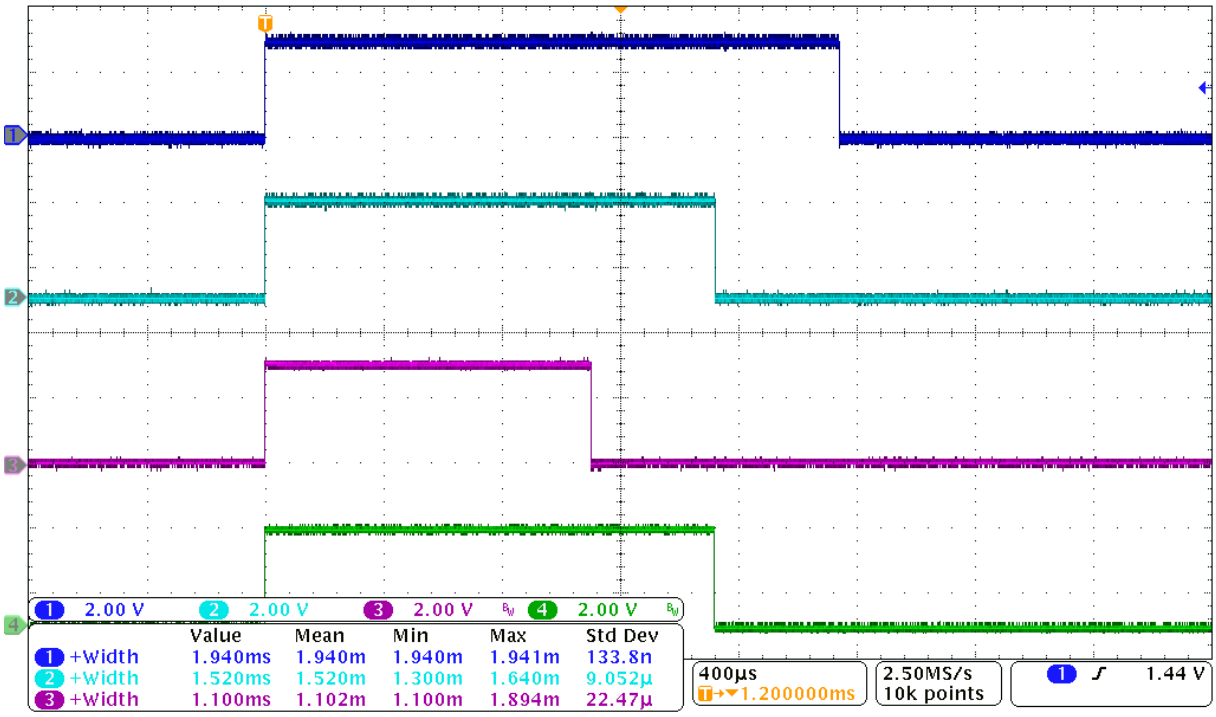
$$T_s \leqslant \frac{840 \cdot 10^{-6}}{2048} \qquad \longrightarrow \qquad f_s \geqslant \frac{2048}{840 \cdot 10^{-6}} = 2'430'095\ Hz$$

Podstawową częstotliwość taktowania liczników 72 MHz podzielimy przez 29

$$f_s = \frac{72'000'000}{29} = 2'482'759 > 2'430'095\ Hz$$

Przy podziale przez 29, w czasie $840\mu s$ licznik zliczy 2086 impulsów:

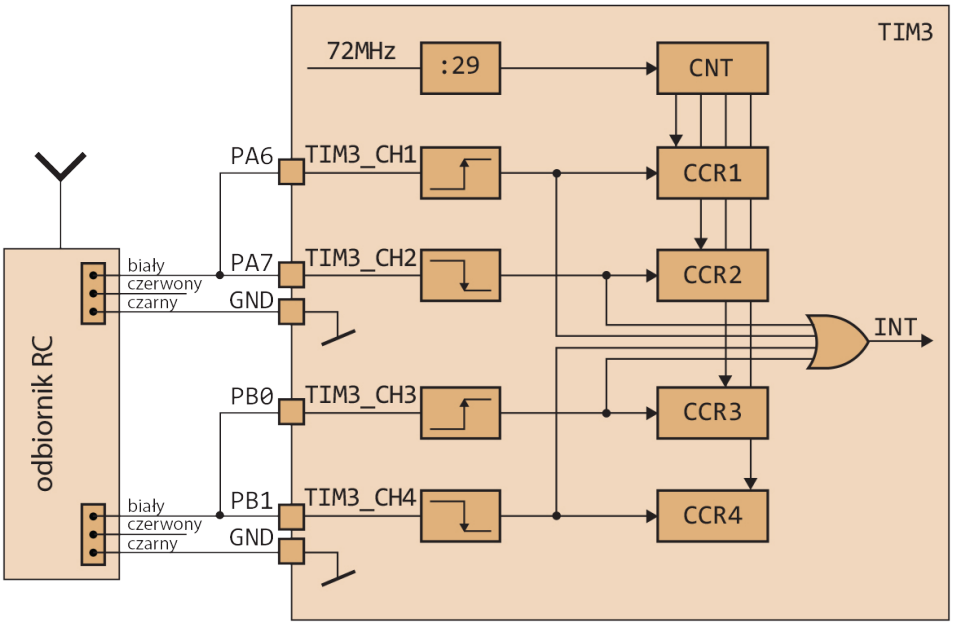
$$x = 2'482'759 \cdot 840\mu s = 2086$$



Rys. 12.11: Sygnały odczytane z odbiornika RC - pierwsze 4 kanały

Zamienimy to na zero w neutrum +/− 1043 takty zegara. Najdłuższy impuls z odbiornika, czyli 1.940 ms to 4'817 taktów. Wykorzystamy licznik `TIM3`. Licznik `TIM2` ma wejścia kanałów dołączone do przycisków `SW0 - SW3`, gdybyśmy tam dołączyli mierzone sygnały, to naciśnięcie któregoś z przycisków spowodowałoby zwarcie i uszkodzenie odbiornika RC. Wygodniej będzie wykorzystać wszystkie cztery kanały licznika `TIM3`. Do pomiaru długości impulsu wykorzystamy jeden kanał zaprogramowany na wykrycie zbocza narastającego. W obsłudze przerwania zapamiętamy zatrzaśniętą wartość licznika. Drugi kanał skonfigurujemy na wykrywanie zbocza opadającego. W obsłudze, od wartości zatrzaśniętej odejmiemy tą zapamiętaną w chwili narastania sygnału. W ten sposób wyliczymy ilość impulsów pomiędzy zboczami, co jest miarą czasu trwania impulsu, czyli położenia drążka w nadajniku. Zgodnie z tabelą 12.1 wejścia kanałów licznika `TIM3` są alternatywne w stosunku do linii portów GPIO: `PA6, PA7, PB0` i `PB1`. Konfiguracja licznika `TIM3` do pomiaru sygnałów z odbiornika RC pokazana jest na rysunku 12.13.

Rys. 12.12: Numeracja kanałów w nadajniku RC *Futaba* (mode 2)



Rys. 12.13: Konfiguracja licznika TIM3, pomiar sygnałów RC

Kopiujemy ostatni projekt i zmieniamy jego nazwę. W pliku main.c zaczniemy od zadeklarowania zmiennych globalnych

```
int RC1stick = 0;          // położenie drążka 1 w nadajniku RC
int RC3stick = 0;          // położenie drążka 3 w nadajniku RC
unsigned int IC1 = 0;      // odczyt rejestru Capture1 z TIM3
unsigned int IC2 = 0;      // odczyt rejestru Capture2 z TIM3
unsigned int IC3 = 0;      // odczyt rejestru Capture3 z TIM3
unsigned int IC4 = 0;      // odczyt rejestru Capture4 z TIM3
```

W funkcji konfigurującej taktowanie systemu RCC_Config() musimy mieć włączone taktowanie portów GPIOA/B/C oraz liczników TIM3/4. W funkcji konfigurującej porty GPIO GPIO_Config(), linie portów PA6, PA7, PB0 oraz PB1 programujemy jako wejścia wysokiej impedancji GPIO_Mode_IN_FLOATING. Do funkcji inicjalizacji przerwań dopiszemy włączenie przerwań od TIM3.

```
// Włącz przerwanie od TIM3 ----------------------------------------------
NVIC_InitStructure.NVIC_IRQChannel = TIM3_IRQn;     // TIM3 global Interrupt
NVIC_InitStructure.NVIC_IRQChannelPreemptionPriority = 0;
NVIC_InitStructure.NVIC_IRQChannelSubPriority = 0;
NVIC_InitStructure.NVIC_IRQChannelCmd = ENABLE;
NVIC_Init(&NVIC_InitStructure);
```

Dotychczas mamy włączone przerwania od SysTick, od TIM3 – nadamy im najwyższy priorytet 0, od zegara RTC – priorytet niższy 2 i od klawiatury, priorytet najniższy 3. Kolejna czynność to konfiguracja licznika TIM3, którą dopiszemy do funkcji TIM_Config(). Konfiguracja samego licznika jest identyczna jak w przypadku TIM4, jedyna różnica to: TIM_Prescaler = 29-1 oraz TIM_Period = 0xFFFF. Wszystkie kanały licznika TIM3 skonfigurujemy jako wejściowe. Na początku funkcji TIM_Config() musimy zdefiniować strukturę:

```
TIM_ICInitTypeDef    TIM_ICInitStructure;
//Konfiguracja kanalu 1 ------------------------------------------------
TIM_ICInitStructure.TIM_Channel = TIM_Channel_1;    // numer kanalu
TIM_ICInitStructure.TIM_ICPolarity = TIM_ICPolarity_Rising;
TIM_ICInitStructure.TIM_ICSelection = TIM_ICSelection_DirectTI;
TIM_ICInitStructure.TIM_ICPrescaler = TIM_ICPSC_DIV1;
TIM_ICInitStructure.TIM_ICFilter = 0;    // filtr wyłączony
TIM_ICInit(TIM3, &TIM_ICInitStructure);
```

Kanał 3 programujemy identycznie jak kanał 1. Kanały 2 i 4 muszą reagować na zbocze opadające

```
.TIM_ICPolarity = TIM_ICPolarity_Falling;
```

Na koniec włączamy licznik `TIM3` (identycznie jak włączyliśmy `TIM4`) oraz zezwalamy na przerwania:

```
TIM_ITConfig(TIM3, TIM_IT_CC1, ENABLE); // włączenie przerwania od CC1/TIM3
```

Powyższą instrukcję musimy powtórzyć jeszcze trzy razy, zmieniając `CC1` kolejno na `CC2`, `CC3` oraz `CC4`. W pliku *stm32f10x_it.c* umieścimy program obsługi przerwania od licznika `TIM3`.

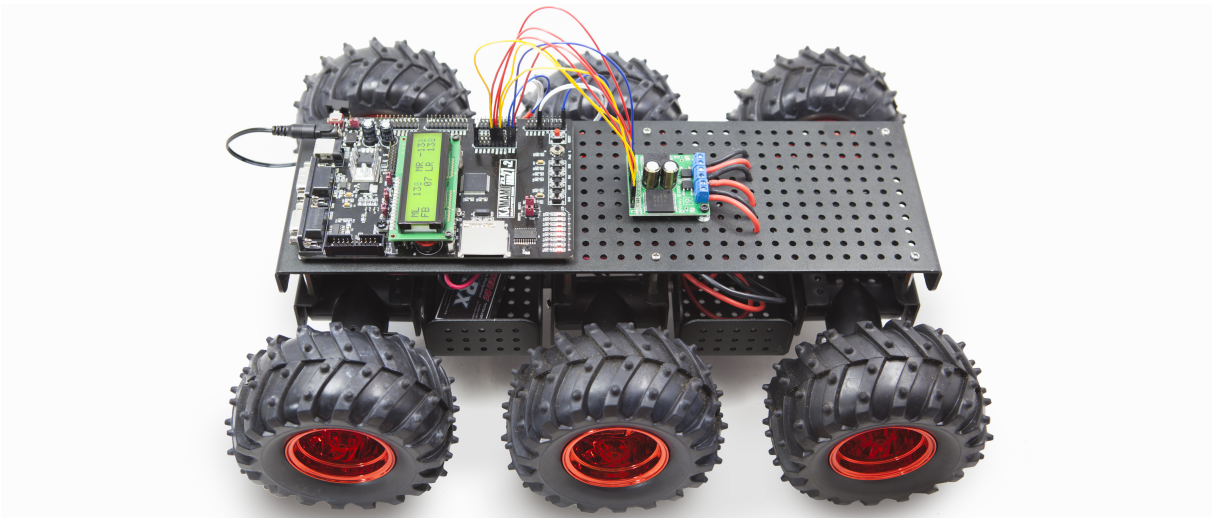
```
/*****************************************************************
* Function Name : TIM3_IRQHandler
* Description : This function handles TIM3 global interrupt request.
*****************************************************************/
extern int RC1stick = 0;        // położenie drążka 1 w nadajniku RC
extern int RC3stick = 0;        // położenie drążka 3 w nadajniku RC
extern unsigned int IC1 = 0;    // odczyt rejestru Capture1 z TIM3
extern unsigned int IC2 = 0;    // odczyt rejestru Capture2 z TIM3
extern unsigned int IC3 = 0;    // odczyt rejestru Capture3 z TIM3
extern unsigned int IC4 = 0;    // odczyt rejestru Capture4 z TIM3

void TIM3_IRQHandler(void){
   if(TIM_GetITStatus(TIM3, TIM_IT_CC1) != RESET){
      TIM_ClearITPendingBit(TIM3, TIM_IT_CC1);    //zeruj bit przerwania
      IC1 = TIM_GetCapture1(TIM3);
   }
   else if(TIM_GetITStatus(TIM3, TIM_IT_CC2) != RESET) {
      TIM_ClearITPendingBit(TIM3, TIM_IT_CC2);    //zeruj bit przerwania
      IC2 = TIM_GetCapture2(TIM3);
      RC1stick = IC2 - IC1;
      if(RC1stick < 0)
         RC1stick += 0x10000;
      RC1stick -= 3775;
   }
   else if(TIM_GetITStatus(TIM3, TIM_IT_CC3) != RESET){
      TIM_ClearITPendingBit(TIM3, TIM_IT_CC3);    //zeruj bit przerwania
```

```
      IC3 = TIM_GetCapture3(TIM3);
   }
   else if(TIM_GetITStatus(TIM3, TIM_IT_CC4) != RESET){
      TIM_ClearITPendingBit(TIM3, TIM_IT_CC4);    //zeruj bit przerwania
      IC4 = TIM_GetCapture4(TIM3);
      RC3stick = IC4 - IC3;
      if(RC3stick < 0)
      RC3stick += 0x10000;
      RC3stick -= 3775;
   }
}
```

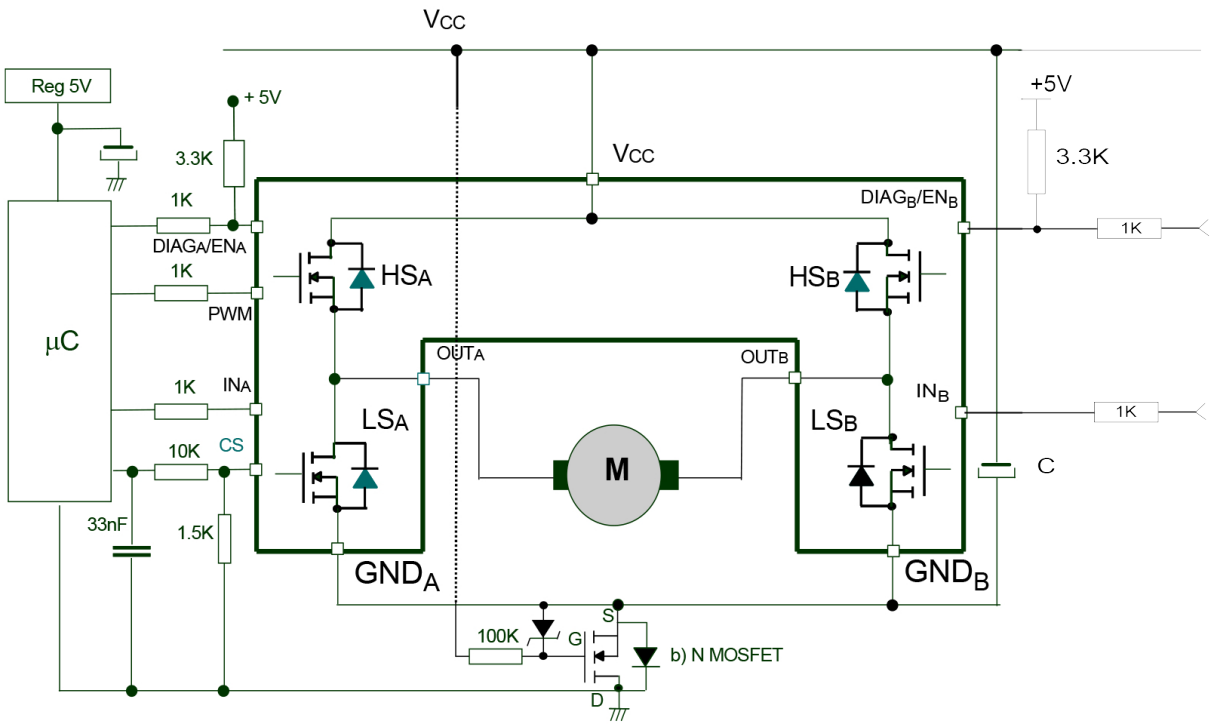
W pętli głównej (pętlą główną nazywamy pętlę `while(1)` w funkcji `main()`) dopiszemy instrukcje wyświeltania wartości obrazujących aktualne położenie drążków nadajnika (jak na rysunku 12.10). Kierunek działania drążków możemy odwrócić za pomocą funkcji *reverse* REV w nadajniku. Numeracja kanałów pokazana jest na rysunku 12.12.

12.7. Projekt11_6WheelDrive

Łącząc dwa ostatnie projekty bardzo łatwo możemy zbudować zdalnie sterowany model samochodu terenowego z napędem na sześć kół. Wykorzystamy do tego celu podwozie *Dagu Wild Thumper 6WD All-Terrain Chassis* (www.pololu.com) pokazane na rysunku 12.14. W osi każdego koła umieszczony jest komutatorowy silnik prądu stałego. To oznacza, że do zacisków silnika wystarczy przyłożyć napięcie stałe. Zmiana polaryzacji spowoduje zmianę kierunku obrotów. Relacja pomiędzy wartością napięcia, a obciążeniem silnika decyduje o obrotach. Do zmiany średniego napięcia zasilania silników wykorzystamy sygnał PWM. Do zmiany kierunku: mostek H. Będziemy sterować oddzielnie lewą i prawą stronę, to znaczy, że trzy silniki z lewej strony pojazdu połączymy równolegle i będą one sterowane jednym sygnałem. Podobnie trzy silniki z prawej strony. Do bezpośredniego wysterowania zastosujemy moduł *VNH2SP30 Motor Driver Carrier MD01B* firmy www.pololu.com. Potrzebne są dwa takie moduły: po jednym na każdą stronę pojazdu.



Rys. 12.14: Pojazd 6WD, napęd na 6 kół, z podwójnym modułem MD03A

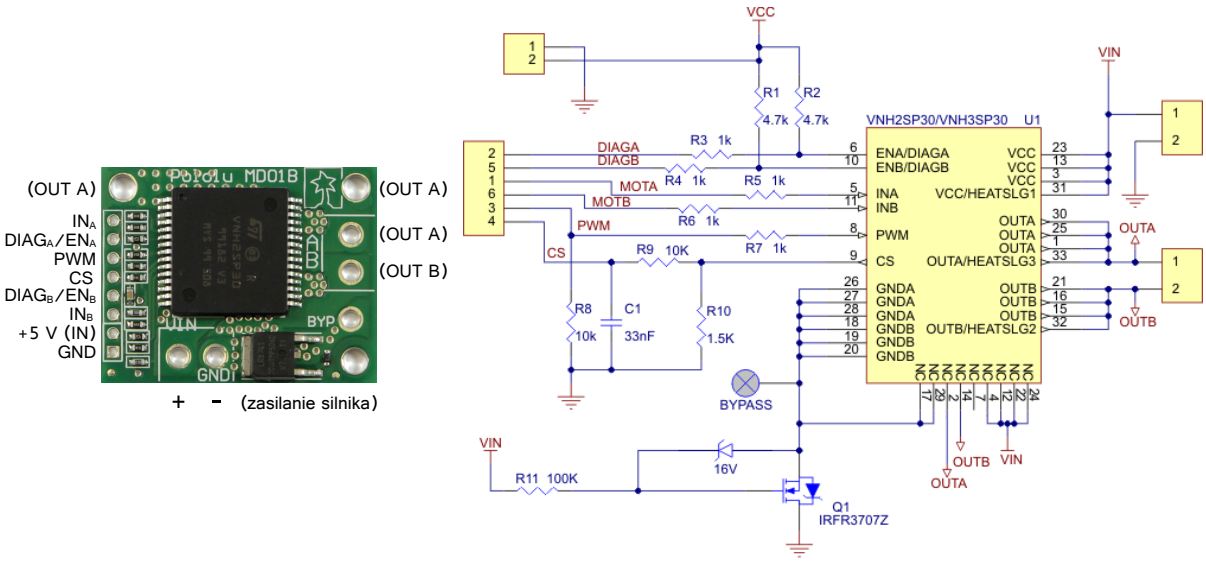


Rys. 12.15: Zasada działania mostka H

W module MD01B stosowany jest układ VNH2SP30, albo VNH3SP30. Schemat aplikacyjny pokazany jest na rysunku 12.15. Jednoczesne załączenie tranzystorów HS_A i LS_B spowoduje polaryzację zacisków silnika: OUT_A do zasilania V_{CC} i OUT_B do masy. Jednoczesne załączenie tranzystorów na drugiej przekątnej mostka, czyli HS_B i LS_A spowoduje przeciwną polaryzację silnika M. Niedopuszczalne jest załączenie tranzystorów z tej samej strony mostka (nawet chwilowe), czyli $HS_A + LS_A$ lub $HS_B + LS_B$, gdyż spowoduje to zwarcie zasilania do masy. Zabezpieczeniem jest tranzystor MOSFET (Q1 na rysunku 12.16) włączony pomiędzy układem scalonym a masą. Przekroczenie wartości granicznej prądu powoduje wyłączenie tego tranzystora i chroni układ przed spaleniem. Rysunek 12.16 zawiera schemat modułu MD01B oraz wyprowadzenia sygnałów w module MD01B:

Tabela 12.2: Funkcje sygnałów modułu MD01B

Symbol	**Funkcja**
OUT_A/OUT_B	Zasilanie elektrycznego silnika komutatorowego prądu stałego
GND_A/GND_B	Masy, strona A/strona B, muszą być zwarte i dołączone do zewnętrznego tranzystora zabezpieczającego
V_{CC}	Zasilanie silnika
$DIAG_A/EN_A$ $DIAG_B/EN_B$	Dwukierunkowe sygnały logiczne z otwartym drenem. Muszą być dołączone do zewnętrznych oporników podciągających. Gdy są zewnętrznie wymuszone do masy (EN_A/EN_B), wyłączają odpowiednią stronę (A/B) mostka H. W przypadku wykrycia błędu (na przykład termiczne wyłączenie tranzystorów strony A /B mostka), układ scalony wymusza stan niski sygnału $DIAG_A/DIAG_B$.
CS	Wyjście analogowe układu pomiaru prądu silnika.
PWM	Wejście sygnału PWM - maksymalnie 20 kHz (VNH3SP30: 10 kHz)
IN_A, IN_B	Wejścia logiczne sterujące: obroty w lewo, w prawo, hamowanie do masy, hamowanie do zasilania. Patrz tabela 12.3



Rys. 12.16: Schemat modułu MD01B

Tabela 12.3: Sygnały sterujące modułu MD01B

IN_A	IN_B	$DIAG_A/EN_A$	$DIAG_B/EN_B$	OUT_A	OUT_B	tryb pracy
1	1	1	1	H	H	hamowanie do V_{CC}
1	0	1	1	H	L	obrót w lewo
0	1	1	1	L	H	obrót w prawo
0	0	1	1	L	L	hamowanie do GND

W najprostrzym przypadku wystarczy sterować sygnałami IN_A, IN_B oraz podać sygnał PWM. Jak zwykle kopiujemy ostatni projekt i zmieniamy jego nazwę. Mamy już działający pomiar sygnałów z nadajnika RC. Do podwozia montujemy dwa moduły MD01B. Do jednego podłączamy trzy silniki z prawej strony pojazdu, do drugiego trzy z lewej strony. Dwa sygnały PWM wygenerujemy za pomocą kanałów 3 i 4 licznika `TIM4`, jak w *Projekt08_toggleLED*. Dzięki temu będziemy mieli podgląd na diodach LED1 i LED2 – intensywność świecenia będzie miarą wypełnienia sygnałów PWM. Do sterowania kierunkiem obrotów wykorzystamy porty `PB10, PB11, PB12, PB13`, co umożliwi podgląd stanu sygnałów sterujących za pomocą diod LED3/4/5/6 w płytce ZL27ARM. Należy wykonać połączenia pomiędzy modułami MD01B i płytą ZL27ARM zgodnie z tabelą 12.4.

Tabela 12.4: Połączenia pomiędzy płytą ZL27ARM i modułami MD01B

ZL27ARM	MD01B lewy	MD01B prawy	LED	Licznik
3.3V	+5V(IN)	+5V(IN)		
GND	GND	GND		
PB8	PWM		LED1	TIM4_CH3
PB9		PWM	LED2	TIM4_CH4
PB10	IN_A		LED3	
PB11	IN_B		LED4	
PB12		IN_A	LED5	
PB13		IN_B	LED6	

Maksymalna częstotliwość PWM, dla układu VNH2SP30, to 20 kHz. Zastosujemy częstotliwość nieznacznie mniejszą. Istotne jest aby częstotliwość PWM była możliwie najwyższa, w pobliżu górnej granicy pasma słyszalnego. W przeciwnym wypadku silnik będzie wydawał charakterystyczny pisk. Chcemy ponadto zachować rozdzielczość aparatury do zdalnego sterowania RC. Dlatego licznik powinien być taktowany taką częstotliwością, aby 1043 takty składały się na 20 kHz:

$$1043 \cdot 20'000 = 20'860'000 \quad \rightarrow \quad \frac{72'000'000}{20'860'000} = 3.45 \quad \rightarrow \quad \frac{72'000'000}{4} = 18'000'000$$

Licznik `TIM4` będziemy taktować 18 MHz, częstotliwość sygnału PWM będzie równa

$$f_{PWM} = \frac{18'000'000}{1043} = 17.258\ kHz$$

Jeżeli został zastosowany moduł MD01B z układem VNH3SP30, o maksymalnej częstotliwości sygnału PWM 10 kHz, to wyliczenie jest następujące:

$$1043 \cdot 10'000 = 10'430'000 \quad \rightarrow \quad \frac{72'000'000}{10'430'000} = 6.9 \quad \rightarrow \quad \frac{72'000'000}{7} = 10'285'714$$

Częstotliwość sygnału PWM będzie wtedy równa

$$f_{PWM} = \frac{10'285'714}{1043} = 9\ 861.663\ Hz$$

Zmienimy konfigurację licznika `TIM4`

```
TIM_TimeBaseStructure.TIM_Prescaler = 4-1;  //taktowanie 72MHz/4 = 18 MHz
TIM_TimeBaseStructure.TIM_Period = 1043-1;  //zakres przepełnień 1043
```

Ponadto kanały 3 i 4 należy zaprogramować na PWM1. W pętli głównej, na podstawie sygnałów odbieranych z nadajnika RC wyliczymy sterowanie lewej strony pojazdu jako sumę kanału przód−tył i kanału lewo−prawo. Sterowanie prawej strony to różnica tych sygnałów. Oba wyniki musimy ograniczyć do zakresu $[-1043, +1043]$. Sygnały PWM to wartości bezwzględne obu wyników. Bity kierunku obrotów silników ustalamy na podstawie znaków obu wyników. Wpisy do rejestrów określających wypełnienia PWM ograniczymy do momentów, gdy nastąpiła zmiana wartości, dlatego zdefiniujemy zmienne "'stare" i "nowe". Na początku funkcji `main()` deklarujemy zmienne lokalne

```
int MotorL, MotorR;
unsigned int PWM_R, PWM_R_old, PWM_L, PWM_L_old;
while (1) {
   MotorL = RC1stick + RC3stick;
   if(MotorL > 1043)
      MotorL = 1043;
   else if(MotorL < -1043)
      MotorL = -1043;
   MotorR = RC3stick - RC1stick;
   if(MotorR > 1043)
      MotorR = 1043;
   else if(MotorR < -1043)
      MotorR = -1043;
```

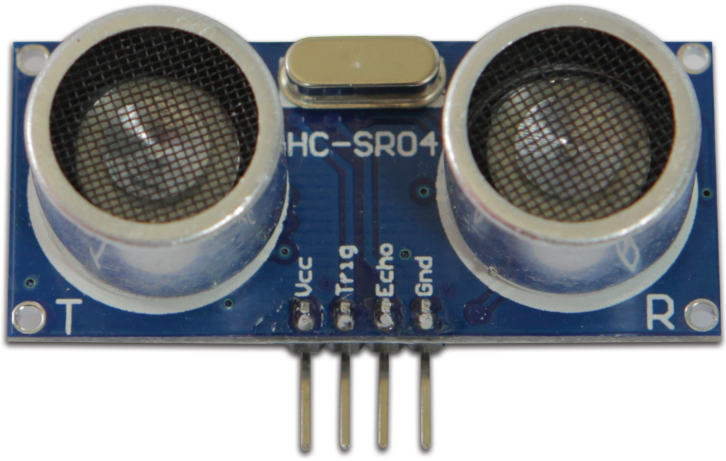
```
   // Motor Prawy
   PWM_R = MotorR > 0 ? MotorR : -MotorR;    // wartość bezwzględna
   if(PWM_R != PWM_R_old){
      PWM_R_old = PWM_R;
      TIM_SetCompare3(TIM4, PWM_R);
      if(MotorR > 0){
         GPIO_ResetBits(GPIOB, GPIO_Pin_11);
         GPIO_SetBits(GPIOB, GPIO_Pin_10);
      }
      else if(MotorR < 0){
         GPIO_ResetBits(GPIOB, GPIO_Pin_10);
         GPIO_SetBits(GPIOB, GPIO_Pin_11);
      }
   }
   // Motor Lewy
   PWM_L = MotorL > 0 ? MotorL : -MotorL;    // wartość bezwzględna
   if(PWM_L != PWM_L_old){
      PWM_L_old = PWM_L;
      TIM_SetCompare4(TIM4, PWM_L);
      if(MotorL > 0){
         GPIO_ResetBits(GPIOB, GPIO_Pin_13);
         GPIO_SetBits(GPIOB, GPIO_Pin_12);
      }
      else if(MotorL < 0){
         GPIO_ResetBits(GPIOB, GPIO_Pin_12);
         GPIO_SetBits(GPIOB, GPIO_Pin_13);
      }
   }
   // tu możemy wyświetlić sterowania ...
   delay_us(800000);
}
```

Funkcje drążków można zmienić dołączając odpowiednio kanały odbiornika do płyty ZL27ARM, zgodnie z numeracją kanałów pokazaną na rysunku 12.12. Na przykład dołączenie kanałów 1 i 2 spowoduje, że pojazd będzie sterowany tylko prawym drążkiem. Dołączenie kanałów 1 i 3 to: lewy drążek przód−tył i prawy lewo−prawo.

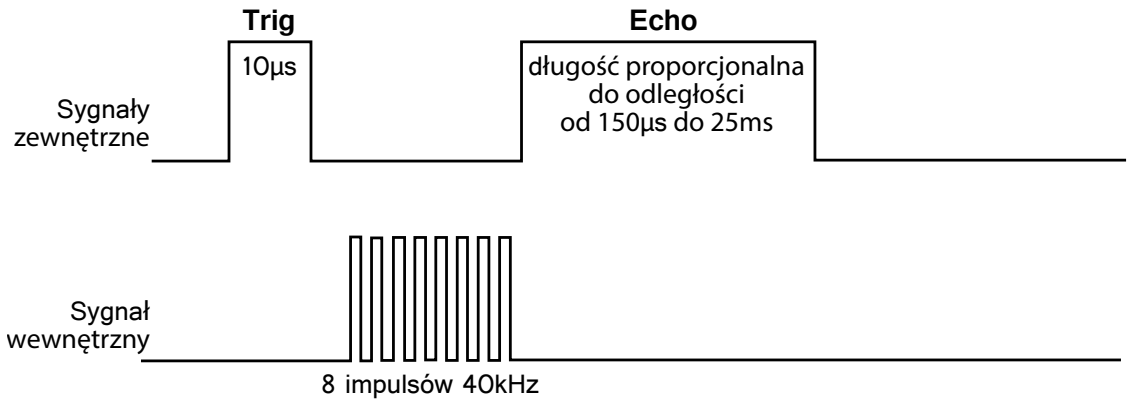
12.8. Projekt12_DistanceSensor, dalmierz ultradźwiękowy

Dalmierz ultradźwiękowy zawiera nadajnik i odbiornik. Nadajnik wysyła krótki sygnał o częstotliwości 40 kHz, który po odbiciu od przeszkody trafia do odbiornika. Droga przebyta przez falę (ultradźwiękową): od nadajnika do przeszkody i od przeszkody do odbiornika jest równa podwojonej odległości czujnika od przeszkody. Zgodnie ze wzorem "droga = prędkość × czas", odległość czujnika od przeszkody to połowa iloczynu prędkości dźwięku w powietrzu v i czasu t jaki upłynął od wysłania impulsu do odebrania czoła fali odbitej.

$$D = \frac{v \cdot t}{2}$$



Rys. 12.17: Ultradźwiękowy czujnik odległości HC-SR04



Rys. 12.18: Sygnały

Dla dokładności pomiaru krytyczna jest wartość prędkości dźwięku w powietrzu, która zależy od gęstości powietrza, czyli od temperatury, ciśnienia i wilgotności. Najsilniejszy jest wpływ temperatury. Przybliżony wzór jest następujący.

$$v\ [m/s] = 331.5\ \sqrt{\frac{t\ [^\circ C] + 273.15}{273.15}}$$

Niestety, nie mamy jeszcze uruchomionego żadnego pomiaru temperatury, dlatego wpiszemy stałą prędkość dźwięku. W temperaturze $18^\circ C$ jest ona równa około $v_{18} = 342.25\ m/s$.

Wykorzystamy czujnik HC-SR04, rysunek 12.17. Zasilimy go z płyty napięciem 3.3 V. Sposób działania pokazany jest na rysunku 12.18. Impuls o długości 10 μs podany na wejście *Trig* powoduje wysłanie ośmiu impulsów o częstotliwości 40 kHz. Czujnik odpowiada sygnałem na wyjściu *Echo*. Czas trwania stanu wysokiego sygnału *Echo* jest proporcjonalny do odległości od przeszkody i zawiera się w przedziale od 150 μs do 25 ms. Jeżeli czujnik nie wykryje żadnej przeszkody, to stan wysoki trwa 38 ms. Dystans wyrazimy w centymetrach, czas stanu wysokiego sygnału *Echo* w mikrosekundach. Założymy, że temperatura powietrza jest równa $18^\circ C$.

$$d_{18}\ [cm] = \frac{100 \cdot v_{18}\ [m/s] \cdot t \cdot 10^{-6}\ [\mu s]}{2} = \frac{t\ [\mu s]}{58.4371}$$

Zmierzony czas stanu wysokiego sygnału *Echo* podzielimy przez 58.4371 i otrzymamy odległość od przeszkody wyrażoną w centymetrach. Zakres pomiarowy czujnika jest więc równy

$$d_{min} = \frac{150}{58.4371} = 2.57\ [cm] \qquad \longrightarrow \qquad d_{max} = \frac{25000}{58.4371} = 427.81\ [cm]$$

Kopiujemy ostatni projekt i zmieniamy jego nazwę. Wykorzystamy licznik `TIM1`. Kanał czwarty, jako jedyny, nie posiada na wyjściu układu opóźniającego, dlatego wykorzystamy go do generowania impulsu *Trig*. Kanały 1 i 2 posłużą do pomiaru czasu trwania odpowiedzi *Echo*. W `RCC_Config()` musimy załączyć taktowanie licznika `TIM1`

```
RCC_APB2PeriphClockCmd(RCC_APB2Periph_TIM1,ENABLE);//taktowanie TIM1 z APB2
```

W `TIM_Config()` konfigurujemy `TIM1` z preskalerem `.TIM_Prescaler = 72-1` – uzyskamy taktowanie licznika 1 MHz, czyli co 1 μs. Rejestr przeładowania `ARR` ustawimy na wartość maksymalną `.TIM_Period = 0xFFFF`. Kanał czwarty zaprogramujemy jako `TIM_OCMode_PWM1` i rejestr `CCR4` na wartość `TIM_Pulse = 10`. Na wyjściu kanału uzyskamy impuls 10 μs powtarzany z okresem 0.065536 sekundy. Zgodnie z tabelą 12.1, wyjście czwartego kanału, to linia `PA11`. W konfiguracji GPIOA należy `PA11` zaprogramować jako `GPIO_Mode_AF_PP`. Kanały 1 i 2 konfigurujemy identycznie, jak w liczniku `TIM3` – do pomiaru czasu trwania stanu wysokiego impulsu.

Jest istotna różnica pomiędzy licznikami ogólnego przeznaczenia `TIM2, TIM3, TIM4`, a licznikiem zaawansowanym `TIM1`: trzy pierwsze kanały `TIM1` mają programowane układy opóźniające. Kanał używany jako wyjściowy PWM musi mieć zezwolone wyjście. Osiągniemy to wykonując poniższe instrukcje:

```
TIM_BDTRStructInit(&TIM_BDTRInitStruct);
TIM_BDTRConfig(TIM1, &TIM_BDTRInitStruct);
TIM_CtrlPWMOutputs(TIM1, ENABLE);
```

Podobnie jak w przypadku `TIM3`, przy pomiarze czasu trwania sygnałów z odbiornika RC, musimy zezwolić na obsługę przerwań od `CC1` i `CC2` oraz załączyć przerwania od `TIM1` w NVIC. W pliku *stm32f10x_it.c* umieścimy pomiar długości trwania stanu wysokiego w obsłudze przerwania

```
void TIM1_CC_IRQHandler(void){
...
}
```

W pętli głównej możemy umieścić program wyświetlania zmierzonego czasu, oraz wyliczoną odległość od przeszkody.

Rozdział 13

Płyta STM3210E-EVAL, silniki krokowe

Kolejne projekty będziemy realizować z wykorzystaniem płyty STM32F10E-EVAL. Projekty będą miały nadawane kolejne numery i umieścimy je w nowym katalogu. Jeżeli będzie istniała możliwość realizacji określonego projektu w wersji na płytę ZL27ARM, to dla wygody tych Czytelników, którzy nie chcą zmieniać płyty, taka wersja projektu (o tym samym numerze i nazwie) będzie umieszczona w katalogu *ARM1CortexM3-ZL27ARM*. Nowa płyta jest wyposażona w wyświetlacz graficzny 240×320. Oczywiście, projekty dotyczące obsługi wyświetlacza graficznego nie mogą być powtórzone w wersji ZL27ARM.

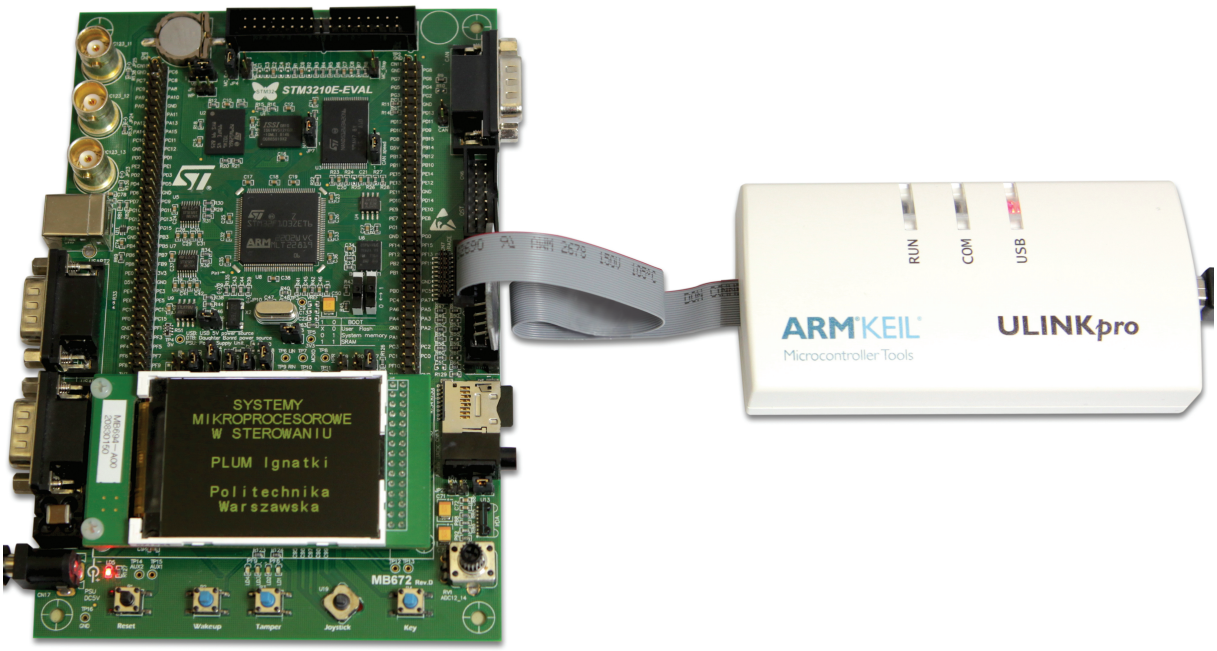
Płyta STM32F10E-EVAL jest wyposażona w procesor STM32F103ZET6, natomiast płyta ZL27ARM w procesor STM32F103VBT6.

Tabela 13.1: Porównanie procesorów z płyt ZL27ARM i STM3210E-EVAL

Układy Peryferyjne		**STM32F103VB**	**STM32F103ZE**
Pamięć Flash		128 kB	512 kB
Pamięć S-RAM		20 kB	64 kB
FSMC		–	1
Timers	General-Purpose	3	4
	Advanced-control	1	2
	Basic	–	2
Communinication	SPI	2	3
	I^2C	2	2
	USART	3	5
	USB	1	1
	CAN	1	1
	SDIO	–	1
Linie GPIO		80	112
Przetworniki ADC 12-bitowe		2	3
Ilość kanałów		16	21
Przetworniki DAC 12-bitowe		–	2
Ilość kanałów		–	2
Taktowanie CPU		72 MHz	72 MHz
Obudowa		LQFP100	LQFP144

Otrzymaliśmy dodatkowo dwa przetworniki cyfrowo-analogowe oraz moduł FSMC *Flexible Static Memory Controler*, który pomaga zarządzać zewnętrznymi pamięciami statycznymi. Płyta STM3210E_EVAL ma trzy takie pamięci:

- IS61WV51216BLL-10MLI – 512k×16 statyczny RAM (8 ns / 3.3V)
- M29W128GL 70ZA6 – NOR Flash 128 Mbit (16M×8 lub 8M×16)
- NAND512W3A2CN6 – NAND Flash 512 Mbit (×8 / ×16)



Rys. 13.1: Płyta STM32f10E-EVAL z programatorem ULINKpro

13.1. Katalog ARM2CortexM3-STM3210E-EVAL

W katalogu głównym, obok katalogu *ARM1CortexM3-ZL27ARM*, założymy nowy katalog *ARM2CortexM3-STM3210E-EVAL*, a w nim dwa podkatalogi:

ARM2CortexM3-STM3210E-EVAL
- *Dokumentacja*
- *Libraries*

Pracę z nową płytą zaczynamy od pobrania dokumentacji ze strony www.st.com/. Wyszukujemy "stm3210e-eval" i wchodzimy na stronę płyty. Z tej strony pobieramy specyfikację, noty aplikacyjne i podręczniki. Pobrane dokumenty umieszczamy w podkatalogu *Dokumentacja*. Następnie klikamy na link STM32CubeF1 i ze strony, która zostanie otwarta pobieramy bibliotekę programów STM32CubeF1, oraz dokumenty związane z tą biblioteką. Bibliotekę rozpakowujemy w katalogu *ARM2CortexM3-STM3210E-EVAL*. W tej bibliotece jest wiele przydatnych przykładów w podkatalogu *Projects\STM3210E_EVAL*.

Cofamy się na stronę płyty i pobieramy bibliotekę programu demonstracyjnego klikając na link STSW-STM32083. Pobrany plik rozpakowujemy. Do naszego podkatalogu *Libraries* przekopiujemy dwa: *CMSIS* i *STM32F10x_StdPeriph_Driver* z *\Libraries*. Natomiast trzeci: *STM32_EVAL*, przekopiujemy z *\Utilities*. Teraz zawartość naszego nowego katalogu powinna być następująca:

- *Dokumentacja*
- *Libraries*
  - *CMSIS*
  - *STM32_EVAL*
  - *STM32F10x_StdPeriph_Driver*
- *STM32Cube_FW_F1_V1.4.0*
- *STM3210E-EVAL_FW_V2.1.0*

13.2. Projekt01_LED

Procesor STM32F103ZET6 jest klasyfikowany jako *high-density*, ale większość układów wewnętrznych (RCC, RTC, GPIO, NVIC, ...) jest identyczna. Dlatego będziemy mogli wykorzystać wcześniej napisane funkcje. Kopiujemy projekt *Projekt01_LED* z katalogu *ARM1CortexM3-ZL27ARM* do obecnego *ARM2CortexM3-STM3210E-EVAL*. Usuwamy zawartości katalogów *Listings* i *Objects*. W *Keil µVision* otwieramy *projekt01*. Zaczniemy od zmiany opcji projektu, w zakładkach:

- *Device*: STM32F103ZE
- *Output, Name of Executable*: projekt01
- *Asm, Misc Controls*: – diag_warning 1546
- *C/C++, Define*: USE_STDPERIPH_DRIVER, STM3210X_HD

W oknie *Project*, w katalogu *RVMDK* usuwamy plik *startup_stm32f10x_md.s*, następnie dołączamy plik *startup_stm32f10x_hd.s* z katalogu *ARM2CortexM3-STM3210E-EVAL\ \Libraries\CMSIS\CM3\DeviceSupport\ST\STM32F10x\startup\arm*

Ponieważ ścieżki dostępu są określane względem bieżącego katalogu, to do projektu zostaną dołączone właściwe pliki. W pliku *main.c* konfiguracja *RCC_Config* pozostaje bez zmian, musimy tylko włączyć taktowanie portu GPIOF. W nowej płycie diody świecące LED są dołączone do portów PF6/7/8/9 i te linie musimy skonfigurować jako `GPIO_Mode_Out_PP`. W pętli głównej wpisujemy naprzemienne załączanie i wyłaczanie diod LED, oczywiście z programowym opóźnieniem. Cały projekt kompilujemy i wpisujemy do procesora.

Jak widać, dzięki temu że oba procesory są tego samego typu, wystarczyły niewielkie zmiany w projekcie, żeby uruchomić nową płytę. W następnym projekcie ożywimy wyświetlacz graficzny.

13.3. Projekt14_TFTLCD

Pełna nazwa to: *Thin-Film-Transistor Liquid-Cristal Display* TFT LCD, gdzie skrót TFT oznacza tranzystor cienkowarstwowy, czyli specjalny rodzaj tranzystora FET, który jest wykorzystywany głównie w kolorowych wyświetlaczach ciekłokrystalicznych z aktywnymi matrycami. TFT LCD to wyświetlacz sterowany cienkowarstwowymi tranzystorami unipolarnymi, pozwalającymi na uzyskanie w monitorach komputerowych i notebookach obrazów o najwyższej jakości. Cechą charakterystyczną tego typu tranzystorów jest fakt, iż umieszczane są na wspólnym szklanym podłożu, ponieważ ich pierwszym zastosowaniem były wyświetlacze ciekłokrystaliczne. To odróżnia je od standardowych tranzystorów, dla których typowym podłożem jest krzem. W wyświetlaczach opartych na TFT każdy z subpikseli (na piksel składają się trzy subpiksele: czerwony R, zielony G i niebieski B) ma własny tranzystor sterujący. Jest to podstawowa cecha, która pozwala na otrzymywanie obrazu bardzo dobrej jakości.

W naszej płycie zainstalowany jest wyświetlacz firmy Ampire, typu AM240320L8 o przekątnej ekranu 2.4 cala, z matrycą RGB o rozdzielczości 240×320. Ma wbudowany kontroler ILI9320. Kolor RGB jest kodowany na 16 bitach w formacie Red – 5 bitów, Green – 6 bitów i Blue – 5 bitów. Wyświetlacz ma wbudowane podświetlenie w postaci czterech białych diod LED.

Kopiujemy ostatni projekt (Projekt01_LED) i zmieniamy jego nazwę. Otwieramy ten nowy projekt. Zaczniemy od dołączenia potrzebnych plików. W oknie *Project*, do katalogu *StdPeriphDrv* dodamy pliki z *\Libraries\STM32F10X_StdPeriph_Driver\src*. Po dodaniu, zawartość tego katalogu powinna być następująca:

- *misc.c*
- *stm32f10x_dma.c*
- *stm32f10x_exti.c*
- *stm32f10x_flash.c*
- *stm32f10x_fsmc.c*
- *stm32f10x_gpio.c*
- *stm32f10x_i2c.c*
- *stm32f10x_rcc.c*
- *stm32f10x_sdio.c*
- *stm32f10x_spi.c*
- *stm32f10x_usart.c*

Funkcje obsługi wyświetlacza są w bibliotece *STM3210E-EVAL*, którą mamy w katalogu *Library*. Odpowiednie pliki musimy dołączyć do projektu. W oknie *Project* tworzymy nowy podkatalog *STM3210E_EVAL* do którego dodajemy pliki z *\Libraries\STM32_EVAL\ \STM3210E_EVAL*.

- *stm32f10e_eval.c*
- *stm32f10e_eval_fsmc_nand.c*
- *stm32f10e_eval_fsmc_nor.c*
- *stm32f10e_eval_fsmc_sram.c*
- *stm32f10e_eval_i2c_tsensor.c*
- *stm32f10e_eval_lcd.c*
- *stm32f10e_eval_sdio_sd.c*
- *stm32f10e_eval_spi_flash.c*

Musimy dodać ścieżki dostępu w opcjach projektu. Po zmianach:

..\Libraries\CMSIS\CM3\CoreSupport
..\Libraries\CMSIS\CM3\DeviceSupport\ST\STM32F10x
..\Libraries\STM32_EVAL\STM3210E_EVAL
..\Libraries\STM32_EVAL\Common
..\Libraries\STM32Fx_StdPeriph_Driver\src
..\Libraries\STM32Fx_StdPeriph_Driver\inc
..\Projekt15_TFT

Otwieramy plik `main.c`. Na początku dołączymy plik nagłówkowy z funkcjami obsługi wyświetlacza

```
#include "stm3210e_eval_lcd.h"
```

Po inicjalizacji układów RCC i GPIO dodamy inicjalizację wyświetlacza

```
STM3210E_LCD_Init();
```

Red					Green						Blue					
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
1	1	1	1	1	0	0	0	0	0	0	0	0	0	0	0	0xF800 Red
0	0	0	0	0	1	1	1	1	1	1	0	0	0	0	0	0x07E0 Green
0	0	0	0	0	0	0	0	0	0	0	1	1	1	1	1	0x001F Blue
1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	0xFFFF White
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0x0000 Black

Rys. 13.2: Kodowanie kolorów RGB 5 - 6 - 5

13.3.1. Kolory

Wyświetlacz graficzny AM240320L8 zainstalowany w płycie wykorzystuje kodowanie kolorów 16-bitowe RGB, w systemie 5 - 6 - 5, jak pokazano na rysunku 13.2. Możemy wyświetlić określony kolor, stosując polecenie

```
LCD_Clear(Blue);
```

W pliku *stm32_eval_legacy.h* zdefiniowane są następujące kolory: `White, Black, Grey, Blue, Blue2, Red, Magenta, Green, Cyan, Yellow`. Możemy stworzyć własny kolor korzystając z dowolnego programu graficznego. Na przykład *Paint* i *Edytowanie kolorów* lub *Photo Shop* i *Próbnik kolorów*.

Na przykład, utworzymy własny kolor który nazwiemy `Brown` o parametrach odczytanych z próbnika kolorów:

R: 238 = 0xEE

G: 122 = 0x7A

B: 19 = 0x13

Bity kodów kolorów (każdy kolor to osiem bitów) wpisujemy, od najstarszego, w pola kolorów, jak na rysunku poniżej

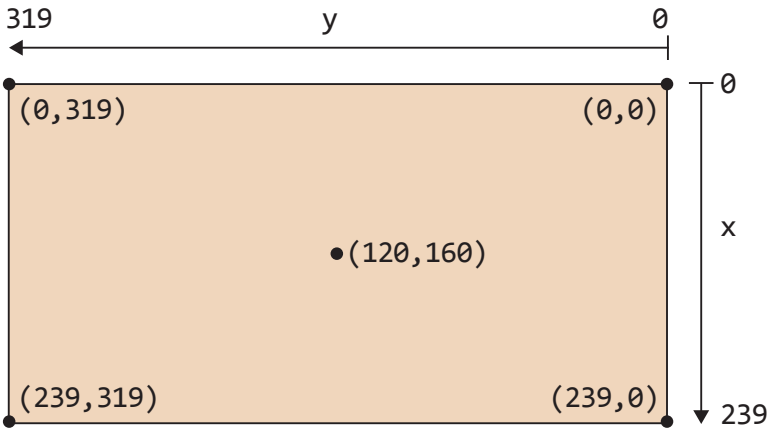
Red					Green						Blue								
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0				
1	1	1	0	1	1	1	0												0xEE
					0	1	1	1	1	0	1	0							0x7A
											0	0	0	1	0	0	1	1	0x13
1	1	1	0	1	0	1	1	1	1	0	0	0	0	1	0				0xEBC2

W pliku *main.c* możemy wpisać definicję: `#define Brown 0xEBC2`

13.3.2. Tekst

Do wyświetlania komunikatów tekstowych wykorzystamy funkcje biblioteczne. Musimy zacząć od wybrania czcionki. Defininicje czcionek zawarte są w pliku *fonts.c*. Do dyspozycji mamy: `Font16x24`, `Font12x12`, `Font8x12` oraz `Font8x8`. Musimy także określić kolor tła i kolor znaków. Przykład wyświetlenia tekstu w linii piątej (linie numerowane są od zera):

```
LCD_Clear(Black);
LCD_SetFont(&Font16x24);
LCD_SetBackColor(Black);
LCD_SetTextColor(Yellow);
LCD_DisplayStringLine(Line5, (uint8_t *)" PLUM Ignatki ");
```



Rys. 13.3: Współrzędne pikseli na ekranie gaficznym

13.3.3. Grafika

Współrzędne pikseli na ekranie pokazane są na rysunku 13.3. Początek układu współrzędnych to prawy górny róg. Kolor rysowanych punktów / lini ustawiamy tą samą funkcją, która ustawiała kolor znaków. Podstawowe funkcje to rysowanie odcinka prostego i rysowanie okręgu:

`LCD_DrawUniLine(`x_1, y_1, x_2, y_2`);`

`LCD_DrawCircle(`x_1, y_1, r`)`

Te i inne funkcje graficzne zawarte są w pliku *stm3210e_eval_lcd.c*.

13.3.4. Zdjęcia i obrazki

Zdjęcie przeznaczone do wyświetlenia należy przekształcić w programie graficznym, takim jak *Photo Shop*. Należy zmienić rozmiar do formatu 320×240 pikseli i zapisać z kodowaniem kolorów RGB 5-6-5 jako bitmapę *.btm*. Przy zapisywaniu należy wybrać *Opcje BMP → Tryby zaawansowane → 16 bitów, R5G6B5*.
Uzyskany plik *obrazek.bmp* przekształcimy w plik *obrazek.c*. W tym celu, ze strony

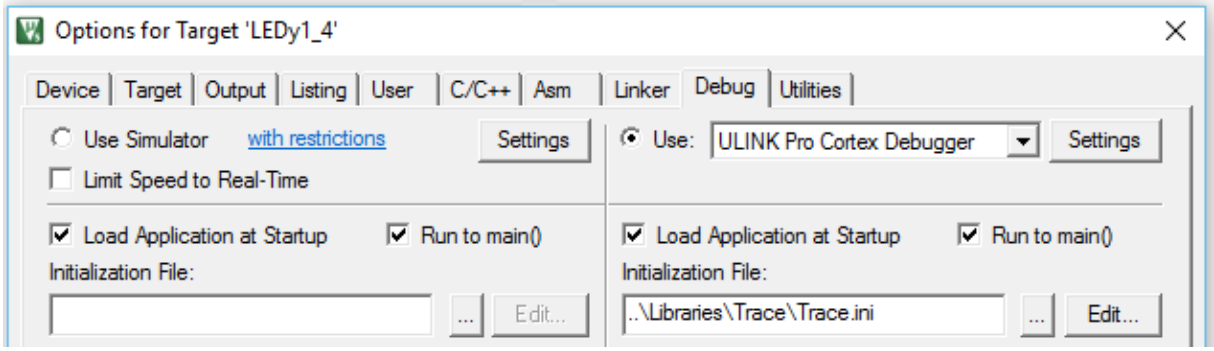
https://www.segger.com/bin2c.html

należy pobrać program *Bin2C.exe*. Po przekształceniu, obrazek dołączamy do projektu. Otwieramy plik *obrazek.c* i dopisujemy `const` przed deklaracją typu:

```
const unsigned char obrazek[] = {
```

Dzięki temu linker umieści obrazek w pamięci stałej (pamięć RAM jest zbyt mała). Zdjęcie możemy wyświetlić wywołując funkcję:

```
LCD_WriteBMP((uint32_t) obrazek);
```



Rys. 13.4: Zakładka *Debug* opcji projektu

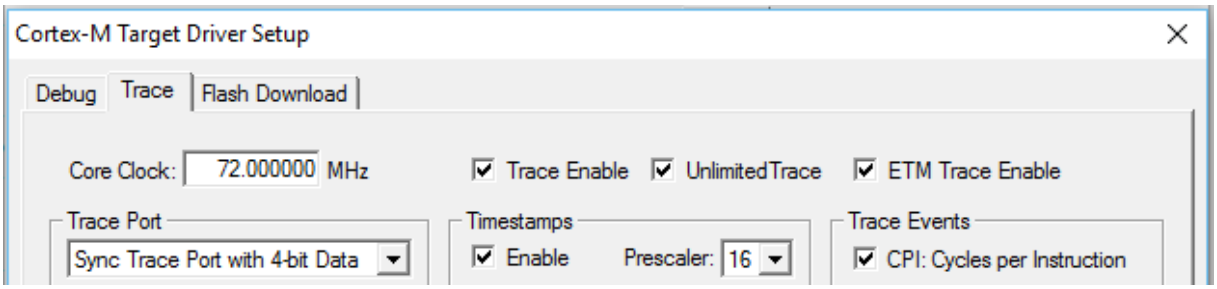
13.4. 4-Pin Trace ETM

Procesor STM32F103ZET6 wyposażony jest w układ śledzenia programu *Embedded Trace Macrocell* ETM. Jest to wszechstronne i bardzo silne narzędzie wspomagające tworzenie i testowanie oprogramowania. Płyta STM32F10E-EVAL zawiera złącze CN7 *trace debugging connector*. Do tego złącza należy podłączyć programator **ULINKpro** firmy ARM-KEIL (rysunek 13.1). Programator ULINKpro umożliwia śledzenie wykonywania instrukcji z częstotliwością do 200 MHz. W płycie STM32F10E-EVAL procesor taktowany jest z częstotliwością 72 MHz. Uruchomimy układ śledzenia i zbadamy opóźnienie programowe z instrukcjami _NOP.

Otwieramy program *Keil µVision*. Tworzymy nowy plik poleceniem *New*, wpisujemy

```
/*----------------------------------------------------------------------
** Define the function to enable the trace port
**----------------------------------------------------------------------*/
FUNC void EnableTPIU(void) {
// Enable trace port and clock signals
                        // RCC_AHB1ENR: IO port E clock enable
   _WDWORD(0x40023830, _RDWORD(0x40023830) | 0x00000010);
   _WDWORD(0x40021000, 0x00002AA0); //GPIOE_MODER:PE2..PE6=Alternate function
   _WDWORD(0x40021008, 0x00003FF0); //GPIOE_OSPEEDR:PE2..PE6=100 MHz speed
   _WDWORD(0x4002100C, 0x00000000); //GPIOE_PUPDR:PE2..PE6=No Pull-up/-down
   _WDWORD(0x40021020, 0x00000000); //GPIOE_AFRL:PE2..PE6=AF0
   _WDWORD(0xE0042004, 0x000000E0); // Set 4-pin tracing via DBGMCU_CR
}
/*----------------------------------------------------------------------
** Invoke the function at debugger startup
**----------------------------------------------------------------------*/
EnableTPIU();
/*----------------------------------------------------------------------
** Execute upon software RESET
**----------------------------------------------------------------------*/
FUNC void OnResetExec(void) {
   EnableTPIU();
}
```

W katalogu *Libraries* tworzymy podkatalog *Trace* i zapisujemy plik jako *Trace.ini*



Rys. 13.5: Zakładka *Trace* okna *Target Driver Setup*

Sygnały `TRACED[0..3]` są wyjściami na liniach `PE3` do `PE6`. Zegar śledzenia, `TRACECK`, jest dołączony do linii `PE2`. Po RESET, sygnały te nie są dołączone do pinów procesora i muszą być skonfigurowane przez ustawienie bitów `TRACE_IOEN` i `TRACE_MODE` w rejestrze `DBGMCU_CR`. Można to zrobić za pomocą komendy `_WDWORD(0xE0042004, 0x000000E0);` Specjalna funkcja `OnResetExec(void)` jest wykonywana po programowym RESET.

Otwieramy okno opcji projektu, wybieramy zakładkę *Debug* i dołączamy plik *Trace.ini* w oknie *Initialization File*, jak na rysunku 13.4.

W tym samym oknie zaznaczamy opcję *Use* i wybieramy (z listy) programator **ULINK Pro Cortex Debugger**. Klikamy *Settings* i pojawia się okno *Target Driver Setup*. Klikamy zakładkę *Trace*. W oknie *Core Clock* wpisujemy częstotliwość taktowania procesora, czyli 72 MHz. W oknie *Trace Port*, należy wybrać z listy opcję **Sync Trace Port with 4-bit Data**. Zaznaczamy opcje *Trace Enable, ETM Trace Enable* oraz *Unlimited Trace*, jak na rysunku 13.5. Zamykamy okna opcji projektu. Teraz możemy otworzyć sesję debugowania i śledzenia.

Dokładny opis funkcji śledzenia (programu i danych) z wykorzystaniem układu ETM zawarty jest w podręczniku *µVision User's Guide – Debugging*, link: *Code and Data Trace (Cortex-M)*, dostępnym na stronie *http://www.keil.com*.

13.4.1. Opóźnienie programowe

Wykorzystamy narzędzie śledzenia do dokładniejszej analizy działania pętli opóźnienia programowego. Śledzenie umożliwia pomiar czasu wykonywnia poszczególnych instrukcji:

```
                                          21:          delay = 100000000;
                                0.028 us 0x080006CA 4C26      LDR      r4,[pc,#152]
                                          22:          while (delay--){
  // test opoznienia programowego           23:          }
 0.028 us   delay = 100000000;
12.500  s   while (delay--){          0.028 us 0x080006CC BF00      NOP
            }                         3.603  s 0x080006CE 1E20      SUBS     r0,r4,#0
                                      3.603  s 0x080006D0 F1A40401  SUB      r4,r4,#0x01
                                      5.293  s 0x080006D4 D1FB      BNE      0x080006CE
```

Rys. 13.6: Czasy wykonania funkcji *Delay* bez instrukcji `NOP`

Rysunek 13.6 pokazuje czasy wykonania funkcji *Delay* bez instrukcji `NOP`. Z lewej strony jest postać źródłowa, z prawej kod maszynowy z deasemblacją. Łączny czas stu milionów wykonań to 12.5 sekundy. Będziemy dodawali kolejno po jednej instrukcji `NOP` i sprawdzimy, jak to wpływa na czas wykonania. Wyniki pokazano na rysunku 13.7. Uzyskaliśmy następujące wyniki: jeden `NOP` – czas identyczny jak w wersji bez `NOP`, dwie instrukcje `NOP`, czas 13.889 s. Trzy i cztery `NOP` dają identyczny czas 16.667 sekundy. Pięć `NOP` to 18.056 i sześć to 20.833 sekundy. Taka rozbieżność wyników jest spowodowana

sposobem działania procesora: instrukcja NOP jest 16-bitowa. Procesor pobiera 32 bity jednocześnie, więc na przykład dwie instrukcje NOP. Pobrane instrukcje trafiają do potoku (rysunek 3.12). Jedna instrukcja NOP może być "'połknięta" i w potoku kolejne instrukcje są dosuwane. Dlatego funkcja *Delay()* ma taki sam czas wykonania w wersji bez NOP, jak i z jednym NOP. Podobnie jest w przypadku trzy i cztery, gdy są cztery NOP, to jeden jest "połykany" i czas jest identyczny jak dla wersji z trzema. Widać, że dobre ilości to dwa, trzy, pięć i sześć. Natomiast nie ma sensu umieszczanie jednej albo czterech instrukcji NOP w pętli opóźnienia programowego *Delay()*.

```
                 // test opoznienia programowego 1NOP
   0.042 us      delay = 100000000;
   0.042 us      while (delay--){
  12.500  s        __NOP();
                 }
                 // test opoznienia programowego 2NOP
   0.028 us      delay = 100000000;
   0.028 us      while (delay--){
   2.418  s        __NOP();
  11.471  s        __NOP();
                 }
                 // test opoznienia programowego 3NOP
   0.028 us      delay = 100000000;
   0.028 us      while (delay--){
   2.237  s        __NOP();
   2.237  s        __NOP();
  12.192  s        __NOP();
                 }
                 // test opoznienia programowego 4NOP
   0.028 us      delay = 100000000;
   0.028 us      while (delay--){
   2.231  s        __NOP();
   2.377  s        __NOP();
   2.387  s        __NOP();
   9.672  s        __NOP();
                 }
                 // test opoznienia programowego 5NOP
   0.028 us      delay = 100000000;
   0.028 us      while (delay--){
   2.262  s        __NOP();
   2.269  s        __NOP();
   2.254  s        __NOP();
   2.260  s        __NOP();
   9.011  s        __NOP();
                 }
                 // test opoznienia programowego 6NOP
   0.028 us      delay = 100000000;
   0.028 us      while (delay--){
   3.883  s        __NOP();
   1.607  s        __NOP();
   1.607  s        __NOP();
   3.396  s        __NOP();
   1.510  s        __NOP();
   8.830  s        __NOP();
                 }
```

Rys. 13.7: Czasy wykonania funkcji *Delay* z instrukcjami NOP

13.5. Silnik krokowy

Silnik krokowy przetwarza sygnał sterujący bezpośrednio na ustalone położenie wirnika. Droga kątowa, którą przebywa wirnik jest proporcjonalna do liczby impulsów, a prędkość obrotowa do częstotliwości tych impulsów. Silniki krokowe są stosowane tam, gdzie kluczowe znaczenie ma precyzyjne sterowanie położeniem wału silnika:

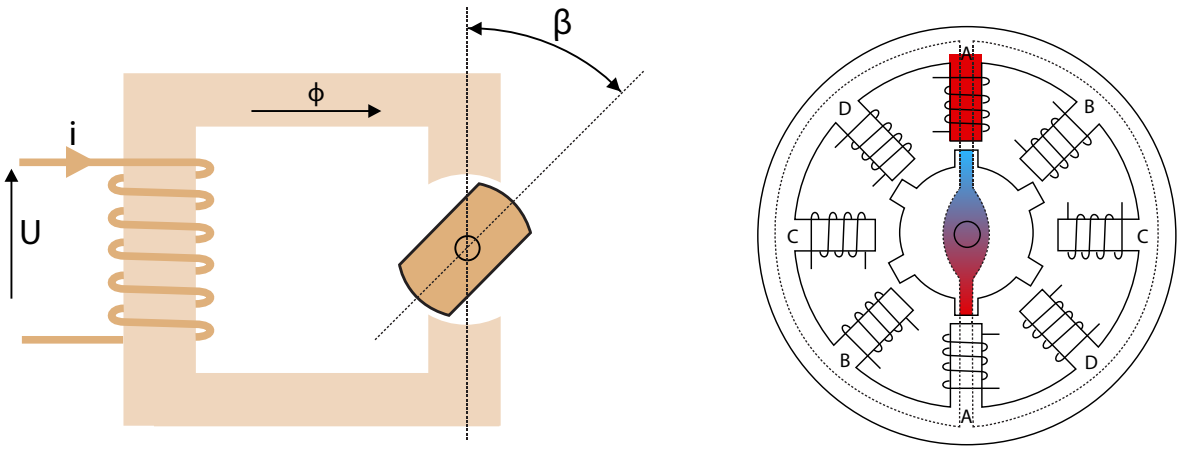
- w automatyce – w mechanicznych urządzeniach regulacyjnych (np. elektrozawory),
- w urządzeniach pomiarowych np. w zegarach do przesuwania wskazówek,
- w maszynach – do ustalania położenia głowicy frezarki
- w robotyce – do sterowania ruchem ramion robotów, kół w podwoziach robotów
- w komputerach – do napędu głowic w twardych dyskach, w ploterach i skanerach.

Dużą zaletą silników krokowych jest możliwość pracy z małymi prędkościami obrotowymi, przy czym silniki te dysponują cały czas pełnym momentem obrotowym. Dzięki temu zastępują stosowane wcześniej klasyczne silniki z przekładnią. Kolejną zaletą jest możliwość pracy w otwartej pętli. Nie potrzeba wówczas sprzężenia zwrotnego i informacji o aktualnym położeniu wirnika oraz kosztownych elementów, takich jak enkodery czy czujniki Halla. Najbardziej znane są trzy rodzaje silników krokowych: o zmiennej reluktancji, z magnesem stałym i hybrydowe. Obecnie produkowane są głównie silniki hybrydowe.

13.5.1. Silnik o zmiennej reluktancji VR – *variable reluctance*

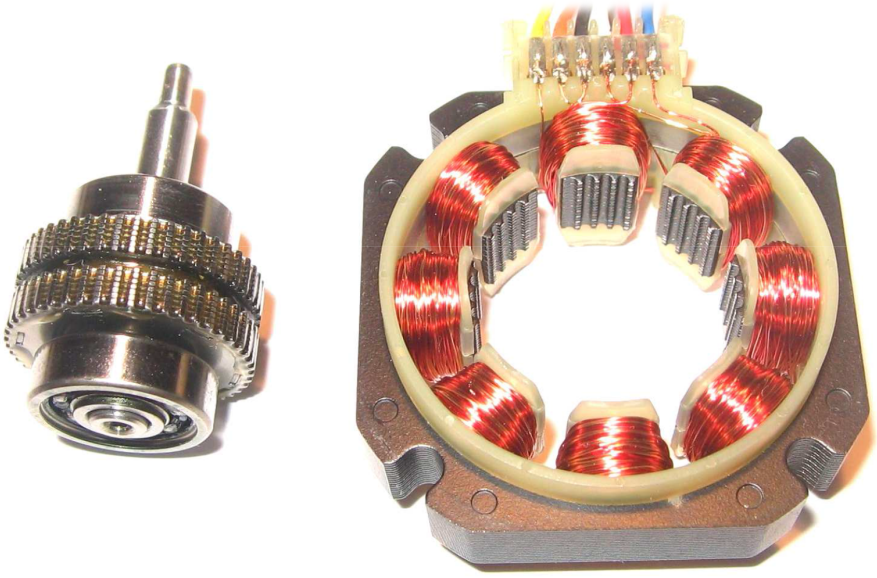
Reluktancja jest to parametr strumienia magnetycznego analogiczny do rezystancji elektrycznej, czyli jest to "rezystancja magnetyczna". Podobnie jak prąd wybiera drogę o najmniejszej rezystancji, tak linie sił pol magnetycznego skupiają się w obszarach o najmniejszej reluktancji. Powstawanie reluktancyjnego momentu obrotowego pokazano na rysunku 13.8a. Strumień indukcji Φ w obwodzie magnetycznym jest funkcją przepływu $(I \cdot N)$ oraz reluktancji (oporu magnetycznego) R_m, zgodnie ze wzorem

$$\Phi = \frac{I \cdot N}{R_m}$$



(a) reluktancyjny moment obrotowy (b) silnik o zmiennej reluktancji

Rys. 13.8: Silnik VR o zmiennej reluktancji



Rys. 13.9: Silnik hybrydowy HB – *hybrid motor*

gdzie: I – prąd, N – ilość zwojów. Reluktancja R_m jest najmniejsza gdy kąt wychylenia wirnika β jest zerowy. Jeżeli obrócimy ruchomy element rdzenia o kąt β, powstanie moment obrotowy, zwany reluktancyjnym, dążący do ustawienia wirnika w położeniu $\beta = 0$.

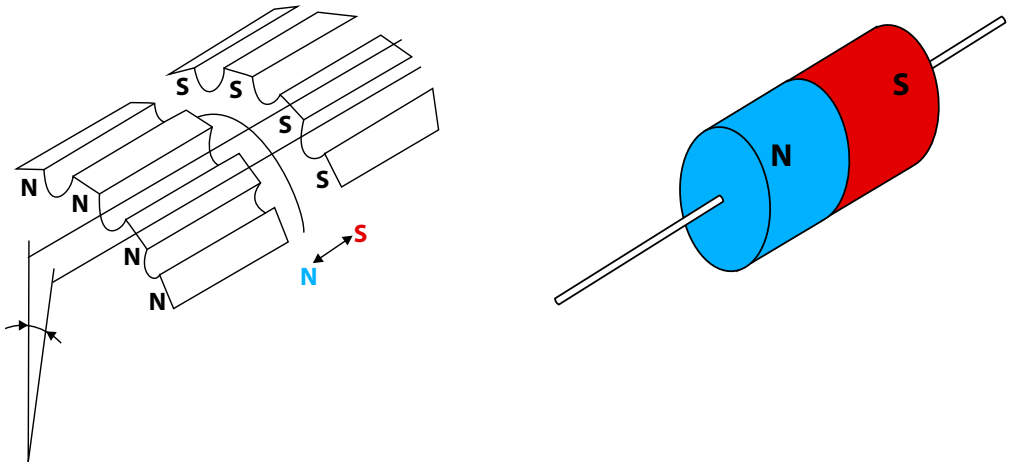
Silnik o zmiennej reluktancji ma wirnik wykonany z miękkiej stali z naciętymi charakterystycznymi zębami – rysunek 13.8b. Przepływ prądu stałego przez uzwojenie powoduje, że zęby wirnika ustawiają się naprzeciw zasilanego uzwojenia. Ruch obrotowy powstaje wskutek naprzemiennego zasilania kolejnych uzwojeń. W silnikach VR nie występuje zmiana kierunku prądu w uzwojeniach, dlatego zwykle wykonane są jako 3-fazowe lub 4-fazowe.

13.5.2. Silnik z magnesem trwałym PM – *permanent magnet*

Wirnik silnika z magnesem trwałym nie ma zębów lecz jest namagnesowany naprzemiennie biegunami N i S tak, iż bieguny te są usytuowane w linii prostej równoległej do osi rotora. Namagnesowane bieguny rotora wpływają na zwiększenie indukcji magnetycznej, dlatego silniki z magnesem trwałym w porównaniu z silnikami o zmiennej reluktancji mają lepszą charakterystykę momentu siły na wale. Poza naprzemiennym sterowaniem uzwojeń występuje także zmiana kierunku przepływu prądu (sterowanie bipolarne). Silniki PM mają niewielką rozdzielczość kątową – typowy skok to 7.5° oraz 15°. Ich najważniejszą zaletą jest niski koszt.

13.5.3. Silnik hybrydowy HB – *hybrid motor*

Silniki krokowe hybrydowe są obecnie najczęściej stosowane. Posiadają one małą długość kroku, a więc dużą dokładność i precyzję działania. Są droższe niż silniki VR czy PM jednak charakteryzują się lepszymi parametrami takimi jak rozdzielczość, prędkość czy moment. Silnik HB łączy w sobie zalety silników VR i PM. Rotor ma wiele zębów jak silnik VR i posiada magnes trwały zwiększający moment i poprawiający stabilność wirnika jak silnik PM. Rysunek 13.9 przedstawia silnik hybrydowy 2-fazowy o znamionowej długości kroku 1.8° (200 kroków na jeden obrót).



(a) przesunięcie pakietów o 1/2 zęba (b) osiowe namagnesowanie wirnika

Rys. 13.10: Wirnik silnika hybrydowego: dwa przesunięte pakiety i magnes pomiędzy nimi

Hybrydowy silnik krokowy ma wirnik reluktancyjny, a ponadto magnes trwały, wzmacniający przepływ wywołany impulsem sterującym dla pożądanego ustawienia zębów. Wirnik składa się z dwu żłobkowanych pakietów i umieszczonym pomiędzy nimi magnesem trwałym namagnesowanym osiowo (rysunek 13.10b). Pakiety wirnika przesunięte są względem siebie o 1/2 zęba – rysunek 13.10a.

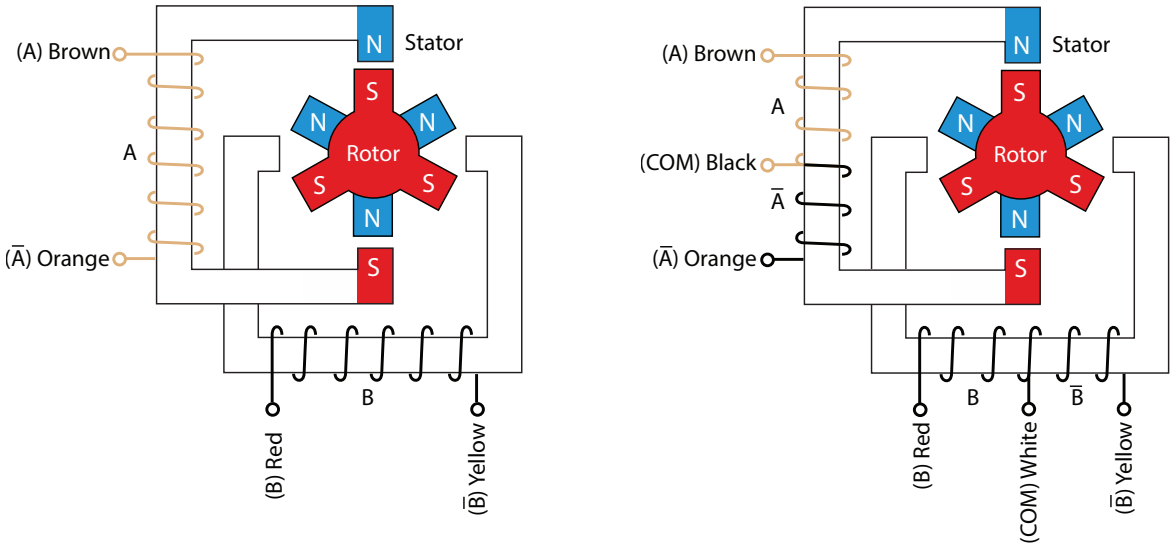
Jeden pakiet ma wszystkie zęby o biegunowości N, drugi pakiet ma zęby o biegunowości S. Jest to możliwe dzięki osiowemu namagnesowaniu wirnika – rysunek 13.10. Magnes trwały umieszczony w wirniku wytwarza strumień magnetyczny, zamykający się w obwodzie magnetycznym: stojan – szczelina powietrzna – wirnik. Po zasileniu uzwojenia stojana impulsem sterującym, strumień magnetyczny pod jednym biegunem stojana dodaje się do strumienia magnesów trwałych, pod drugim zaś – odejmuje. Wirnik przesunie się w położenie, przy którym osie zębów stojana i wirnika pokryją się. Silnik wykona jeden krok. Bieguny stojana są wzbudzane w odpowiedniej kolejności tak, aby zrealizować kolejne przemieszczenia kątowe.

13.5.4. Silniki krokowe unipolarne i bipolarne

Silniki krokowe bipolarne mają tylko jedno uzwojenie na fazę. Do poprawnej pracy wymagają dwóch polaryzacji napięcia (prądu) – stąd nazwa. Do zmiany kierunku przepływu prądu stosowane są mostki H, po jednym na fazę. Jak widać na rysunku 13.11a, dwufazowy silnik bipolarny ma cztery wyprowadzenia.

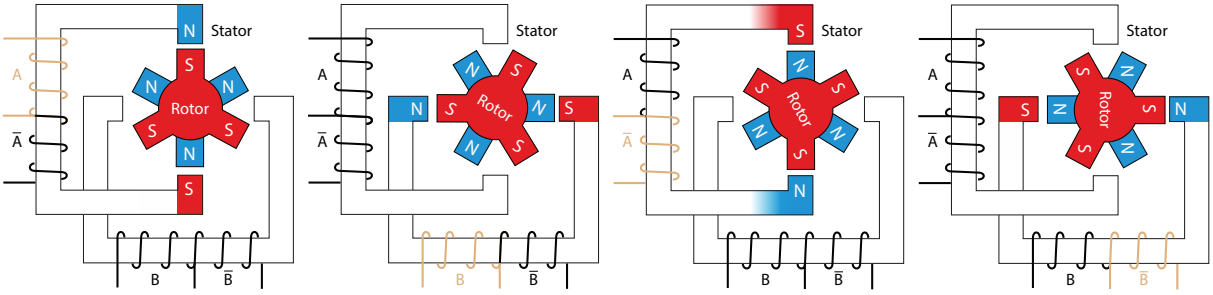
Silniki krokowe unipolarne mają po dwa uzwojenia na fazę, a właściwie to jedno uzwojenie z odczepem w środku. Sterowanie jest prostsze, nie wymaga mostków H, potrzebne są tylko dwa tranzystory na fazę. Jak widać na rysunku 13.11b, dwufazowy silnik unipolarny ma sześć wyprowadzeń.

Silnik unipolarny wymaga prostego sterowania (tylko cztery tranzystory), ale charakteryzuje go mniejszy moment siły na wale, gdyż w każdej chwili wykorzystywana jest



(a) silnik bipolarny (b) silnik unipolarny

Rys. 13.11: Silnik krokowy 2-fazowy (faza A i faza B), wirnik 3+3=6 zębów



Rys. 13.12: Sterowanie falowe silnika unipolarnego

tylko połowa uzwojeń. Silnik bipolarny to osiem tranzystorów (dwa mostki H), ale wykorzystywane są wszystkie uzwojenia i dostarczany moment jest większy. Silnik unipolarny można sterować bipolarnie – wtedy nie korzystamy z odczepów, wykorzystujemy tylko cztery skrajne wyprowadzenia.

Zaczniemy od sterowania silnika unipolarnego. Każde sterowanie unipolarne można łatwo przekształcić w wersję bipolarną: sterowanie drugiej połówki uzwojenia należy zamienić na załączenie tranzystorów na przeciwnej przekątnej mostka H.

13.5.5. Sterowanie falowe (jednofazowe)

W sterowaniu jednofazowym, zwanym inaczej falowym (*wave stepping*), w każdej chwili czasowej, zasilana jest tylko jedna faza. Jak widać na rysunku 13.12, sekwencja sterowań to

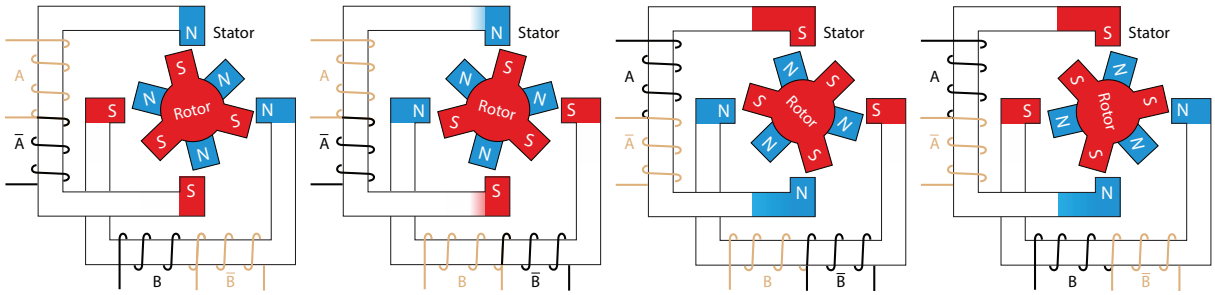
$$A \quad \rightarrow \quad B \quad \rightarrow \quad \bar{A} \quad \rightarrow \quad \bar{B}$$

Gdy sygnał sterujący ma stan wysoki, wtedy odpowiednia połówka uzwojenia jest zasilana i silnik wykonuje jeden krok. W silniku unipolarnym, uzyskiwany moment siły na wale silnika jest szczególnie mały, gdyż w każdej chwili czasowej zasilana jest tylko tylko jedna połówka z obu uzwojeń. Z tego powodu, sterowanie falowe praktycznie nie jest stosowane.

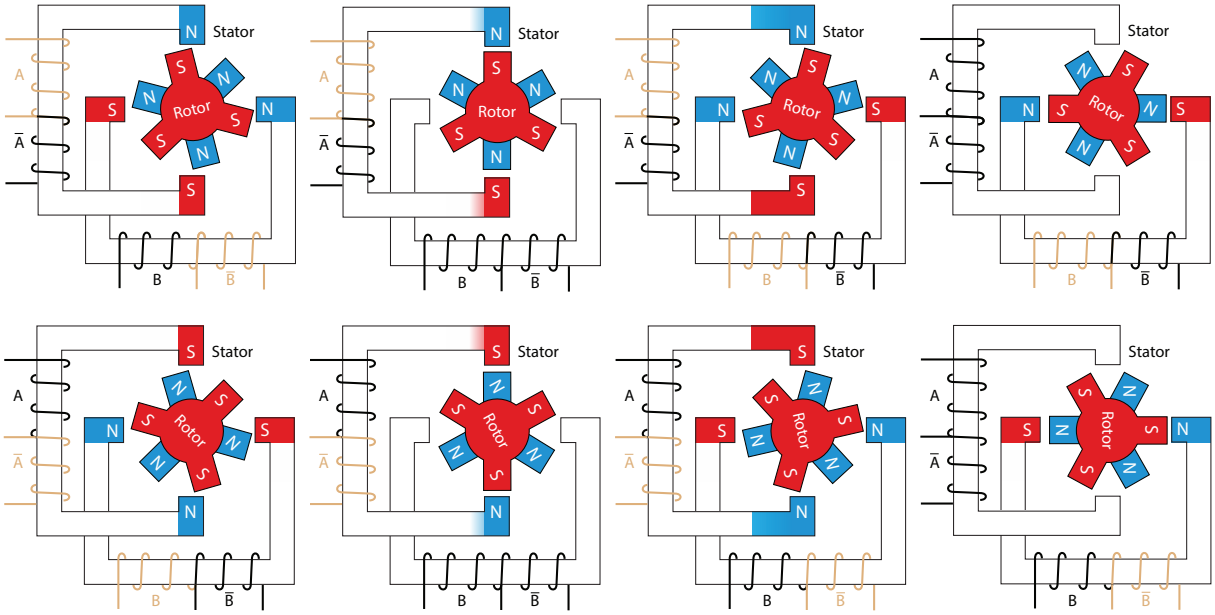
13.5.6. Sterowanie pełnokrokowe (dwufazowe)

W sterowaniu dwufazowym, czyli pełnokrokowym (*full stepping*), w każdej chili czasowej zasilane są dwie fazy. Wynikiem tego jest taki sam obrót wału, jak w sterowaniu falowym, z tym, że pozycja wału jest przesunięta o pół kroku. Dzięki jednoczesnemu sterowaniu obu faz, uzyskujemy większy moment siły na wale silnika. Sekwencja sterowań to (rysunek 13.13)

$$A\bar{B} \quad \rightarrow \quad AB \quad \rightarrow \quad \bar{A}B \quad \rightarrow \quad \bar{A}\bar{B}$$



Rys. 13.13: Sterowanie pełnokrokowe silnika unipolarnego



Rys. 13.14: Sterowanie półkrokowe silnika unipolarnego

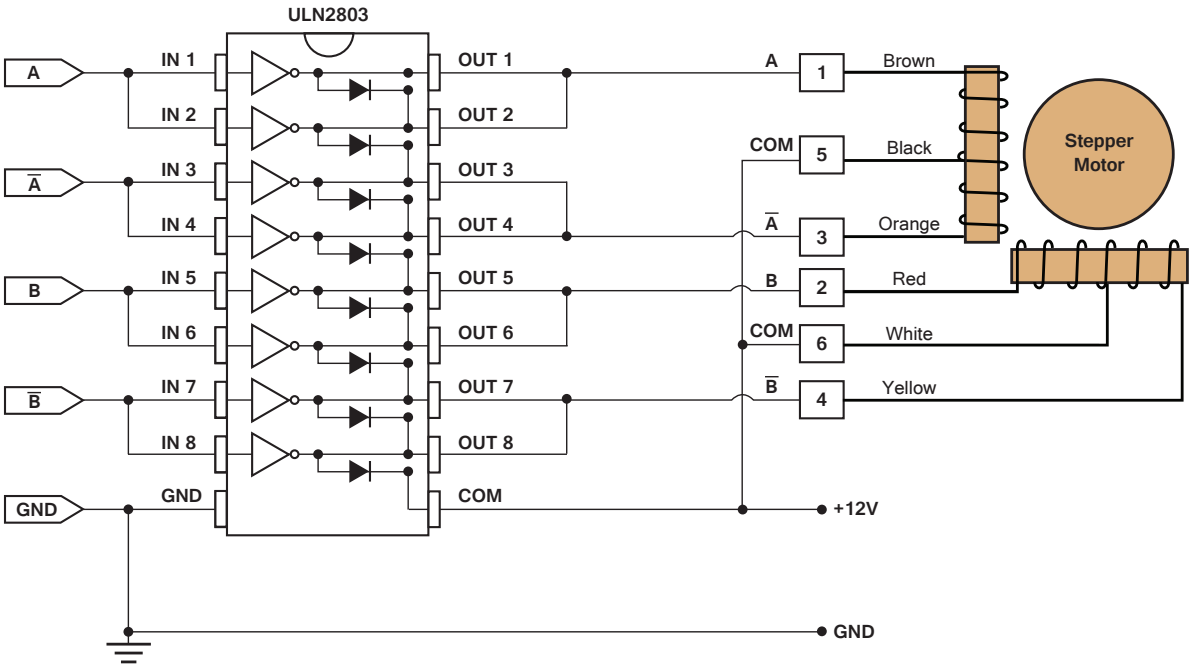
13.5.7. Sterowanie półkrokowe

Porównując oba poprzednie sterowania, widzimy, że pozycja wału jest przesunięta o pół kroku. Wykorzystamy ten fakt i połączymy obie sekwencje sterowań biorąc pierwsze sterowanie z pełnokrokowego, następnie pierwsze z falowego, drugie z pełnokrokowego, drugie z falowego, ... i tak dalej. Uzyskamy sekwencję ośmiu sterowań takich, że kolejne sterowania przesuwają wirnik o pół kroku. Będzie to sterowanie półkrokowe (*half stepping*). Charakterystyczną cechą tego sterowania jest naprzemienne zasilanie jednej fazy – dwóch faz – jednej – dwóch – ... Uzyskamy mniejsze "wypełnienie energią" silnika, czyli

mniejszy moment siły na wale, ale za to płynniejszy ruch, gdyż przesuwamy wał co pół kroku, a nie co krok, jak poprzednio. Poniżej pokazane jest złożenie obu sekwencji

$$\begin{array}{ccccccccccccccc}
A\bar{B} & & \longrightarrow & & AB & & \longrightarrow & & \bar{A}B & & \longrightarrow & & \bar{A}\bar{B} & & \\
 & & A & & \longrightarrow & & B & & \longrightarrow & & \bar{A} & & \longrightarrow & & \bar{B} \\
A\bar{B} & \rightarrow & A & \rightarrow & AB & \rightarrow & B & \rightarrow & \bar{A}B & \rightarrow & \bar{A} & \rightarrow & \bar{A}\bar{B} & \rightarrow & \bar{B}
\end{array}$$

Najbardziej płynny ruch można uzyskać sterując obie fazy dwiema przesuniętymi względem siebie sinusojdami: jest to *micro – stepping*. Rysunek 13.14 przedstawia kolejnych osiem kroków sterowania półkrokowego.



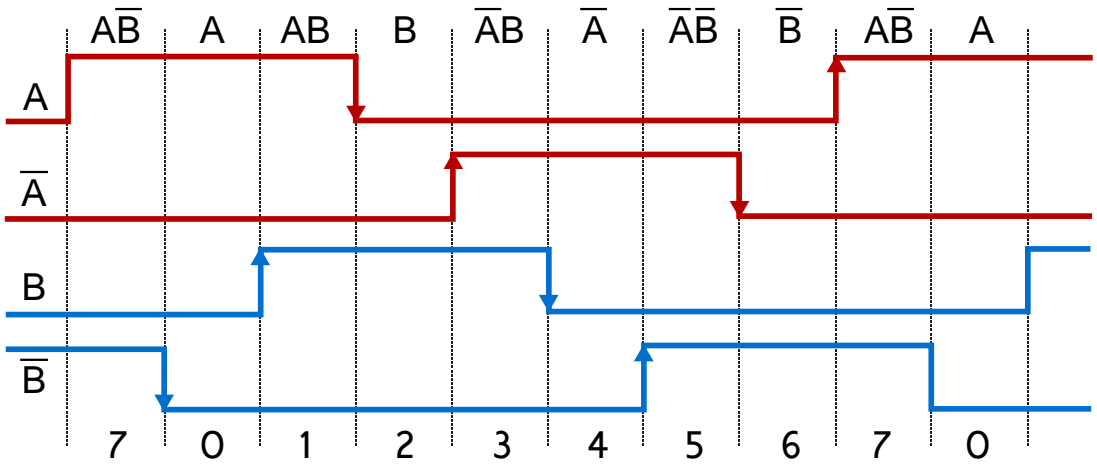
Rys. 13.15: Schemat połączenia silnika krokowego z płytą mikrokontrolera

13.6. Projekt15_StepperMotor

W projekcie wykorzystamy silnik krokowy hybrydowy ST4118M0406 produkcji firmy NANOTEC. Jest to silnik bipolarny, 2-fazowy, o sześciu wyprowadzeniach. Oporność połowy uzwojenia to 30Ω, czyli w konfiguracji bipolarnej, uzwojenie jednej fazy ma oporność 60Ω. Silnik dołączymy do płyty mikrokontrolera w konfiguracji unipolarnej i zasilimy napięciem 12V. Prąd sterowania jednej fazy będzie równy około 400 mA. Musimy pamiętać, że sterujemy indukcyjność napięciowo, przebiegiem prostokątnym, co będzie wywoływać zmiany prądu. Zgodnie ze wzorem 12.1 napięcie zaindukowane na indukcyjności w której nastąpiła zmiana prądu jest równe

$$U_L = -L \cdot \frac{di}{dt}$$

Napięcie zaindukowane będzie przeciwstawiało się zmianom prądu i utrudni dostarczenie odpowiednio dużej ilości energii do silnika (stała czasowa uzwojeń), dlatego nie uzyskamy wysokich obrotów. Sposoby przeciwdziałania temu zjawisku będą opisane w dalszej części



Rys. 13.16: Sterowanie półkrokowe silnika hybrydowego

rozdziału. Silnik krokowy dołączymy do procesora za pośrednictwem specjalizowanego układu scalonego ULN2803. Wyjścia tego układu wytrzymują obciążenie prądowe 500 mA i można je łączyć równolegle. Maksymalne napięcie sterujące to 50 V. Wejścia są kompatybilne z logiką CMOS 3.3V oraz 5V. Układ spełnia wymagania stawiane przez zastosowany silnik krokowy (400 mA i 12 V). Dla większego bezpieczeństwa połączymy równolegle po dwa kanały. Schemat połączeń pokazany jest na rysunku 13.15.

Zrealizujemy sterowanie półkrokowe wykorzystując przerwania generowane przez zegar systemowy `SysTick`.

Tabela 13.2: Połączenia pomiędzy płytą ZL27ARM a układem ULN2803

wejście ULN2803	**nazwa sygnału**	**port GPIO**	**LED**
IN1+IN2	A	PB8	LED1
IN3+IN4	$\bar{A}$	PB9	LED2
IN5+IN6	B	PB10	LED3
IN7+IN8	$\bar{B}$	PB11	LED4

Tabela 13.3: Połączenia pomiędzy płytą STM32F10E-EVAL a układem ULN2803

wejście ULN2803	**nazwa sygnału**	**port GPIO**	**LED**	**złącze CN10**
IN1+IN2	A	PF6	LD1 Green	pin 65
IN3+IN4	$\bar{A}$	PF7	LD2 Orange	pin 66
IN5+IN6	B	PF8	LD3 Red	pin 67
IN7+IN8	$\bar{B}$	PF9	LD4 Blue	pin 68

13.6.1. Płyta ZL27ARM

Kopiujemy projekt *Projekt04_SysTick* i zmieniamy jego nazwę. Mamy uruchomione przerwania od `SysTick`. Musimy wygenerować cztery przebiegi pokazane na rysunku 13.16. Zrobimy to w obsłudze przerwań. Kierunek obrotów można zamienić na przeciwny przez odwrócenie kolejności stanów z $0, 1, \ldots, 7$ (jak na rysunku 13.16) na $7, 6, \ldots, 0$.

Prędkość obrotową będziemy regulować zmieniając częstotliwość wywoływania przerwań od `SysTick`. Silnik NANOTEC ST4118M0406 ma 200 kroków na obrót. Stosujemy

sterowanie półkrokowe, czyli potrzebujemy 400 przerwań na jeden obrót. Zegar systemowy `SysTick` jest taktowany częstotliwością 72/8 = 9 MHz. Wpisanie do rejestru `STK_LOAD` wartości $9 \cdot 10^6$ oznacza, że przerwanie będzie wywoływane co jedną sekundę, czyli silnik wykona jeden obrót w ciągu 400 sekund. Analogicznie wpisanie `STK_LOAD` $= 9 \cdot 10^5$ to jeden obrót na 40 sekund, ... `STK_LOAD` $= 9 \cdot 10^3$ to jeden obrót na 0.4 sekundy. W naszym projekcie stosujemy uproszczone sterowanie napięciowe. Zgodnie z kartą katalogową sinika, można zasilać go prądem 400 mA/fazę. Oporność uzwojenia to 30 Ω, razem otrzymujemy napięcie zasilania 12V.

Układ ULN2803 z rysunku 13.15 wysterujemy sygnałami, które załączają diody świecące LED1–4 w płycie ZL27ARM. Dzięki temu będziemy mieli podgląd sterowań.
W pliku *main.c* zadeklarujemy dwie zmienne globalne:

```
unsigned int SMcounter = 0, SMclockwise = 1;
```

Pierwsza jest licznikiem stanów (jak na rysunku 13.16), druga określa kierunek obrotów. Generowanie sygnałów sterujących wykonamy w obsłudze przerwania od `SysTick` (w pliku *stm32f10x_it.c*).

```
// This function handles SysTick Handler.
extern unsigned int SMcounter, SMclockwise;

void SysTick_Handler(void){
   unsigned int i;
   SMcounter++;
   SMcounter &= 0x07;
   if(SMclockwise)    // kierunek obrotow zależy od 'SMclockwise'
      i = SMcounter;
   else
      i = 0x07 - SMcounter;
   switch(i){
      case 0: GPIO_ResetBits(GPIOB, GPIO_Pin_11);    // B̄ = 0
              break;
      case 1: GPIO_SetBits(GPIOB, GPIO_Pin_10);      // B = 1
              break;
      case 2: GPIO_ResetBits(GPIOB, GPIO_Pin_8);     // A = 0
              break;
      case 3: GPIO_SetBits(GPIOB, GPIO_Pin_9);       // Ā = 1
              break;
      case 4: GPIO_ResetBits(GPIOB, GPIO_Pin_10);    // B = 0
              break;
      case 5: GPIO_SetBits(GPIOB, GPIO_Pin_11);      // B̄ = 1
              break;
      case 6: GPIO_ResetBits(GPIOB, GPIO_Pin_9);     // Ā = 0
              break;
      case 7: GPIO_SetBits(GPIOB, GPIO_Pin_8);       // A = 1
   }
}
```

Teraz w pętli głównej możemy zmieniać prędkość obrotową wpisując do rejestru `STK_LOAD` liczby z zakresu od $9 \cdot 10^3$ do $9 \cdot 10^6$. Kierunek obrotów zależy od zmiennej `SMclockwise`, która przyjmuje wartości logoczne zero i jeden.

13.6.2. Płyta STM32F10E-EVAL

W katalogu *ARM2CortexM3-STM3210E-EVAL* kopiujemy ostatni projekt i zmieniamy jego nazwę. Inicjalizacja zegara systemowego i obsługa przerwań są identyczne. Teraz także układ ULN2803 dołączymy do sygnałów sterujących diodami świecącymi LED1–4. W obsłudze przerwania należy zmienić linie portów zgodnie z tabelą połączeń 13.3.

13.7. Indukcyjność

Indukcyjność określa zdolność obwodu do wytwarzania strumienia pola magnetycznego Φ powstającego w wyniku przepływu prądu i. Oznaczana jest symbolem L. Jednostką indukcyjności jest Henr (H). Ze strumieniem indukcji magnetycznej Φ i natężeniem prądu i związana jest wzorem:

$$\Phi = L \cdot i \tag{13.1}$$

Każda zmiana strumienia obejmowanego przez obwód (także strumienia wytworzonego przez ten obwód) wywołuje powstanie siły elektromotorycznej indukcji

$$\xi = -\frac{d\Phi}{dt} \tag{13.2}$$

Tę właściwość obwodów nazywa się samoindukcją. Zatem indukcyjność ma wpływ na wartość siły elektromotorycznej indukcji. Podstawiając wzór (13.1) do (13.2) otrzymujemy

$$\xi = -L\frac{di(t)}{dt} \tag{13.3}$$

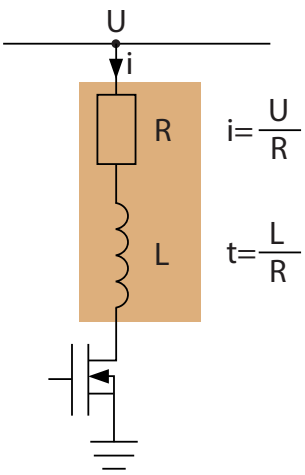
Siła elektromotoryczna indukcji jest więc proporcjonalna do prędkości zmian natężenia prądu elektrycznego. Współczynnikiem proporcjonalności jest indukcyjność wyrażona w Henrach H:

$$H = \frac{1\ V}{1\ A/s}$$

Cewka ma indukcyjność jednego Henra, gdy zmiana prądu o jeden Amper w ciągu sekundy, powoduje zaindukowanie napięcia jednego Volta.

13.7.1. Stała czasowa obwodu RL

Każda cewka ma pewną oporność uzwojenia oraz indukcyjność, które razem tworzą impedancję. Pamiętamy, że impedancja = rezystancja + j·reaktancja. Zgodnie z kartą katalogową silnika ST4118M0406 (rysunek C.3), przy połączeniu bipolarnym rezystancja uzwojenia jednej fazy jest równa 60Ω, natomiast indukcyjność to $100mH$. Uproszczony układ sterowania jednej fazy silnika, ze schematem zastępczym uzwojenia pokazany jest na rysunku 13.17.



Rys. 13.17: Uproszczone sterowanie jednej fazy, ze schematem zastępczym uzwojenia.

Przeanalizujemy teraz układ szeregowo połączonej indukcyjności i oporności, zasilanych ze źródła napięciowego. Indukcyjność (cewka) jest magazynem energii. Energia jest gromadzona w polu magnetycznym cewki. Ten magazyn jest ładowany prądem, który zgodnie ze wzorem 13.1 wytwarza strumień indukcji magnetycznej. Są dwa ograniczniki prądu: jeden to samoindukcja (wzór 13.3), a drugi to rezystancja uzwojeń. Pierwszy przeciwstawia się przepływowi prądu w stanach dynamicznych, drugi zawsze. Fazę ładowania magazynu energii jakim jest cewka możemy opisać równaniem różniczkowym (rysunek 13.18a)

$$U = R \cdot i(t) + L\frac{di(t)}{dt}, \qquad i(0) = 0 \tag{13.4}$$

z warunkiem początkowym $i(0) = 0$. Parametry U, L, R są stałe, natomiast prąd $i(t)$ jest funkcją czasu. Fazę rozładowania możemy opisać równaniem różniczkowym (rysunek 13.18b)

$$0 = R \cdot i(t) + L\frac{di(t)}{dt}, \qquad i(0) = \frac{U}{R} \tag{13.5}$$

z warunkiem początkowym $i(0) = U/R$, gdyż po pewnym czasie od włączenia, prąd płynący w obwodzie jest stały i równy $i = U/R$. Uzyskaliśmy (13.5) liniowe równanie różniczkowe jednorodne (*homogeneous*), które jest łatwiejsze do rozwiązania niż niejednorodne (13.4).

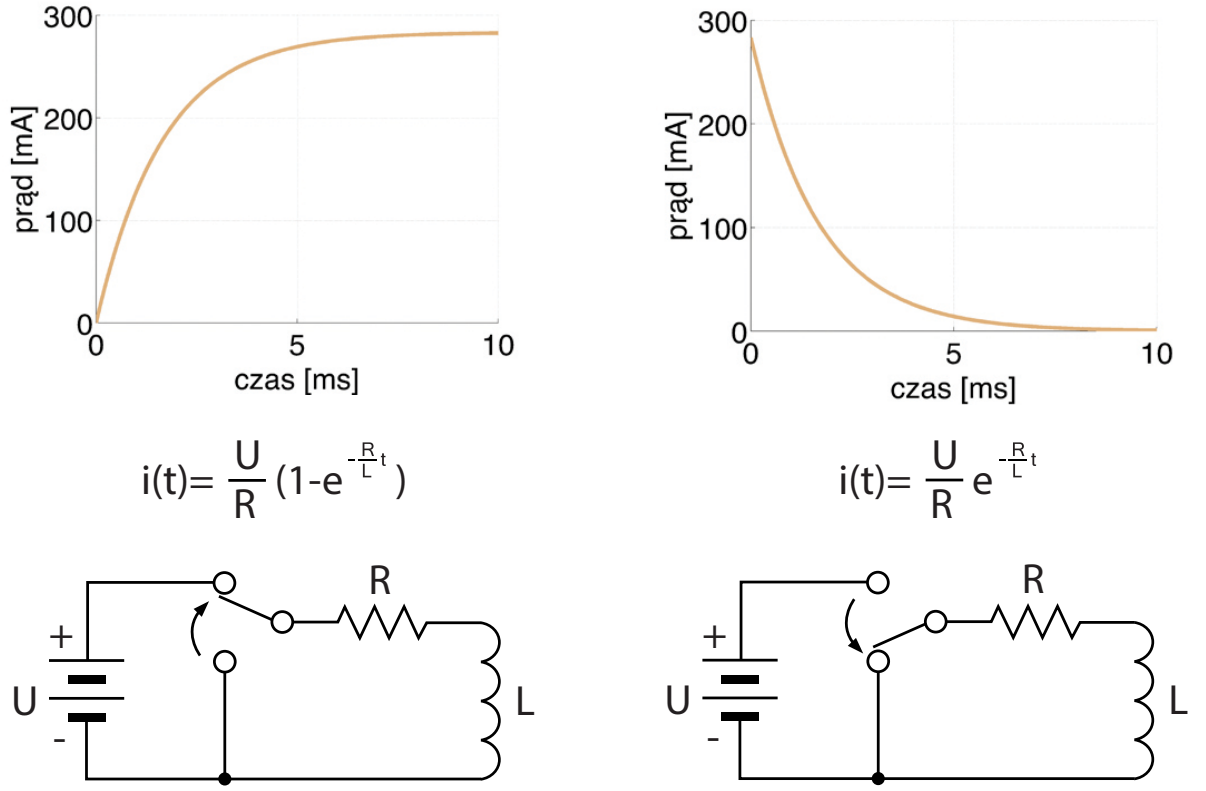
$$0 = R \cdot i + L\frac{di}{dt} \quad \rightarrow \quad \frac{di}{dt} = -\frac{R}{L}i \quad \rightarrow \quad \frac{di}{i} = -\frac{R}{L}dt \tag{13.6}$$

Całkujemy obustronnie (13.6)

$$\int \frac{di}{i} = -\frac{R}{L}\int dt \quad \rightarrow \quad ln|i| = -\frac{R}{L}t + C_1 \quad \rightarrow \quad i(t) = e^{-\frac{R}{L}t + C_1} = Ce^{-\frac{R}{L}t}$$

Do wyznaczenia stałej C wykorzystamy warunek początkowy

$$i(0) = Ce^{-\frac{R}{L}0} = \frac{U}{R} \quad \rightarrow \quad C = \frac{U}{R} \quad \rightarrow \quad i(t) = \frac{U}{R}e^{-\frac{R}{L}t} \tag{13.7}$$



(a) ładowanie magazynu energii

(b) rozładowanie

Rys. 13.18: Obwód indukcyjność – rezystancja

Uzyskaliśmy wzór (13.7) opisujący prąd rozładowania indukcyjności przez opornik. Wykres prądu rozładowania dla parametrów uzwojenia silnika ST4118M0406 (L = 100 mH, R = 60 Ω, U = 17V) pokazany jest na rysunku 13.18b.

Równanie dla fazy ładowania (13.4) rozwiążemy stosując podstawienie

$$u(t) = \frac{U}{R} - i(t) \quad \rightarrow \quad i(t) = \frac{U}{R} - u(t) \quad \rightarrow \quad \frac{di(t)}{dt} = -\frac{du(t)}{dt}$$
$$U = R \cdot i(t) + L\frac{di(t)}{dt} \quad \rightarrow \quad \frac{L}{R}\frac{di(t)}{dt} = \frac{U}{R} - i(t) \quad \rightarrow \quad -\frac{L}{R}\frac{du(t)}{dt} = u(t) \quad \rightarrow$$
$$\rightarrow \quad \frac{du}{u} = -\frac{R}{L}dt$$

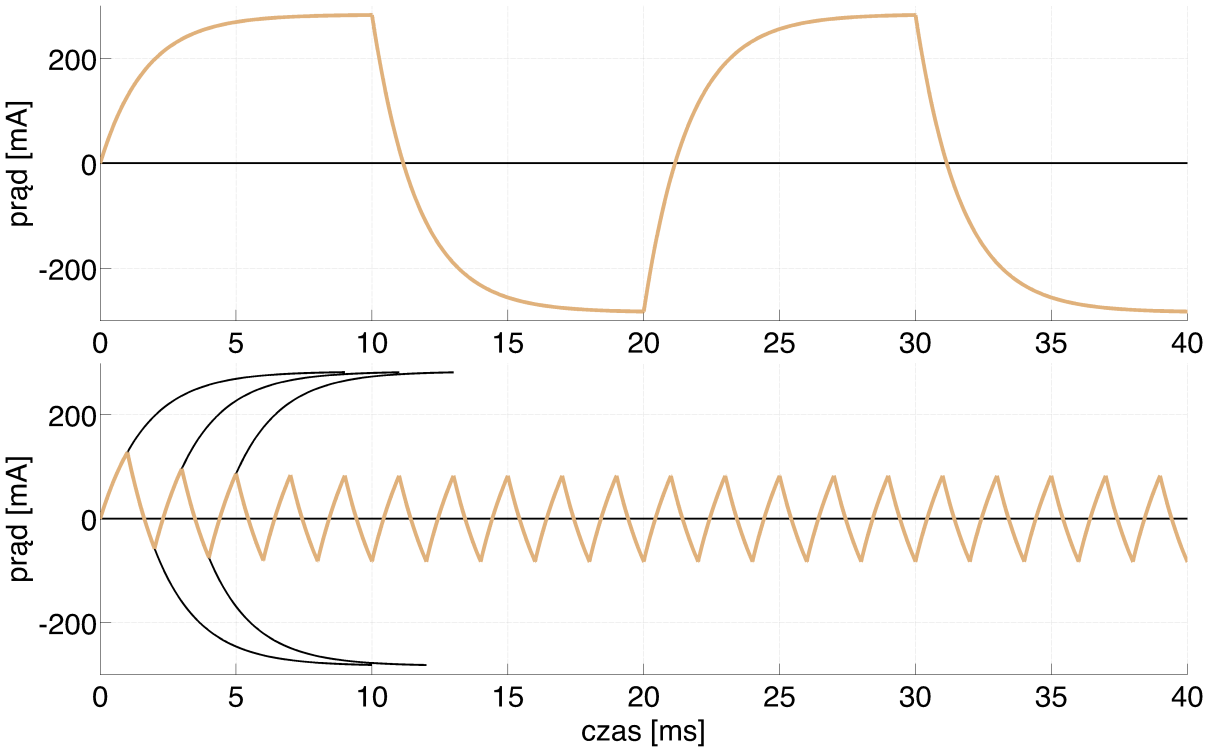
Korzystając z rozwiązania równania (13.6), otrzymujemy

$$u(t) = Ce^{-\frac{R}{L}t} \quad \rightarrow \quad \frac{U}{R} - i(t) = Ce^{-\frac{R}{L}t} \quad \rightarrow \quad i(t) = \frac{U}{R} - Ce^{-\frac{R}{L}t}$$

Do wyznaczenia stałej C wykorzystamy warunek początkowy

$$i(0) = \frac{U}{R} - Ce^{-\frac{R}{L}0} = 0 \quad \rightarrow \quad C = \frac{U}{R} \quad \rightarrow \quad i(t) = \frac{U}{R}\left(1 - e^{-\frac{R}{L}t}\right) \tag{13.8}$$

Uzyskaliśmy wzór (13.8) opisujący prąd ładowania indukcyjności. Wykres prądu ładowania dla parametrów uzwojenia silnika ST4118M0406 (L = 100 mH, R = 60 Ω, U = 17V) pokazany jest na rysunku 13.18a.



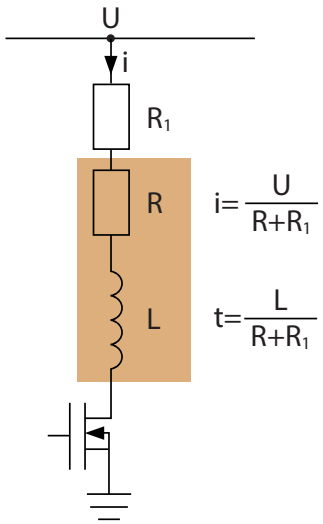
Rys. 13.19: Prąd w uzwojeniu silnika. Częstotliwość przełączania 100 Hz i 1000 Hz

Najprostsze sterowanie (napięciowe) polega na zasilaniu napięciem podanym w karcie katalogowej silnika. Na przykład silnik z uzwojeniem o oporności 5Ω i prądzie znamionowym $1A$ zasilimy napięciem $5V$.

Parametr L/R to stała czasowa układu LR, jednostki: $H = \Omega \cdot s \quad \rightarrow \quad s = H/\Omega$. Jeżeli silnik ST4118M0406 podłączymy w układzie bipolarnym, to stała czasowa jego uzwojeń będzie równa $L/R = 100mH/60\Omega = 1.667ms$. Stała czasowa uzwojeń silnika jest przeszkodą w uzyskaniu wysokich obrotów. W kolejnym projekcie będziemy sterować tym samym silnikiem, ale w układzie bipolarnym (rysunek 13.11a), czyli bez wykorzystania odczepów w środkach uzwojeń. W sterowaniu bipolarnym konieczne jest przełączanie kierunku prądu w uzwojeniu. Prąd w uzwojeniu silnika dla różnych częstotliwości sterowania (różnych prędkości obrotowych wału), pokazany jest na rysunku 13.19.

Górna część rysunku 13.19 pokazuje prąd w uzwojeniu silnika przy częstotliwości 100 Hz, czyli przełączenie kierunku następuje co 10 ms. Stała czasowa uzwojenia jest znacznie mniejsza (1.667 ms), dlatego prąd uzyskuje wartość $i = U/R$, jak w stanie ustalonym. Do silnika dostarczana jest znaczna energia, czyli uzyskujemy oczekiwany moment obrotowy. Dolna część rysunku jest wykresem prądu przy częstotliwości 1000 Hz. Czas pomiędzy przełączeniami (1 ms) jest mniejszy od stałej czasowej. Przełączenie następuje zanim prąd osiągnie wymaganą wartość – do silnika dostarczana jest zbyt mała energia. Czarne linie na tym rysunku to wykresy prądu w sytuacji gdy przełączenia następowałyby później.

Jak zwiększyć prąd w silniku, pomimo dużej częstotliwości przełączania? Analizując wzór (13.8) widzimy dwie metody. Możemy zwiększyć napięcie zasilania, ale wtedy należy ograniczyć wartość prądu tak, aby nie przekroczyć jego dopuszczalnej wartości. Druga metoda to zmniejszenie stałej czasowej uzwojenia $\tau = L/R$. Można to łatwo zrobić zwiększając rezystancję w mianowniku, na przykład przez dodanie zewnętrznego opornika. Zaczniemy od opornika.

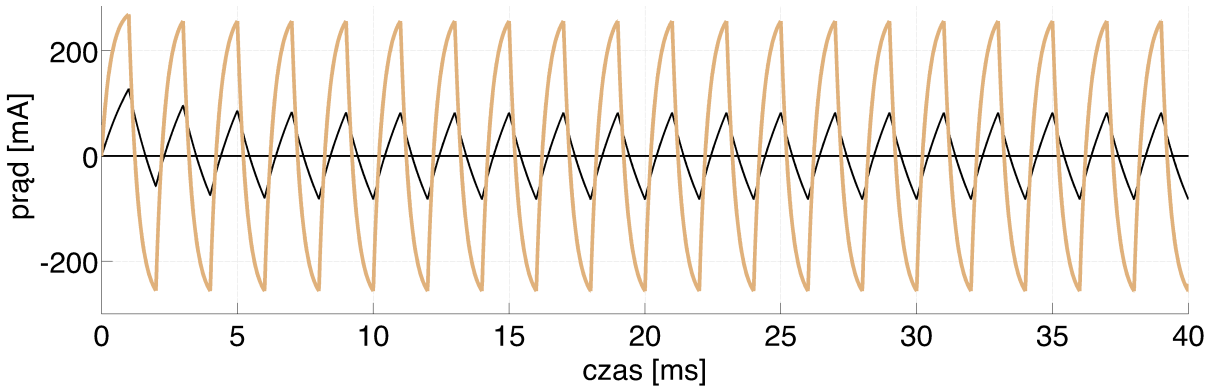


Rys. 13.20: Sterowanie L/nR

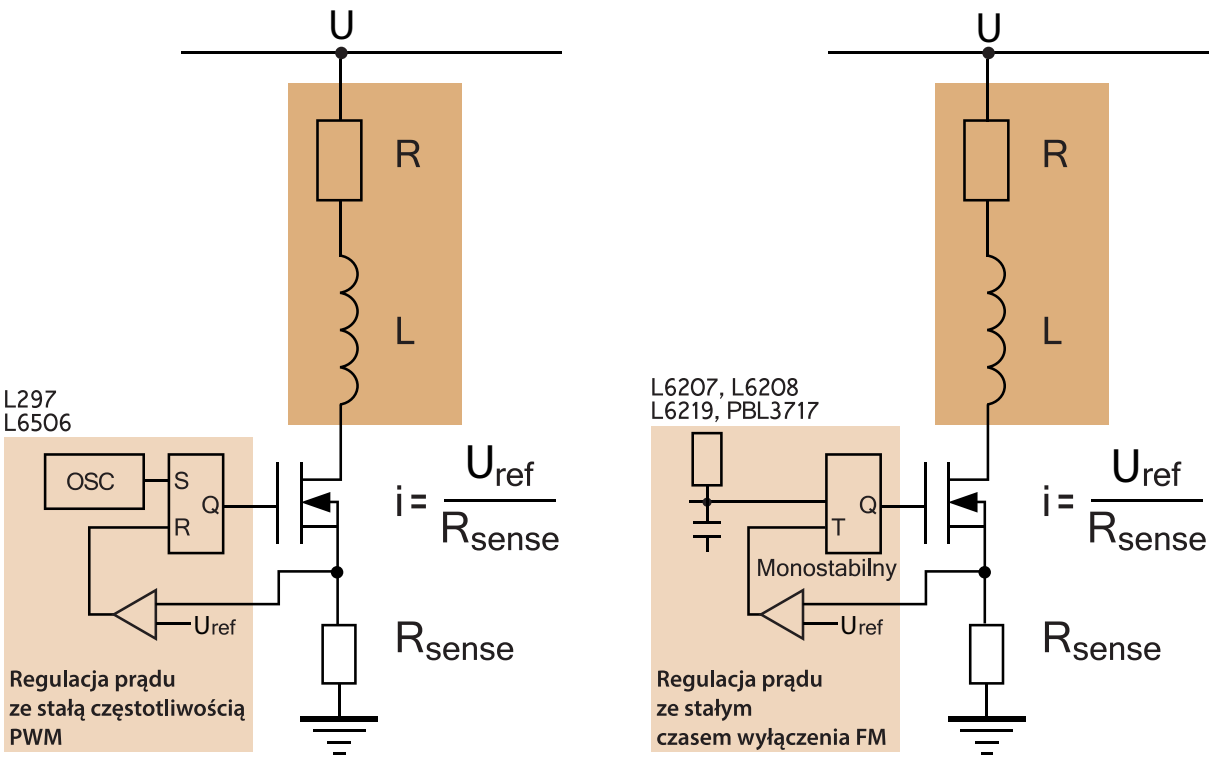
13.7.2. Sterowanie L/nR

Nie mamy wpływu na rezystancję i indukcyjność samego uzwojenia, ale możemy zmniejszyć jego stałą czasową dodając zewnętrzny opornik, jak na rysunku 13.20. Dodając opornik musimy jednocześnie zwiększyć napięcie zasilania tak, aby w stanie ustalonym przez uzwojenie płynął ten sam prąd co poprzednio. Wykonamy symulację. Dolny rysunek 13.19 przedstawia wykres prądu w uzwojeniu silnika przy parametrach: $L = 100mH, R = 60\Omega, U = 17V$, częstotliwość przełączania to $1000Hz$. Dodamy opornik $R_1 = 4 \cdot R = 240\Omega$, czyli pięciokrotnie zwiększymy całkowitą oporność. Musimy także pięciokrotnie zwiększyć napięcie zasilania ($U_5 = 85V$). W stanie ustalonym uzyskamy ten sam prąd $i = 283.333mA$. Rysunek 13.21 pokazuje wyraźną poprawę. Wzrosła amplituda prądu. Moment obrotowy jest proporcjonalny do prądu: widać wyraźny wzrost. Ideałem byłoby uzyskanie przebiegu prostokątnego, czyli takiego w którym prąd w uzwojeniu narasta nieskończenie szybko. Kolejna metoda sterowania, z kluczowaniem prądu zbliży nas do tego ideału.

Poważną wadą sterowania L/nR jest niska sprawność, czyli ilość energii uzyskanej do ilości dostarczonej. Zauważmy, że stosując opornik $4 \cdot R$ tylko jedna piąta (20%) dostarczonej energii trafia do silnika, pozostałe 80% jest tracone w opornikach. Z tego powodu obecnie nie stosuje się sterowania L/nR (można je spotkać w starych drukarkach).



Rys. 13.21: Porównanie sterowania L/5R ze sterowaniem L/R (linia czarna)



Rys. 13.22: Sterowanie z kluczowaniem prądu

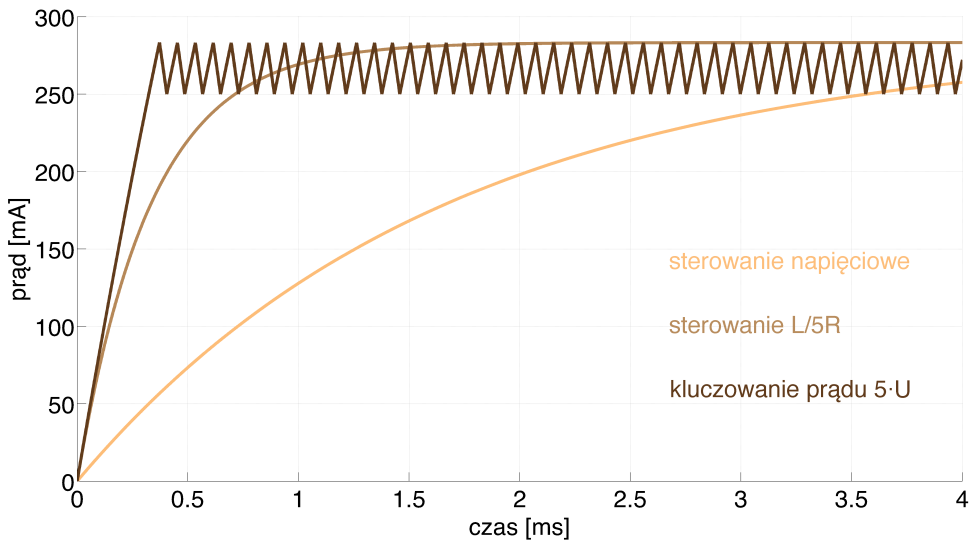
13.7.3. Sterowanie z kluczowaniem prądu

Zrezygnujemy z zewnętrznych oporników, ale pozostawimy zwiększone napięcie zasilania. Zgodnie ze wzorem (13.8) uzyskamy wzrost szybkości narastania prądu w uzwojeniu. Nachylenia zboczy, czyli szybkości narastania, porównane są na rysunku 13.23. Sterowanie napięciowe, to sterowanie napięciem wynikającym z prawa Ohma. Maksymalny prąd, to napięcie zasilania podzielone przez rezystancję uzwojenia. Widać wyraźny wpływ stałej czasowej $\tau = L/R$. Sterowanie L/5R spowodowało poprawę szybkości narastania prądu dzięki jednoczesnemu zmniejszeniu stałej czasowej i zwiększeniu napięcia zasilania. Maksymalny prąd jest ilorazem napięcia i sumy rezystancji uzwojenia i opornika zewnętrznego. Sterowanie bez opornika zewnętrznego, ale z podwyższeniem napięcia ma największą szybkość narastania, jednak nie ogranicza prądu w uzwojeniu. Dlatego należy dodać układ kluczowania prądu. Dwa obecnie stosowane rozwiązania pokazane są na rysunku 13.22.

W układach L297 i L6506 wewnętrzny oscylator załącza tranzystor (poprzez przerzutnik RS) ze stałą częstotliwością. Przepływ prądu powoduje powstanie napięcia na zewnętrznym oporniku pomiarowym R_{sense}. Najczęściej stosowane są oporniki dużej mocy, o wartości 0.5Ω. To napięcie porównywane jest z napięciem odniesienia U_{ref}. Wyłączenie tranzystora następuje, gdy $i = U_{ref}/R_{sense}$. W efekcie uzyskujemy sterowanie tranzystora stałą częstotliwością o zmiennym wypełnieniu (PWM). Układy scalone wymienione z prawej strony rysunku, wykorzystują przerzutnik monostabilny. Prąd jest kluczowany tranzystorem sterowanym sygnałem PWM o stałym czasie wyłączenia.

13.7.4. Sterowanie bipolarne silników krokowych

Moment obrotowy silnika krokowego jest proporcjonalny do natężenia pola magnetycznego, które jest proporcjonalne do prądu i oraz ilości zwojów N, czyli moment obrotowy T (*torque*) jest proporcjonalny do ilorazu $N \cdot i$. Naturalnym ograniczeniem na



Rys. 13.23: Porównanie prądów w uzwojeniu silnika dla trzech metod sterowania

wielkość prądu jest ilość energii rozpraszanej w uzwojeniu P_d (*power dissipaton*). Energia wydzielana w uzwojeniu jest iloczynem kwadratu prądu i oporności uzwojenia. Ponieważ rezystancja jest proporcjonalna do ilości uzwojeń, to energia jest proporcjonalna do $N \cdot I^2$. W przypadku silnika unipolarnego (indeks dolny u), mamy

$$P_d \sim N \cdot i_u^2 \qquad T_u \sim N \cdot i_u \tag{13.9}$$

Gdy ten sam silnik jest sterowany w sposób bipolarny (bez odczepu w środku - zasilane jest całe uzwojenie), to ilość zwojów podwaja się i uzyskujemy (indeks dolny b)

$$P_d \sim 2N \cdot i_b^2 \qquad T_b \sim 2N \cdot i_b \tag{13.10}$$

Jeżeli założymy, że moc wydzielana w silniku P_d dla obu sposobów sterowania silnika jest taka sama, to porównując wzory (13.9) i (13.10) otrzymujemy

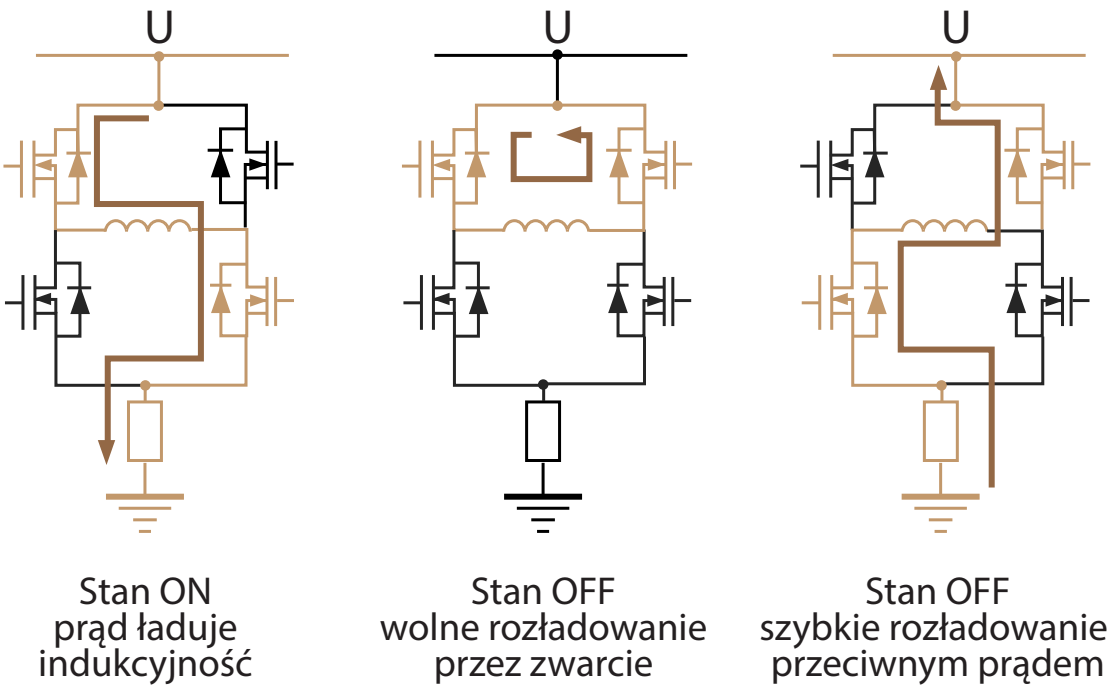
$$N \cdot i_u^2 = 2N \cdot i_b^2 \quad \rightarrow \quad i_b = \frac{i_u}{\sqrt{2}} \quad \rightarrow \quad T_b = \sqrt{2} \cdot T_u \approx 1.41 \cdot T_u$$

Ponieważ silnik sterowany bipolarnie wykorzystuje całe uzwojenia (a nie połowy jak w sterowaniu unipolarnym), to przy tej samej energii rozpraszanej wewnątrz silnika uzyskujemy moment obrotowy większy o około 40%.

Sekwencje sterowań są identyczne. Zasilane są całe uzwojenia. W przypadku silnika o dwóch fazach, sekwencje sterowań w kolejności: pełnokrokowe, falowe i ich złożenie, czyli sterowanie półkrokowe są następujące:

$$\begin{array}{ccccccccccccccc}
A\bar{B} & & \longrightarrow & & AB & & \longrightarrow & & \bar{A}B & & \longrightarrow & & \bar{A}\bar{B} & & \\
 & & A & & \longrightarrow & & B & & \longrightarrow & & \bar{A} & & \longrightarrow & & \bar{B} \\
A\bar{B} & \rightarrow & A & \rightarrow & AB & \rightarrow & B & \rightarrow & \bar{A}B & \rightarrow & \bar{A} & \rightarrow & \bar{A}\bar{B} & \rightarrow & \bar{B}
\end{array}$$

Inna jest interpretacja symboli: A oznacza zasilanie całego uzwojenia fazy A, natomiast $\bar{A}$ oznacza zasilanie tego samego uzwojenia, ale w przeciwnym kierunku. Analogicznie dla fazy B, symbole B i $\bar{B}$, to zasilanie całego uzwojenia prądami w przeciwnych kierunkach.



Rys. 13.24: Wolne i szybkie rozładowanie energii uzwojenia

13.7.5. Wolne i szybkie rozładowanie energii uzwojenia

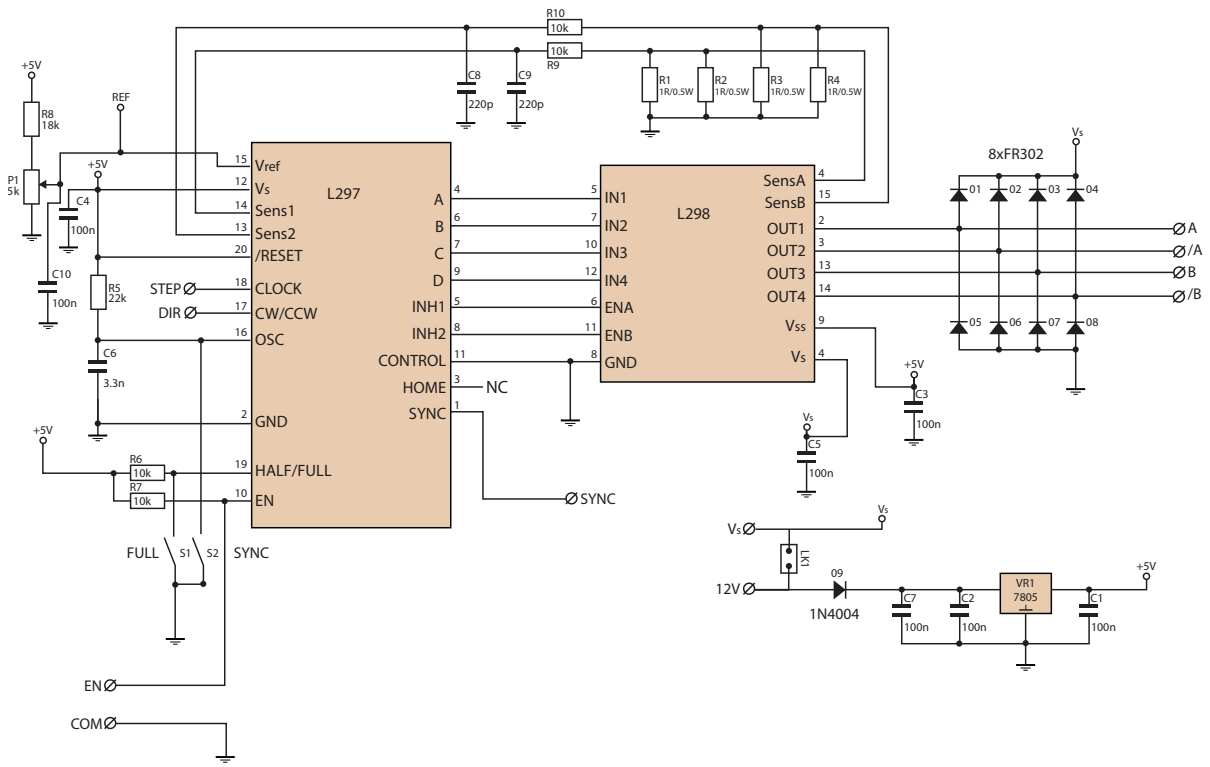
W uzwojeniu (cewce) energia zgromadzona jest w postaci pola magnetycznego.W trakcie sterowania półkrokowego, każde uzwojenie jest zasilane prądem (ON), następnie prąd jest wyłączany (OFF) i w kolejnym cyklu uzwojenie jest znów zasilane prądem (ON), ale w przeciwnym kierunku. Po każdym cyklu ON uzwojenie musi zostać rozmagnesowane, gdyż w następnym cyklu ON będzie namagnesowane w przeciwnym kierunku.

Jedna z metod rozmagnesowania, to załączenie (w cyklu OFF) obu górnych tranzystorów mostka H. Spowoduje to zwarcie końców uzwojenia i jego rozładowanie, zgodnie ze wzorem (13.7), ze stałą czasową $\tau = L/R$, czyli stosunkowo wolno. Ta metoda jest pokazana w środkowej części rysunku 13.24.

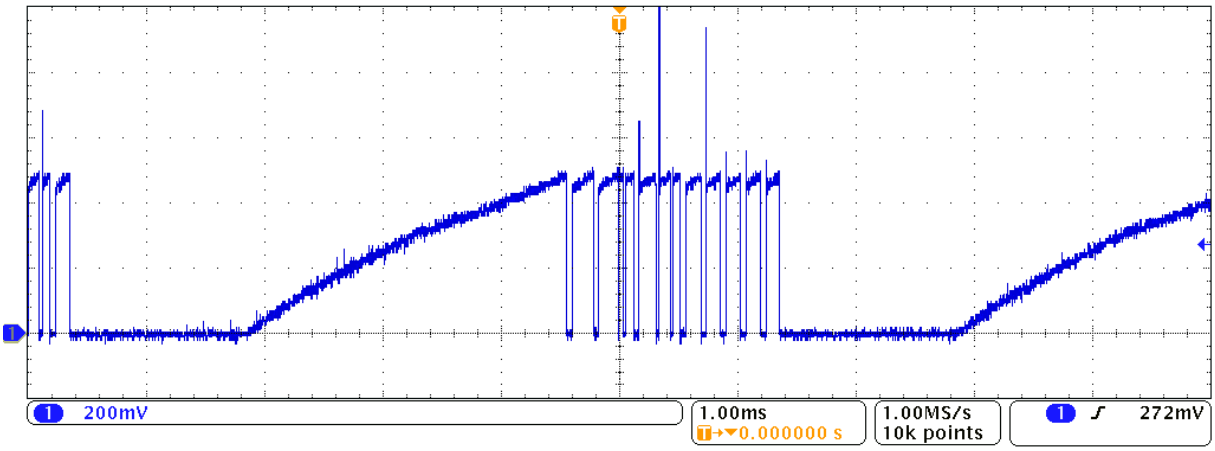
Wykorzystanie dużego napięcia zasilania (gdy stosujemy kluczowanie prądu) umożliwia szybkie rozładowanie uzwojenia. W cyklu OFF należy załączyć tranzystory na przeciwnej przekątnej mostka H. Duży, przeciwnie skierowany prąd spowoduje szybkie rozładowanie energii uzwojenia (prawa strona rysunku 13.24).

13.8. Projekt16_StepperMotor

Do sterowania prądowego silnika krokowego wykorzystamy gotową płytkę AS3187KT firmy QUASAR electronics. Dokumentację płytki możemy pobrać ze strony producenta www.QuasarElectronics.com. Schemat ideowy pokazany jest na rysunku 13.25. Układ scalony L297 to sterownik silników krokowych (*Stepper Motor Controller*). Umożliwia sterowanie pełnokrokowe oraz półkrokowe. Wyboru można dokonać za pomocą przełącznika na płytce AS3187KT. Do sterowania wystarczą dwa sygnały cyfrowe: `STEP` – każde zbocze opadające to jeden krok (półkrok), `DIR` – to kierunek obrotów.

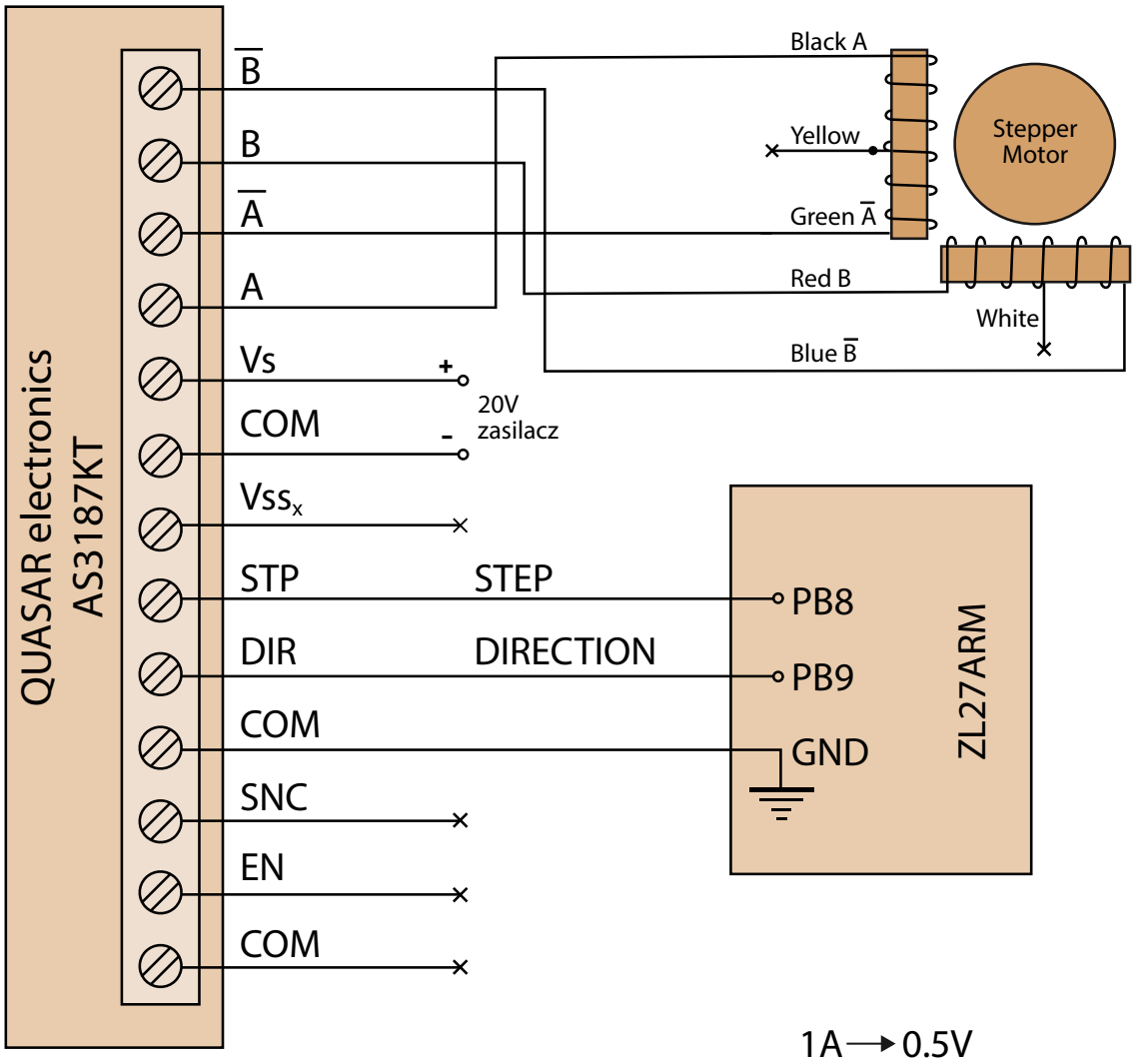


Rys. 13.25: Schemat ideowy płytki AS3187KT



Rys. 13.26: Oscylogram napięcia na oporniku układu kluczowania prądu

Układ scalony L298 to podwójny mostek (*Dual Full–Bridge Driver*). Karty katalogowe układów L297 i L298 można pobrać ze strony firmy STMicroelectronics www.st.com. Na płytce AS3187KT zainstalowane są cztery oporniki 1 Ω / 0.5 Wata, połączone równolegle po dwa. Są to czujniki prądu do układu kluczowania napięcia. W projekcie zastosowano silnik krokowy SY57STH41-1006A, którego karta katalogowa została zamieszczona w dodatku C, rysunek C.5. Zgodnie z tą kartą silnik będziemy sterować prądem 1 Amper, czyli na oporniku pomiarowym 0.5 Ω uzyskamy napięcie 0.5 Volta. Przed dołączeniem silnika, należy ustawić napięcie referencyjne komparatora układu kluczowania prądu. Służy do tego potencjometr P1 oraz punkt pomiarowy oznaczony jako `REF`.



Rys. 13.27: Schemat połaczeń układu sterowania prądowego silnikiem krokowym

Schemat połączeń układu sterowania pokazany jest na rysunku 13.27. Oscylogram napięcia na oporniku pomiarowym 0.5 Ω jest na rysunku 13.26. Po załączeniu mostka L298, prąd narasta aż do napięcia około 0.5 Volta (co odpowiada prądowi jednego Ampera) i wtedy następuje kluczowanie prądu. Szybkość narastania zależy od napięcia zasilania, zgodnie ze wzorem (13.8).

Porównanie trzech metod sterowania: napięciowe, napięciowe z opornikiem zmniejszającym stałą czasową uzwojenia oraz sterowanie prądowe prowadzi do oczywistego wniosku, że tylko sterowanie prądowe z kluczowaniem prądu pozwala dostarczyć do silnika wystarczająco dużo energii bez pogorszenia sprawności. Dzięki temu możemy uzyskać duże prędkości obrotowe, a sterowanie jest "mocne". Wadą są znacznie silniejsze rezonanse przy małych prędkościach obrotowych. Dlatego jeżeli naszym zadaniem jest wykonanie układu, który będzie bardzo wolno obracał reklamowany towar na wystawie sklepu – lepiej zastosować silnik o dużej oporności uzwojeń i sterowanie napięciowe. W przypadku silników krokowych ustalających położenie stołu frezarki musimy zastosować sterowanie prądowe – ze względu na wymagane duże momenty sił.

Tabela 13.4: Połączenia pomiędzy płytą ZL27ARM a płytą AS3187KT

zacisk AS3187KT	**nazwa sygnału**	**port GPIO**	**LED**
STP	step	PB8	LED1 Red
DIR	direction	PB9	LED2 Red
COM	masa GND		

Tabela 13.5: Połączenia pomiędzy płytą STM32F10E-EVAL a płytą AS3187KT

zacisk AS3187KT	**nazwa sygnału**	**port GPIO**	**LED**	**złącze CN10**
STP	step	PF6	LD1 Green	pin 65
DIR	direction	PF7	LD2 Orange	pin 66
COM	masa GND			pin 70

Rozdział 14

Liczby wymierne stałopozycyjne

W poprzednich rozdziałach omawialiśmy liczby całkowite ze znakiem i bez. Teraz zajmiemy się liczbami wymiernymi o ustalonym, stałym formacie. Zapis pozycyjny liczby przy podstawie b (*base*), ma postać:

$$\pm\, d_0.d_{-1}d_{-2}d_{-3}d_{-4}\cdots d_{-n+1}\cdot b^e \tag{14.1}$$

gdzie każdy symbol d_i to cyfra (*digit*), $0 \leqslant d_i < b$. Powszechnie używany system dziesiętny ma podstawę $b = 10$. Każda cyfra jest mnożona przez potęgę podstawy. Na przykład:

$$12.3456 = 1\cdot 10^1 + 2\cdot 10^0 + 3\cdot 10^{-1} + 4\cdot 10^{-2} + 5\cdot 10^{-3} + 6\cdot 10^{-4}$$

Liczby wymierne o podstawie $b = 2$ wykorzystują tylko dwie cyfry (0, 1) i są zapisywane z kropką binarną (zamiast dziesiętnej), na przykład

$$101.0101 = 1\cdot 2^2 + 0\cdot 2^1 + 1\cdot 2^0 + 0\cdot 2^{-1} + 1\cdot 2^{-2} + 0\cdot 2^{-3} + 1\cdot 2^{-4}$$

zapisując to dziesiętnie otrzymujemy

$$101.0101_2 = 4 + 1 + 0.25 + 0.0625 = 5.3125$$

Reprezentacja pozycyjna nie daje możliwości zapisania wszystkich liczb wymiernych za pomocą skończonej ilości cyfr. Na przykład liczba 1/3 ma nieskończoną reprezentację dziesiętną 0.3333(3). Podobnie jest w przypadku repezentacji binarnej. Liczba 1/10 to dziesiętnie 0.1, ale jej reprezentacja binarna jest nieskończona

$$1/10 = 0.1 = 0.0001100110011001100110011001100110011(0011)_2$$

Liczby będziemy zapisywać w standardowej postaci wykładniczej, to znaczy jako *mantysę* pomnożoną przez potęgę podstawy. W postaci standardowej, mantysa ma jedną cyfrę przed kropką dziesiętną /binarną. Wykładnik e, wzór (14.1), będziemy nazywać *cechą*.

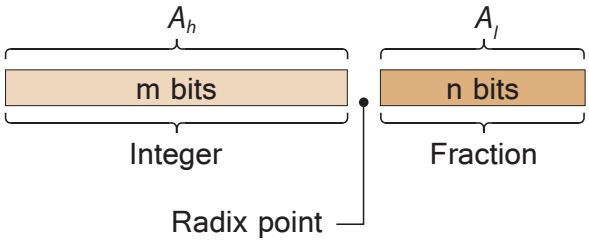
Jeżeli w danej reprezentacji, wykładnik e jest stały, to reprezentacja jest stałopozycyjna, gdy zmienny, to zmiennopozycyjna. Na przykład, dla stałych wartości $n = 6$ oraz $e = 3$

$$1.23456\cdot 10^3 = 1234.56 \tag{14.2}$$

Mamy łącznie sześć cyfr ($n = 6$), kropka dziesiętna jest na stałej pozycji pomiędzy czwartą i piątą cyfrą licząc od lewej. Reprezentacja stałopozycyjna w postaci ogólnej (dla ustalonych n oraz e)

$$\pm\, d_0\ .\ d_{-1}\ d_{-2}\ d_{-3}\ d_{-4}\cdots d_{-n+1}\cdot b^e = \pm d_e\cdots d_0\ .\ d_{-1}\cdots d_{-n+e+1}$$

Reprezentacja stałopozycyjna ma określoną rozdzielczość i zakres. Rozdzielczość liczby ze wzoru (14.2) to 0.01 , a zakres jest równy $[-9999.99,\ 9999.99]$.



Rys. 14.1: Format $UQm.n$ liczby stałopozycyjnej bez znaku

Arytmetyka stałopozycyjna jest realizowana za pomocą istniejącej w mikroprocesorze arytmetyki całkowitoliczbowej. Dzięki temu obliczenia są wykonywane łatwo i szybko. Arytmetyka zmiennopozycyjna może być zrealizowana na dwa sposoby: sprzętowo, gdy mikrokontroler jest wyposażony w koprocesor *floating-point*, albo programowo. Niestety, brak koprocesora powoduje konieczność realizacji programowej, co znacząco obniża wydajność obliczeniową systemu mikrokomputerowego stosowanego do sterowania.

Procesory Cortex-M7: STM32F745,7x6 są wyposażone w koprocesor *FPU Single Precision* (liczby 4-bajtowe `float`). Procesory Cortex-M7: STM32F765/7x7/7x9 mają nawet koprocesor *FPU Double Precision* (liczby 8-bajtowe `double`). Dlatego do realizacji zaawansowanych regulatorów predykcyjnych będziemy stosowali mikrokontrolery Cortex-M7 z koprocesorem FPU (SMS, część II).

Arytmetyka stałopozycyjna jest szybka i może być wystarczająco dokładna. Możliwe jest zbudowanie cyfrowego regulatora PID z wykorzystaniem mikrokontrolera bez koprocesora arytmetycznego, czyli na przykład Cortex-M3.

14.1. Binarna arytmetyka stałopozycyjna

W arytmetyce stałopozycyjnej jest przyjęta jedna, ustalona pozycja kropki binarnej. Ta pozycja jest wirtualna i istnieje tylko w głowie programisty: procesor wykonuje operacje na zwykłych liczbach całkowitych. Programista musi sam odpowiednio zaprogramować działania i zinterpretować wyniki.

14.1.1. Liczby stałopozycyjne bez znaku

Liczby stałopozycyjne bez znaku oznaczane są jako $UQm.n$, gdzie m to ilość bitów reprezentujących część całkowitą, a n to ilość bitów części ułamkowej, jak pokazano na rysunku 14.1. Wartość dziesiętną liczby można wyliczyć, korzystając z zależności

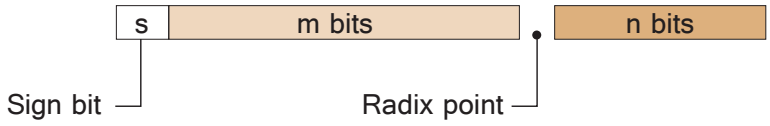
$$f = A_i + A_f \cdot 2^{-n}$$

gdzie A_i oraz A_f to całkowite wartości części całkowitej (*integer*) i ułamkowej (*fraction*) liczby. Na przykład, gdy $A_i = 101_2 = 5$ oraz $A_f = 1010_2 = 10$, to postać dziesiętna liczby $UQ3.4$ jest równa

$$101.1010_2 \qquad \longrightarrow \qquad f = A_i + A_f \cdot 2^{-4} = 5 + 10 \cdot 2^{-4} = 5\frac{10}{16} = 5.625$$

Oznaczmy przez I_A liczbę całkowitą o tej samej postaci binarnej (jaką widzi procesor). W powyższym przykładzie $I_A = 1011010_2 = 90$, mamy:

$$101.1010_2 \qquad \longrightarrow \qquad f = \frac{I_A}{2^n} = \frac{90}{16} = 5.625 \qquad \longrightarrow \qquad I_A = 2^n \cdot f \tag{14.3}$$



Rys. 14.2: Format $Qm.n$ liczby stałopozycyjnej ze znakiem

Wzór (14.3) jest szczególnie ważny, gdyż procesor będzie wykonywał operacje na liczbach całkowitych, a obowiązkiem programisty jest odpowiednia intepretacja wyników.

14.1.2. Liczby stałopozycyjne ze znakiem

Format liczb stałopozycyjnych ze znakiem (rysunek 14.2) oznaczany jest jako $Qm.n$. Całkowita ilość bitów reprezentacji, to $N = m + n + 1$, gdyż dodajemy bit znaku. Zastosujemy kodowanie U2 (uzupełnień do dwóch). Przykład: rozszyfrujemy liczbę Q3.4 o wartości 0xA6. Najstarszy bit jest jedynką, czyli jest to liczba ujemna. Negujemy bit po bicie i dodajemy jedynkę. Uzyskujemy -0x5A, czyli -5.625.

14.1.3. Konwersja liczby wymiernej do postaci stałopozycyjnej

Zamienimy liczbę $f = 3.14159265$ na postać $Q4.12$. W tym celu, zgodnie ze wzorem (14.3), pomnożymy f przez 2^n i część całkowitą wyniku zamienimy na postać binarną:

$$3.14159265 \cdot 2^{12} = 12\ 867.9634944 \qquad \longrightarrow \qquad 12\ 868 = 11\ 0010\ 0100\ 0100_2 = \texttt{0x3244}$$

Zamiana w drugą stronę: $\texttt{0x3244}[Q4.12] = 12\ 868/4\ 096 = 3.1416015625$, błąd reprezentacji w formacie Q4.12 jest w tym przypadku równy

$$\epsilon = 3.14159265 - 3.1416015625 = -8.9125 \cdot 10^{-6}$$

Tą samą liczbę, ale ze znakiem minus, otrzymamy gdy zamienimy znak liczby `0x3244` na przeciwny: $-3.14159265[Q4.12] = \texttt{0xCDBC}$

14.1.4. Kompromis pomiędzy rozdzielczością a zakresem

Liczba stałopozycyjna $UQm.n$ zajmuje $m+n$ bitów, liczba ze znakiem $Qm.n$ zajmuje $n + m + 1$ bitów. Jeżeli na reprezentację liczby możemy przeznaczyć łącznie N bitów, musimy znaleźć kompromis pomiędzy rozdzielczością a zakresem. Kompromis ten jest uzależniony od wymagań aplikacji.

Rozdzielczość (*resolution*) jest definiowana jako najmniejsza liczba różna od zera, która ma reprezentację w formacie $Qm.n$. Jest to jednocześnie odległość pomiędzy dwiema sąsiednimi liczbami. Dla liczb $UQm.n$ oraz $Qm.n$ rozdzielczość jest równa 2^{-n}. Na przykład, rozdzielczość liczby $Q3.3$ to 2^{-3}, czyli 0.125. Musimy odróżnić rozdzielczość od dokładności:

- Dokładność (*accuracy*) jest to stopień zgodności rzeczywistej (prawdziwej) wartości liczby z jej reprezentacją liczbową. Na przykład, jeżeli założymy, że prawdziwą wartością liczby Π jest 3.14159265, to dokładność reprezentacji $Q4.12$ jest równa $\epsilon = 8.9125 \cdot 10^{-6}$. Rozdzielczość tej samej reprezentacji, to $2^{-12} = 0.00244$
- Rozdzielczość (*resolution*) opisuje precyzję reprezentacji liczby. Jest to najmniejsza zmiana liczby, którą można zapisać w danym formacie.

Zakres (*range*) jest to przedział liczb, które mogą być zapisane w danym formacie. Przedziałem jest różnica pomiędzy najmniejszą i największą liczbą, jaka może być zapisana w danym formacie.

- **Zakres liczb stałopozycyjnych bez znaku**. Zakres liczb całkowitych (*integer*) bez znaku, zapisanych na $m + n$ bitach to $[0,\ 2^{m+n} - 1]$. Zgodnie ze wzorem (14.3) zakres liczb stałopozycyjnych zapisanych w formacie $UQm.n$ jest równy $[0,\ 2^{m+n} - 1] \cdot 2^{-n}$, czyli ostatecznie: $[0,\ 2^m - 2^{-n}]$. Na przykład dla formatu $UQ4.12$ (16 bitów) jest to $[0, 2^4 - 2^{-12}] = [0,\ 15.9998]$
- **Zakres liczb stałopozycyjnych ze znakiem**. Zakres liczb całkowitych ze znakiem, zapisanych na $m+n+1$ bitach (w kodzie uzupełnień do dwóch), to $[-2^{m+n},\ 2^{m+n}-1]$. Zgodnie ze wzorem (14.3) zakres liczb stałopozycyjnych zapisanych w formacie $Qm.n$ jest równy $[-2^{m+n},\ 2^{m+n}-1] \cdot 2^{-n}$, czyli ostatecznie: $[-2^m,\ 2^m - 2^{-n}]$. Na przykład dla formatu $Q3.12$ (16 bitów ze znakiem) jest to $[-2^3,\ 2^3 - 2^{-12}] = [-8,\ 7.9998]$

Reprezentacja stałopozycyjna oznacza kompromis pomiędzy rozdzielczością a zakresem. Gdy długość reprezentacji (ilość bitów do zapisania liczby) jest stała i równa na przykład $N = 32$, to format $Q20.11$ zapewnia duży zakres i małą rozdzielczość, a format $Q11.20$ odwrotnie: mały zakres i dużą rozdzielczość. Format musi być dostosowany do wymagań aplikacji.

14.1.5. Arytmetyka stałopozycyjna z nasyceniem

Najważniejszym celem pisania arytmetyki stałopozycyjnej jest stworzenie szybkiego i dokładnego regulatora PID na bazie ARM Cortex-M3. Regulator musi działać niezawodnie w każdej sytuacji, czyli przekroczenie zakresu liczb nie może spowodować błędnej regulacji. Dlatego napiszemy arytmetykę z nasyceniem: jeżeli wynik (na przykład mnożenia) przekracza zakres liczb, to wynikiem staje się zakres reprezentacji. W przypadku arytmetyki 32-bitowej ze znakiem, maksymalana liczba dodatnia, to `0x7FFFFFFF`, minimalna ujemna: `0x80000000`. Wartości liczbowe przypisane do tych reprezentacji zależą od formatu, na przykład dla $Q15.16$ mamy:

`0x7FFFFFFF`	→	32'767.998047
`0x80000000`	→	-32'768.0

14.2. Działania niezależne od formatu

Funkcje, które realizują: negację liczby, dodawanie i odejmowanie nie zależą od fomatu - są uniwersalne dla wszystkich liczb określonej długości.

14.2.1. Negacja i moduł liczby

Zaczniemy od funkcji realizującej zamianę znaku liczby na przeciwny. Połączymy ją z funkcją obliczającą moduł, czyli wartość bezwzględną. W kodzie uzupełnień do dwóch (patrz rozdział 2.6.2) zmiana znaku polega na zanegowaniu wszystkich bitów i dodaniu jedynki. Jeżeli w wyniku tego dodawania zapalona zostanie flaga V (*Overflow*), to oznacza, że wynik nie mieści w założonej ilości bitów. W takim przypadku, wynikiem operacji jest zakres, czyli `0x7FFFFFFF` gdy liczba jest dodatnia, albo `0x80000000` gdy ujemna.

```
;------------------- Q32 wartość bezwzględna -------------------
          EXPORT Q32_abs
          ALIGN
Q32_abs   PROC
          MOVS    r1,#1        ; maska:
          ROR     r1,#1        ; r1 = 0x80000000
          TST     r0,r1        ; bit r0[31]
          BNE     Q32_neg      ; skocz, liczba ujemna
          BX      lr
          ENDP
;------------------------- Q32 negacja -------------------------
          EXPORT Q32_neg
          ALIGN
Q32_neg   PROC
          MOVS    r2,r0        ; pamiętaj znak oryginału
          MVNS    r0,r0        ; negacja bit po bicie
          ADDS    r0,#1        ; r0 = r0 + 1, Flagi
          BVC     label01      ; skocz, gdy overflow clear
          MOVS    r0,#1
          ROR     r0,#1        ; w r0 mam 0x80000000
          TST     r2,r0
          BEQ     label01      ; pamiętany znak jest odwrotny
          MVNS    r0,r0        ; negacja bit po bicie
label01   BX      lr
          ENDP
```

14.2.2. Dodawanie i odejmowanie

Niech f_A, f_B i f_C to będą trzy liczby stałopozycyjne bez znaku, w formacie $UQn.m$ albo $Qn.m$. Chcemy obliczyć sumę i różnicę:

$$f_C = f_A + f_B \qquad\qquad f_C = f_A - f_B \tag{14.4}$$

Zgodnie ze wzorem (14.3) $f_X = I_X \cdot 2^{-n}$, gdzie I_X jest liczbą całkowitą o takiej samej postaci bitowej jak f_X, czyli jest to f_X, ale interpretowana jako liczba całkowita. Działania (14.4) możemy zapisać w postaci:

$$\begin{aligned}
f_C = f_A + f_B \quad &\rightarrow \quad I_C \cdot 2^{-n} = I_A \cdot 2^{-n} + I_B \cdot 2^{-n} = (I_A + I_B) \cdot 2^{-n} \quad \rightarrow \quad I_C = I_A + I_B \\
f_C = f_A - f_B \quad &\rightarrow \quad I_C \cdot 2^{-n} = I_A \cdot 2^{-n} - I_B \cdot 2^{-n} = (I_A - I_B) \cdot 2^{-n} \quad \rightarrow \quad I_C = I_A - I_B
\end{aligned}$$

Jak widać, w przypadku dodawania i odejmowania, wykonujemy zwykłe działania na liczbach całkowitych. Wynik interpretujemy zgodnie z założonym formatem. Reguła ta dotyczy liczb ze znakiem i bez, dla wszystkich wartości n oraz m.

Jeżeli wykonujemy działania na liczbach ze znakiem, to zapalenie flagi V (*Overflow*) oznacza, że wynik nie mieści w założonej ilości bitów, patrz rozdział 2.8. W takim przypadku, wynikiem operacji jest zakres, czyli `0x7FFFFFFF` gdy liczba jest dodatnia, albo `0x80000000` gdy ujemna. Analiza działań opisanych w rozdziale 2.8 prowadzi do wniosku, że zawsze gdy występuje przepełnienie, to bit znaku jest nieprawidłowy. Jeżeli wynik powinien być dodatni, to w efekcie przepełnienia jego bit znaku jest jedynką. I odwrotnie: gdy wynik powinien być ujemny, bit znaku jest zerem. Dzięki temu możemy ustalić, który zakres należy wybrać: zawsze odwrotnie do bitu znaku wyniku.

```
;---------------- Q32 odejmowanie z nasyceniem ----------------
          EXPORT Q32_sub
          ALIGN
Q32_sub   PROC
          SUBS    r0,r1       ; r0 = r0 - r1, Flagi
          BVS     label02     ; skocz, gdy overflow set
          BX      lr
          ENDP
;----------------- Q32 dodawanie z nasyceniem -----------------
          EXPORT Q32_add
          ALIGN
Q32_add   PROC
          ADDS    r0,r1       ; r0 = r0 + r1, Flagi
          BVC     label03     ; skocz, gdy overflow clear
label02   MOVS    r1,r0
          MOVS    r0,#1
          ROR     r0,#1       ; w r0 mam 0x80000000
          TST     r1,r0
          BEQ     label03     ; pamiętany znak jest odwrotny
          MVNS    r0,r0       ; negacja bit po bicie
label03   BX      lr
          ENDP
```

14.3. Działania zależne od formatu (Q15.16)

Kolejne działania, czyli mnożenie i dzielenie zależą od ustalonego formatu liczby, czyli położenia kropki binarnej. Przyjmiemy format $Q15.16$. Mamy: jeden bit znaku, 15 bitów przed i 16 za kropką binarną, razem 32 bity. Liczba mieści się w rejestrze, kropka binarna jest w połowie, czyli pomiędzy słowami 16-bitowymi – dzięki temu łatwiej będzie formatować wyniki mnożenia i dzielenia. Ponadto uzyskujemy całkiem dobrą rozdzielczość: $2^{-16} = 1.5259 \cdot 10^{-05}$. Pierwszym programem będzie konwersja do liczby zmiennoprzecinkowej pojedynczej precyzji (4-bajtowy `float`). Umożliwi nam to wyświetlanie wyników działań za pośrednictwem biblioteki standardowej języka C. Dokładny opis reprezentacji zmiennoprzecinkowej jest zawarty w normie IEEE 754, my ograniczymy się do podstaw. Zapis binarny liczby:

$$\pm 1 \,.\, d_{-1}\, d_{-2}\, d_{-3}\, d_{-4} \cdots d_{-23} \cdot 2^{e}$$

gdzie d_i to cyfra binarna (`0` albo `1`). Przed kropką binarną jest zawsze jedynka – dlatego nie trzeba jej pamiętać (zero ma wyróżnioną reprezentację: wszystkie bity są zerami). Jedynka jest więc odrzucana i zapisywane są 23 bity części ułamkowej (*fraction*). Wykładnik jest liczbą 8-bitową ze znakiem. Jednak wygodniej jest realizować działania na liczbach, gdy wykładnik (*exponent*) pamiętany jest z przesunięciem +127. Liczba zmiennopozycyjna, wyrażona wzorem

$$\pm \,.\, d_{-1}\, d_{-2}\, d_{-3}\, d_{-4} \cdots d_{-23} \cdot 2^{e+127}$$

(z domyślną jedynką przed kropką), pamiętana jest w formacie pokazanym na rysunku 14.3.

Rys. 14.3: Format liczby zmiennopozycyjnej pojedynczej precyzji

```
;----------- zamiana reprezentacji Q15.16 na float ------------
;INPUT:   r0 = liczba stałopozycyjna ze znakiem Q15.16
;OUTPUT:  r0 = liczba zmiennopozycyjna float (4 bajty)
            EXPORT Q1516_float
            ALIGN
Q1516_float PROC
            MOVS    r1,r0          ; ustaw flagi
            BEQ     label07        ; gdy zero, to koniec
            LSLS    r0,r0,#1       ; bit znaku do Carry
            MOVS    r0,#0          ; wynik będzie w r0
            BCC     label04        ; skocz, gdy C = 0
            MVNS    r1,r1          ; negacja bit po bicie
            ADD     r1,#1          ; w r1 jest liczba nieujemna
            ORR     r0,r0,#(1<<31); bit znaku jest jedynką
label04     MOVS    r2,#32         ; licznik -> wykładnik
label05     LSLS    r1,r1,#1       ; bit znaku do Carry
            BCS     label06        ; skocz, gdy C = 1 (mamy jedynkę)
            SUBS    r2,#1          ; licznik --
            BNE     label05        ; jeszcze nie koniec
label06     UBFX    r3,r1,#9,#23   ; mantysa (bez jedynki) do r3
            ORR     r0,r0,r3
            ADD     r2,#110         ; przesunięcie wykładnika
            LSL     r2,r2,#23
            ORR     r0,r0,r2
label07     BX      lr
            ENDP
```

14.3.1. Mnożenie stałopozycyjne

Przyjmując te same oznaczenia:

$$f_C = f_A \cdot f_B = I_A 2^{-n} \cdot I_B 2^{-n} = I_A \cdot I_B 2^{-2n} = I_C 2^{-n} \quad \rightarrow \quad I_C = I_A \cdot I_B 2^{-n} \tag{14.5}$$

Przykład mnożenia liczb w formacie $UQ16.16$ lub $Q15.16$, pokazany jest na rysunku 14.4. Wynikiem mnożenia dwóch liczb całkowitych 32-bitowych jest liczba całkowita 64-bitowa. Należy odciąć po 16 bitów z obu stron. Wynik będzie liczbą 32-bitową, zgodnie z wybranym formatem. Odcięcie najstarszych szesnastu bitów jest równoważne pomnożeniu wyniku przez 2^{-16}, co jest zgodne ze wzorem (14.5). Najmłodsze odcięte bity są cyframi, które nie mieszczą się w założonej długości (32) reprezentacji liczby.

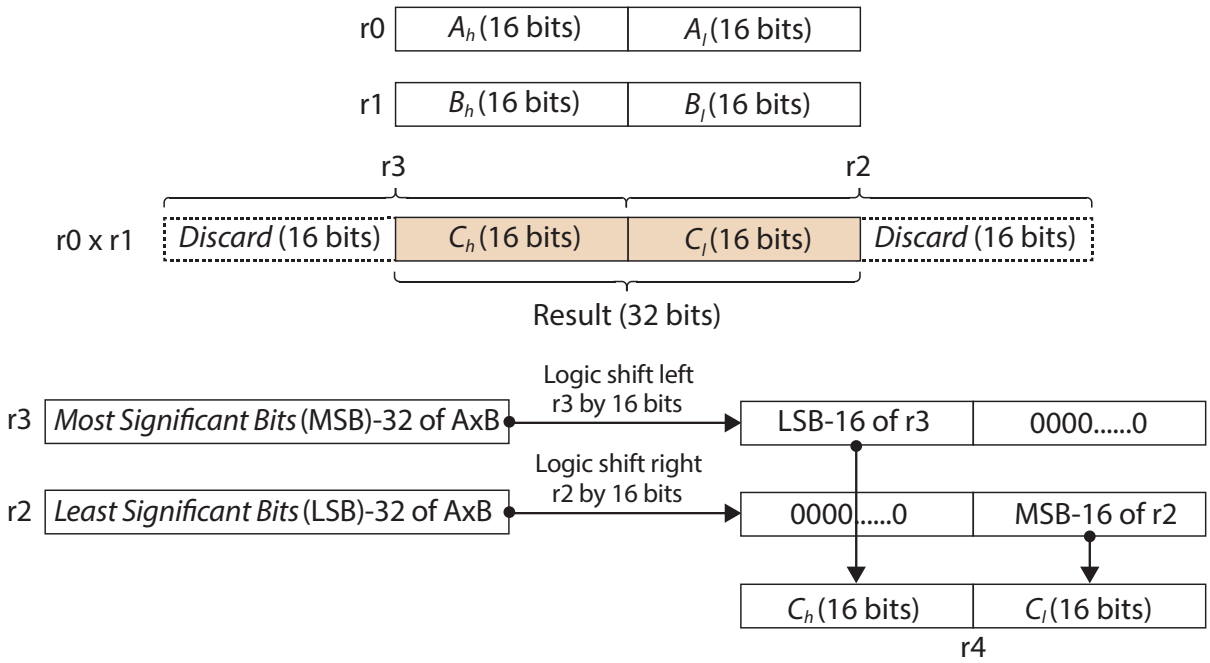
Wynik wykazuje przepełnienie, gdy znak wyniku nie jest zgodny z oczekiwanym. Należy także sprawdzić zawartość najstarszych szesnastu bitów, 64-bitowego wyniku. Te bity są odrzucane w trakcie formatowania wyniku. Jeżeli wynik powinien być liczbą dodatnią

(lub zerem), to najstarsze odrzucane bity muszą być zerami (wszystkie). Jeżeli wynik powinien być ujemny, to wszystkie (najstarsze) odrzucane bity muszą być jedynkami.

```
;---------------- Q15.16 mnożenie z nasyceniem ----------------
;INPUT:  r0 = liczba stałopozycyjna ze znakiem Q15.16
;        r1 = liczba stałopozycyjna ze znakiem Q15.16
;OUTPUT: r0 = r0 * r1
           EXPORT Q1516_mult
           ALIGN
Q1516_mult PROC
           PUSH    {r4-r8,lr}  ; parzysta ilość rejestrów
           ; maski:
           MOVS    r4,#0
           MVNS    r5,r4
           LSL     r5,#16      ; w r5 mam 0xFFFF0000
           MVNS    r6,r4
           LSR     r6,#1       ; w r6 mam 0x7FFFFFFF
           MVNS    r4,r6       ; w r4 mam 0x80000000
           ; signed long multiply:
           SMULL   r2,r3,r0,r1 ; (r3:r2) = r0 * r1
           ; czy wynik jest zerem
           ORRS    r8,r2,r3
           BNE     label15     ; skocz, gdy nie
           MOVS    r0,#0
           POP     {r4-r8,pc}
label15    EOR     r7,r0,r1    ; r7[b31] to znak wyniku
           MOVS    r1,r3       ; do sprawdzenia OVER
           LSLS    r3,r3,#16   ; shift left high word
           LSRS    r2,r2,#16   ; shift right low word
           ORR     r0,r2,r3    ; w r0 jest wynik
           ; sprawdzenie OVER
           LSLS    r7,r7,#1    ; bit znaku do Carry
           BCC     label16     ; skocz, gdy C = 0
           ANDS    r8,r0,r4    ;TU: wynik musi być ujemny
           BEQ     label18     ; skocz, gdy plus = nasycenie
           MVNS    r2,r1       ; najstarsze 16 bitów
           ANDS    r2,r5
           BNE     label18     ; skocz, gdy było zero = nasycenie
           POP     {r4-r8,pc}
label16    ANDS    r8,r0,r4    ;TU: wynik musi być dodatni
           BNE     label17     ; skocz, gdy minus = nasycenie
           ANDS    r1,r5
           BEQ     label19     ; skocz gdy OK
label17    MOVS    r0,r6       ; nasycenie (+) r0 = 0x7FFFFFFF
           POP     {r4-r8,pc}
label18    MOVS    r0,r4       ; nasycenie (-) r0 = 0x80000000
label19    POP     {r4-r8,pc}
           ENDP
```

Kolejne działanie to dzielenie stałopozycyjne. Zaczniemy od podstaw związanych z formatem liczby. Do rozwiązania problemu, będziemy potrzebowali dzielenia dwóch liczb

całkowitych, 64-bitowych ze znakiem. Uzyskamy część całkowitą wyniku i resztę z dzielenia. Dzielenie liczb 64-bitowych sprowadzimy do dzielenia liczby nieujemnej przez dodatnią. Zastosujemy metodę *long division*. Będzie to najtrudniejszy program ze wszystkich, jakie napisaliśmy do tej pory.



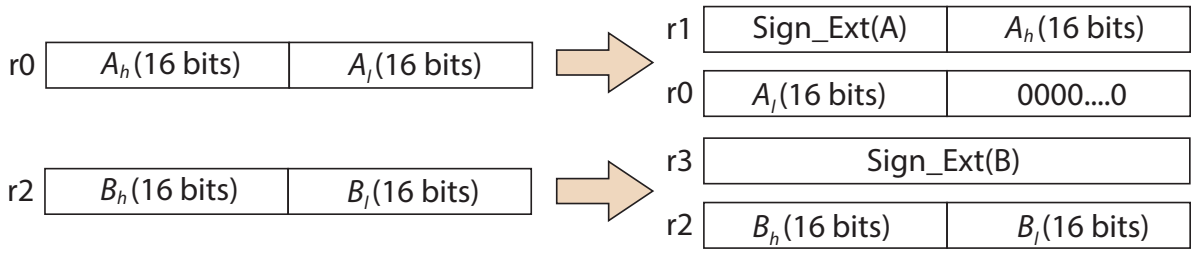
Rys. 14.4: Mnożenie liczb $UQ16.16$ lub $Q15.16$

14.3.2. Dzielenie stałopozycyjne Q15.16

Zaczniemy od podstaw związanych z formatem liczby.

$$f_C = \frac{f_A}{f_B} = \frac{I_A 2^{-n}}{I_B 2^{-n}} = \frac{I_A}{I_B} = I_C 2^{-n} \quad \rightarrow \quad I_C = \frac{I_A 2^n}{I_B} \tag{14.6}$$

Przykład dzielenia liczb w formacie $UQ16.16$ lub $Q15.16$, pokazany jest na rysunku 14.5. Zgodnie ze wzorem (14.6) licznik musimy przesunąć (z zachowaniem znaku) o 16 bitów w lewo (mnożenie przez 2^{16}). Licznik i mianownik zamieniamy na liczby 64-bitowe i wykonujemy dzielenie całkowitoliczbowe. Zgodnie z rysunkiem 14.5 są to dwie liczby *integer* 64-bitowe ze znakiem.



Rys. 14.5: Dzielenie liczb $UQ16.16$ lub $Q15.16$

```
;---------------- Q15.16 dzielenie z nasyceniem ---------------
;INPUT:  r0 = liczba stałopozycyjna ze znakiem Q15.16
;        r1 = liczba stałopozycyjna ze znakiem Q15.16
;OUTPUT: r0 = r0 / r1
;dzielenie przez zero powoduje nasycenie zależne od znaku licznika
           EXPORT Q1516_div
           ALIGN
Q1516_div  PROC
           PUSH    {r4-r8,lr}  ; parzysta ilość rejestrów
           ; czy licznik ułamka jest zerem
           ORRS    r8,r0,r0
           BNE     label20     ; skocz, gdy nie
           POP     {r4-r8,pc}
           ; maski:
label20    MOVS    r2,#0
           MVNS    r3,r2       ; w r3 mam 0xFFFFFFFF
           MOVS    r5,r3
           MOVS    r6,r3
           LSL     r5,#16      ; w r5 mam 0xFFFF0000
           LSR     r6,#1       ; w r6 mam 0x7FFFFFFF
           MVNS    r4,r6       ; w r4 mam 0x80000000
           ; czy dzielenie przez zero ?
           EOR     r7,r0,r1    ; r7[b31] to znak wyniku
           ANDS    r8,r1,r3    ; czy dzielnik jest zerem ?
           BNE     label21     ; skocz, gdy nie
           LSLS    r0,r0,#1    ; bit znaku do Carry
           BCC     label24     ; skocz, gdy C = 0
           B       label25
           ; dzielenie ze znakiem:
label21    MOVS    r2,r1       ; mianownik do r2
           ASRS    r1,r0,#31   ; r1[31:0] bity znaku A
           LSLS    r1,r1,#16   ; przesuwamy
           ORR     r1,r1,r0,LSR #16
           LSLS    r0,r0,#16   ; w r1:r0 jest rozszerzenie A
           ASRS    r3,r2,#31   ; w r3:r2 rozszerzone B
           BL      Int64_sdiv  ; dzielenie liczb całkowitych, wynik w r0
           ; sprawdzę przekroczenie zakresu
           MOVS    r2,#0
           MVNS    r2,r2       ; r2 = 0xFFFFFFFF
           ANDS    r3,r1,r2    ; czy w r1 są jedynki ?
           BNE     label22     ; skocz, gdy TAK
           ANDS    r3,r0,r4    ;TU: r1=0, bit znaku
           BEQ     label26     ; skocz, gdy OK
           B       label24     ; over 0x7FFFFFFF
           ;TU: w r1 są jedynki
label22    EORS    r3,r1,r2    ; czy w r1 same jedynki
           BNE     label23     ; skocz, gdy NIE
           ANDS    r3,r0,r4    ;TU: TAK, bit znaku r0
           BNE     label26     ; skocz, gdy OK
           B       label25     ; over 0x80000000
```

```
            ;TU: w r1 są jedynki i zera
label23     ANDS    r3,r1,r4     ; bit znaku
            BEQ     label24      ; over 0x7FFFFFFF
            B       label25      ; over 0x80000000
label24     MOVS    r0,r6        ; nasycenie (+) r0 = 0x7FFFFFFF
            POP     {r4-r8,pc}
label25     MOVS    r0,r4        ; nasycenie (-) r0 = 0x80000000
label26     POP     {r4-r8,pc}
            ENDP
```

Powyższa funkcja realizuje dzielenie liczb $Q15.16$ zgodnie z rysunkiem 14.5. Wykorzystuje procedurę dzielenia liczb 64-bitowych, całkowitych ze znakiem `Int64_sdiv`.

14.3.3. Dzielenie liczb 64-bitowych ze znakiem

Musimy napisać program dzielenia liczb całkowitych, 64-bitowych ze znakiem. Wynikiem dzielenia jest część całkowita wyniku oraz reszta z dzielenia (obie liczby są całkowite, czyli nie wychodzimy poza pierścień liczb całkowitych), zgodnie ze wzorem

$$\frac{a}{b} = c,\ r \qquad \longrightarrow \qquad a = b \cdot c + r$$

- a – dzielna, licznik ułamka, gdy $a = 0$, to $c = 0, r = 0$
- b – dzielnik, mianownik ułamka, procedura wywołująca musi zapewnić, że $b \neq 0$
- c – część całkowita wyniku dzielenia
- r – reszta z dzielenia

Możliwe są cztery przypadki:

$$\begin{array}{lrcl}
a > 0,\ b > 0 & 7/3 & = 2,\ r = 1 & \longrightarrow \quad 7 = 3 \cdot 2 + 1 \\
a > 0,\ b < 0 & 7/(-3) & = -2,\ r = 1 & \longrightarrow \quad 7 = (-3) \cdot (-2) + 1 \\
a < 0,\ b > 0 & (-7)/3 & = -2,\ r = -1 & \longrightarrow \quad -7 = 3 \cdot (-2) - 1 \\
a < 0,\ b < 0 & (-7)/(-3) & = 2,\ r = -1 & \longrightarrow \quad -7 = (-3) \cdot 2 - 1
\end{array}$$

$$\textit{znak wyniku} = (\textit{znak licznika}) \cdot (\textit{znak mianownika})$$
$$\textit{znak reszty} = \textit{znak licznika}$$

Dzielenie liczb całkowitych, 64-bitowych ze znakiem podzielimy na dwa etapy. Zaczniemy od ustalenia znaku wyniku i znaku reszty, następnie zamienimy licznik i mianownik na liczby dodatnie i wykonamy dzielenie liczb całkowitych 64-bitowych bez znaku. Na koniec ustawimy znaki wyniku i reszty odpowiednio do znaków argumentów. W poniższym programie będziemy dzielić licznik zawarty w rejestrach `(r1:r0)` przez mianownik `(r3:r2)`. Znaki argumentów zapamiętamy w rejestrze `r4`, zgodnie z poniższą regułą

```
; gdy r1 >= 0 i r3 >= 0, to r4[31:30] = (00) EOR (00) = 00
; gdy r1 >= 0 i r3 < 0, to r4[31:30] = (00) EOR (01) = 01
; gdy r1 < 0 i r3 >= 0, to r4[31:30] = (11) EOR (00) = 11
; gdy r1 < 0 i r3 < 0, to r4[31:30] = (11) EOR (01) = 10
; bit r4[31] oznacza znak dzielnej (licznika)
; bit r4[30] czy obie liczby są jednocześnie dodatnie, czy ujemne
```

```
; ******** INTEGER64 funkcje całkowitoliczbowe, 64-bity *******
;------------- Int64_sdiv: 64-bit Signed Division -------------
;INPUT:  (r1:r0)/(r3:r2)dzielenie liczb całkowitych ze znakiem
;OUTPUT: (r1:r0) część całkowita wyniku
;        (r3:r2) reszta z dzielenia
          ALIGN
Int64_sdiv PROC
          PUSH    {r4-r8,lr}  ; parzysta ilość rejestrów
          ; maski:
          MOVS    r6,#1
          ROR     r6,#1       ; r6 = 0x80000000
          ROR     r7,r6,#1    ; r7 = 0x40000000
          ; znak wyniku i znak reszty
          ASRS    r4,r1,#1    ; gdy r1 < 0, to r4[31:30] = 11,
          EOR     r4,r4,r3,LSR #1
          ; zamień licznik (r1:r0) na liczbę dodatnią (gdy jest ujemny)
          BPL     label27     ; skocz, gdy licznik >= 0
          MVNS    r0,r0
          MVNS    r1,r1
          ADDS    r0,#1
          ADC     r1,#0       ; (r1:r0) = -(r1:r0)
          ; zamień mianownik (r3:r2) na liczbę dodatnią (gdy jest ujemny)
label27   TST     r3,r3       ; czy mianownik > 0
          BPL     label28     ; skocz, gdy mianownik > 0
          MVNS    r2,r2
          MVNS    r3,r3
          ADDS    r2,#1
          ADC     r3,#0       ; (r3:r2) = -(r3:r2)
          ; teraz dzielenie liczb dodatnich (bez znaku) (r1:r0)/(r3:r2)
label28   BL      Int64_udiv
          ; jeżeli obie liczby mają różne znaki, to wynik jest ujemny
          TST     r4,r7       ; bit r4[30]
          BEQ     label29     ; skocz, znaki zgodne
          MVNS    r0,r0
          MVNS    r1,r1
          ADDS    r0,#1
          ADC     r1,#0       ; (r1:r0) = -(r1:r0)
          ; znak reszty = znak licznika
label29   TST     r4,r6       ; bit r4[31]
          BEQ     label30     ; skocz, gdy licznik dodatni
          MVNS    r2,r2
          MVNS    r3,r3
          ADDS    r2,#1
          ADC     r3,#0       ; (r3:r2) = -(r3:r2)
label30   POP     {r4-r8,pc}
          ENDP
```

Najtrudniejsza część operacji, to szybkie dzielenie liczb 64-bitowych bez znaku. Wcześniej musimy napisać kilka działań pomocniczych. Zaczniemy od odejmowania liczb całkowitych. Następny program zlicza ilość zer poprzedzających liczbę (*Leading Zeros*). Ostatni to przesunięcie logiczne w lewo.

14.3.4. Operacje pomocnicze na liczbach 64-bitowych

```
;------------------------- Int64_sub -------------------------
;INPUT:  (r1:r0), (r3:r2) dwie liczby
;OUTPUT: (r1:r0) = (r1:r0) - (r3:r2)
            ALIGN
Int64_sub   PROC
            PUSH    {r4-r6,lr}  ; parzysta ilość rejestrów
            SUBS    r4,r0,r2    ; subtract two lower words (update Carry)
            SBC     r5,r1,r3    ; subtract two upper words and Borrow
            MOVS    r1,r5
            MOVS    r0,r4
            POP     {r4-r6,pc}
            ENDP
;---------------- Int64 Counting Leading Zeros ----------------
;INPUT:  (r1:r0) = (upper:lower)
;OUTPUT: r2 = # of leading zero bits in the 64-bit data
            ALIGN
Int64_LeadingZeros  PROC
            ; Counting # of leading zeroes in upper word
            CLZ     r2,r1       ; CLZ = Count Leading Zeros
            ; Counting # of leading zeroes in lower word
            CMP     r2,#32
            CLZEQ   r3,r0       ; if r2 == 32, then count leading zero
                                ; bits of the lower word
            ADDEQ   r2,r2,r3    ; if all bits of the upper word are zero,
                                ; add the leading zeroes of the lower word
            BX      lr
            ENDP
;------------------- Int64 Logic Shift Left -------------------
;INPUT:  (r1:r0) = (upper:lower), r2 = przesunięcie
;OUTPUT: (r1:r0) = (upper:lower) przesunięte
            ALIGN
Int64_ShiftLeft  PROC
            PUSH    {r4-r6,lr}  ; parzysta ilość rejestrów
            MOV     r3,r0       ; backup the lower word
            MOV     r1,r1, LSL r2   ; shift left the upper word
            MOV     r0,r0, LSL r2   ; shift left the lower word
            ; shift bits of the lower word into the upper word
            CMP     r2,#32
            ; if r2 < 32
            RSBLO   r5,r2,#32   ; r5 = 32 - r2
            LSR     r4,r3,r5    ; r4 = r3 >> (32 - r2)
            ORRLO   r1,r1,r4    ; upper |= lower >> (32 - r2)
            ; if r2 > 32
            SUBHS   r5,r2,#32   ; r5 = r2 - 32
            LSLHS   r1,r3,r5    ; upper = lower << (r2 - 32)
            POP     {r4-r6,pc}
            ENDP
```

Do napisania pozostało tylko dzielenie liczb całkowitych dodatnich.

14.3.5. Dzielenie liczb całkowitych dodatnich *long division*

```
;------------------------- Int64_udiv -------------------------
;INPUT:  (r1:r0)/(r3:r2) dzielenie liczb całkowitych dodatnich
;OUTPUT: (r1:r0) część całkowita wyniku, (r3:r2) reszta z dzielenia
            ALIGN
Int64_udiv  PROC
            PUSH    {r4-r12,lr} ; parzysta ilość rejestrów
            MOVS    r8,#0       ; (r1:r0) reminder
            MOVS    r9,r8       ; (r9:r8) quotient = 0
label33     PUSH    {r0-r3}
            POP     {r4-r7}
            ; koniec, gdy licznik < mianownik
            BL      Int64_sub   ; (r1:r0) = (r1:r0) - (r3:r2)
            MOVS    r10,r1
            PUSH    {r4-r7}
            POP     {r0-r3}
            TST     r10,#0x80000000  ; check the sign bit
            BNE     label36
            ; (1) dividend_64_bit
            PUSH    {r0-r3}
            BL      Int64_LeadingZeros  ; INPUT: (r1:r0), r2 = ilość zer
            MOVS    r10,r2      ; r10 = x
            BL      Int64_ShiftLeft
            MOVS    r4,r0
            MOVS    r5,r1
            ; (2) divisor_64_bit
            MOVS    r0,r6
            MOVS    r1,r7
            BL      Int64_LeadingZeros  ; INPUT: (r1:r0), r2 = ilość zer
            MOVS    r11,r2      ; r11 = y
            BL      Int64_ShiftLeft
            MOVS    r6,r0
            MOVS    r7,r1
            ; (3) c_32_bit
label34     UDIV    r12,r5,r7   ; r12 = c_32_bit
            ORRS    r12,#0
            BNE     label35
            ; za duży dzielnik, przesuń (r7:r6) w prawo
            SUB     r11,#1
            LSRS    r7,#1
            RRX     r6,r6
            B       label34
label35     MOVS    r0,r12
            MOVS    r1,#0
            SUB     r2,r11,r10
            BL      Int64_ShiftLeft
            ; quotient = quotient + wynik dzielenia
            ADDS    r8,r8,r0    ; młodsze
            ADC     r9,r9,r1    ; starsze
            ; remainder = remainder - wynik * divisor
```

```
            ; remainder = remainder - wynik * divisor
            MOVS    r4,r0
            MOVS    r5,r1        ; (r5:r4) = wynik dzielenia
            POP     {r0-r3}
            ; mnożenie (wynik)*(divisor)
            UMULL   r6,r7,r4,r2 ; (r7:r6) = r4 * r2
            MLA     r7,r5,r2,r7 ; r7 = r7 + r5 * r2
            MLA     r7,r4,r3,r7 ; r7 = r7 + r4 * r3
            ; remainder = remainder - (wynik)*(divisor)
            PUSH    {r2-r3}
            MOVS    r3,r7
            MOVS    r2,r6
            BL      Int64_sub   ; (r1:r0) = (r1:r0) - (r3:r2)
            POP     {r2-r3}
            B       label33
            ; koniec: copy the results to the return registers
label36     MOVS    r3,r1
            MOVS    r2,r0        ; reminder
            MOVS    r1,r9
            MOVS    r0,r8        ; quotient
            POP     {r4-r12,pc}
            ENDP
```

Zastosowaliśmy metodę dzielenia liczb wielocyfrowych *long division*. Algorytm prześledzimy na przykładzie 224/5 = 44, reszta 4. HEX: 0xE0/5=0x2C, reszta 4. Binarnie:

$$
\begin{array}{cccccccc:ccc}
 & & & 1 & 0 & 1 & 1 & 0 & 0 & & & \\
\hline
1 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & & 1 & 0 & 1 \\
1 & 0 & 1 & & & & & & & & & \\
\cline{1-3}
 & 1 & 0 & 0 & 0 & & & & & & & \\
 & & 1 & 0 & 1 & & & & & & & \\
\cline{4-6}
 & & & 1 & 1 & 0 & & & & & & \\
 & & & 1 & 0 & 1 & & & & & & \\
\cline{4-8}
 & & & & & 1 & 0 & 0 & & & &
\end{array}
$$

W momencie wywołania funkcji `Int64_udiv` w rejestrach (`r1:r0`) zawarta jest dzielna (*dividend*), czyli licznik ułamka, a w rejestrach (`r3:r2`) jest dzielnik (*divisor*), czyli mianownik ułamka. W rejestrach (`r9:r8`) będziemy przechowywać wynik, czyli *quotient*. Rejestry (`r1:r0`) i (`r3:r2`) są wykorzystywane przy wywoływaniu podprogramów, dlatego ich kopie będą przechowywane w rejestrach (`r5:r4`) oraz (`r7:r6`). Od dzielnej będziemy stopniowo odejmować iloczyny kolejnych przybliżeń wyniku pomnożonych przez dzielnik – tak długo, aż kolejny wynik odejmowania stanie się mniejszy od dzielnika. Ten wynik to reszta z dzielenia (*remainder*). Po wykonaniu pięciu pierwszych instrukcji zawartości rejestrów są następujące:

rejestry	**zawartość**	**opis**
`r1:r0`	`0x00000000 : 0x000000E0`	dividend = remainder
`r3:r2`	`0x00000000 : 0x00000005`	divisor
`r5:r4`	`0x00000000 : 0x000000E0`	dividend = remainder – kopia
`r7:r6`	`0x00000000 : 0x00000005`	divisor – kopia
`r9:r8`	`0x00000000 : 0x00000000`	quotient

Teraz sprawdzamy, czy reszta z dzielenia jest mniejsza od dzielnika. W naszym przypadku tak nie jest – przechodzimy do `(1) divident_64_bit` i wyznaczamy ilość zer poprzedzających dzielną. Tą ilość (`0x38`) oznaczymy jako `x` i zapamiętamy w rejestrze `r10`. Dzielną przesuwamy o `x` w lewo. To samo wykonujemy z dzielnikiem. Ilość zer poprzedzających (`0x3D`) oznaczymy `y` i zapamiętamy w rejestrze `r11`. Kolejna czynność to dzielenie (etykieta `label34`). Wynikiem jest jedynka, którą zapamiętujemy w rejestrze `r12`. Ten wynik musimy przesunąć o różnicę `y-x=0x3D-0x38=5` w lewo. Uzyskujemy `0x20`, czyli binarnie `100000`. Jest to pierwsza cyfra wyniku. Dodajemy ją do rejestrów (`r9:r8`). Teraz od dzielnej odejmiemy uzyskany wynik pomnożony przez dzielnik. Doszliśmy do końca pierwszej pętli. Stan rejestrów po pierwszym przebiegu programu:

rejestry	**zawartość**	**opis**
`r1:r0`	`0x00000000 : 0x00000040`	remainder
`r3:r2`	`0x00000000 : 0x00000005`	divisor
`r9:r8`	`0x00000000 : 0x00000020`	quotient

Jak widać, reszta z dzielenia nadal jest większa od dzielnika. Wykonamy więc kolejną pętlę. Jeżeli po przesunięciu obu dzielonych liczb licznik (`0x8000`) okaże się mniejszy od mianownika (`0xA000`), to przesuniemy mianownik w prawo, jednocześnie dekrementując liczbę `y`. Stan rejestrów po drugim przebiegu programu:

rejestry	**zawartość**	**opis**
`r1:r0`	`0x00000000 : 0x00000018`	remainder
`r3:r2`	`0x00000000 : 0x00000005`	divisor
`r9:r8`	`0x00000000 : 0x00000028`	quotient

Nadal reszta z dzielenia jest większa od dzielnika. Wykonamy więc kolejną pętlę. Stan rejestrów po trzecim przebiegu programu:

rejestry	**zawartość**	**opis**
`r1:r0`	`0x00000000 : 0x00000004`	remainder
`r3:r2`	`0x00000000 : 0x00000005`	divisor
`r9:r8`	`0x00000000 : 0x0000002C`	quotient

Teraz już reszta z dzielenia (`0x04`) jest mniejsza od dzielnika (`0x05`), co kończy program. Uzyskaliśmy poprawny wynik (`0x2C`). Zauważmy, że ilość pętli (przebiegów programu) jest równa ilości jedynek w wyniku.

14.4. Projekt17_FixedPoint_sin

Wbrew pozorom zakres stosowania liczb stałopozycyjnych jest bardzo szeroki. Sprawdzają się zawsze, gdy zakres liczb jest ograniczony (na przykład współrzędne pikseli na ekranie), a jednocześnie jesteśmy w stanie określić granicę wymaganej rozdzielczości. Nagrodą jest duża szybkość wykonywanych obliczeń. Ilustracją możliwości liczb stałopozycyjnych będzie program obracający (na ekranie) kontury sześcianu. Zaczniemy od obliczania wartości funkcji $sin(x)$, gdzie x jest wyrażone w radianach. Wykorzystamy rozwinięcie w szereg Taylora

$$sin(x) = \sum_{k=0}^{\infty} \frac{(-1)^k}{(2k+1)!} x^{2k+1} = x - \frac{x^3}{3!} + \frac{x^5}{5!} - \frac{x^7}{7!} + \tag{14.7}$$

Realizacja wzoru (14.7) z zastosowaniem liczb zmiennopozycyjnych:

```
float sinf(float x){
   float ret = 0.0, pow = x, term = x;
   float sign = 1.0, fact = 1.0;
   int k = 1;
   while(fabs(term) >= 0.00001f){
     term = sign*pow/fact;
     ret += term;
     pow *= x*x;
     fact *= 2*k * (2*k+1);
     sign = 0 - sign;
     k++;
   }
   return ret;
}
```

Realizacja stałopozycyjna musi uwzględniać ograniczenie zakresu liczb. We wzorze (14.7) występują nieparzyste potęgi argumentu x^{2k+1}. Jeżeli x jest duże, nastąpi nasycenie, wynikające z przekroczenia zakresu. Dlatego, przed rozpoczęciem obliczeń, musimy sprowadzić wartość argumentu x do przedziału $[0, 2\pi]$. W przypadku ujemnych wartości argumentu x wykorzystamy zależność $sin(-x) = -sin(x)$. Następnie określamy w której ćwiartce okresu zawarty jest argument x. Wzór (14.7) obliczymy dla pierwszej ćwiartki, czyli zakresu $[0, \pi/2]$. Argument i wynik należy przekształcić odpowiednio do ćwiartki, korzystając z własności funkcji $sin(x)$.

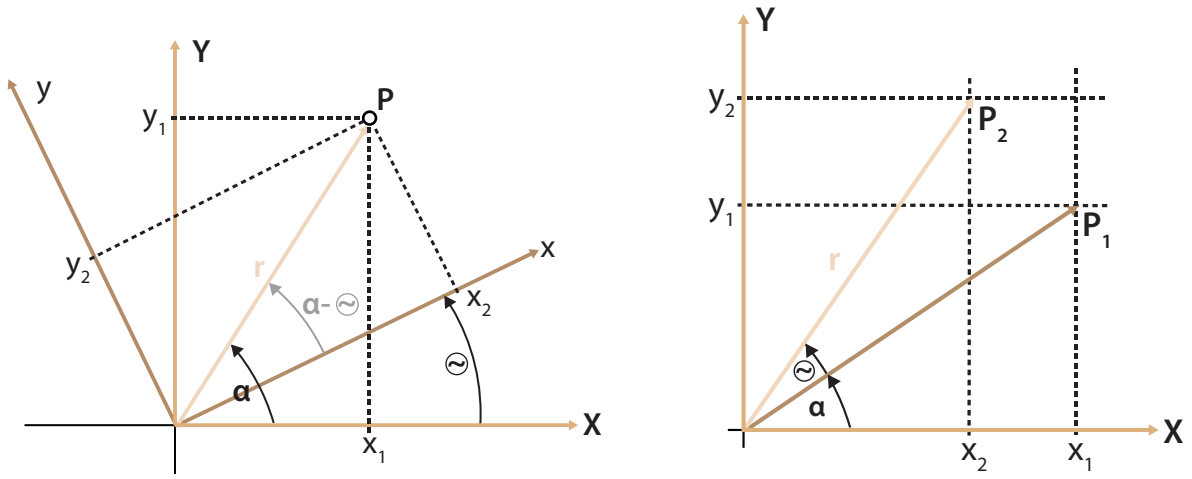
```
int Q1516_sin(int x){
   int minus = 0x00010000, quarter = 0;
   int ret, pow, term, sign, fact, k2, k21, k;
   if(x < 0){
     x = Q32_neg(x);         // sin(-x) = - sin(x)
     minus = Q32_neg(minus);
   }
   while(x > 0x0006487F)        // 2*pi
     x -= 0x0006487F;
   while(x > 0x00019220){        // pi/2
     x -= 0x00019220;
     quarter++;{        // numer ćwiartki
   }
   switch(quarter){
   case 1: x = Q32_sub(0x00019220, x);
           break;
   case 2: minus = Q32_neg(minus);
           break;
   case 3: x = Q32_sub(0x00019220, x);
   minus = Q32_neg(minus);
   }
```

```
  // liczymy sin(x) w pierwszej ćwiartce
  ret = 0; pow = x; term = pow;
  sign = 0x00010000; fact = 0x00010000; k = 0x00010000;
  while(Q32_abs(term) >= 0x00000080){
    term = Q1516_div(pow, fact);
    term = Q1516_mult(sign, term);   // term = sign*pow/fact;
    ret = Q32_add(ret, term);        // ret += term;
    pow = Q1516_mult(pow, x);
    pow = Q1516_mult(pow, x);        // pow *= x*x;
    k2 = Q1516_mult(k, 0x00020000);
    k21 = Q32_add(k2, 0x00010000);
    fact = Q1516_mult(fact, k2);
    fact = Q1516_mult(fact, k21);    // fact *= 2*k * (2*k+1);
    sign = Q32_neg(sign);
    k = Q32_add(k, 0x00010000);      // k++;
  }
  return Q1516_mult(ret, minus);
}
```

Możemy teraz narysować wykres funkcji $sin(x)$ na ekranie graficznym.

14.5. Projekt18_cube3D

Narysujemy obracający się kontur sześcianu, korzystając z wbudowanych funkcji arytmetycznych języka C. Na początek trochę matematyki.



(a) obrót układu odniesienia (b) obrót punktu

Rys. 14.6: Obroty na płaszczyźnie

14.5.1. Obroty na płaszczyźnie

Obracamy układ odniesienia o kąt Θ, punkt P pozostaje niezmienny (rysunek 14.6a). Wektor r jest zaczepiony w początku układu odniesienia. W pierwotnym układzie

$$x_1 = r\cos(\alpha), \qquad y_1 = r\sin(\alpha) \tag{14.8}$$

W nowym układzie odniesienia kąt wektora r jest równy $\alpha - \Theta$, czyli

$$x_2 = r\cos(\alpha - \Theta) = r\cos\alpha \cdot \cos\Theta + r\sin\alpha \cdot \sin\Theta$$
$$y_2 = r\sin(\alpha - \Theta) = r\sin\alpha \cdot \cos\Theta - r\cos\alpha \cdot \sin\Theta$$

Do powyższych równań wstawimy zależności (14.8).

$$\begin{aligned} x_2 &= x_1\cos\Theta + y_1\sin\Theta \\ y_2 &= -x_1\sin\Theta + y_1\cos\Theta \end{aligned}$$

Uzyskaliśmy macierz obrotu układu współrzędnych

$$\begin{bmatrix} x_2 \\ y_2 \end{bmatrix} = \begin{bmatrix} \cos\Theta & \sin\Theta \\ -\sin\Theta & \cos\Theta \end{bmatrix} \begin{bmatrix} x_1 \\ y_1 \end{bmatrix}$$

Obracamy punkt P_1 do punktu P_2 o kąt Θ, układ odniesienia pozostaje niezmienny, rysunek 14.6b.

$$x_1 = r\cos(\alpha), \qquad y_1 = r\sin(\alpha)$$

gdzie r to długość wektorów $\vec{OP_1}$ i $\vec{OP_2}$.

$$x_2 = r\cos(\alpha + \Theta) = r\cos\alpha \cdot \cos\Theta - r\sin\alpha \cdot \sin\Theta = x_1\cos\Theta - y_1\sin\Theta$$
$$y_2 = r\sin(\alpha + \Theta) = r\sin\alpha \cdot \cos\Theta + r\cos\alpha \cdot \sin\Theta = y_1\cos\Theta + x_1\sin\Theta$$

Macierz obrotu punktu

$$\begin{bmatrix} x_2 \\ y_2 \end{bmatrix} = \begin{bmatrix} \cos\Theta & -\sin\Theta \\ \sin\Theta & \cos\Theta \end{bmatrix} \begin{bmatrix} x_1 \\ y_1 \end{bmatrix}$$

Obrót układu odniesienia i obrót punktu w nieruchomym układzie są to, relatywnie, obroty o kąt Θ oraz $-\Theta$, czyli w przeciwnych kierunkach. Zauważmy, że obie macierze są swoimi **transpozycjami**, czyli jeżeli macierz obraca o kąt Θ, to macierz transponowana obraca o kąt $-\Theta$, gdyż $\sin(-\Theta) = \sin(\Theta)$. To oznacza, że

$$\boldsymbol{A}^T\boldsymbol{A} = \boldsymbol{A}\boldsymbol{A}^T = \boldsymbol{I} \qquad \longrightarrow \qquad \boldsymbol{A}^T = \boldsymbol{A}^{-1}$$

gdzie $\boldsymbol{I}$ to macierz jednostkowa.

14.5.2. Macierze obrotu

Każda macierz obrotu (*rotation matrix*) ma wyznacznik równy jeden, $det(\boldsymbol{A}) = 1$ oraz jest ortogonalna $\boldsymbol{A}^T\boldsymbol{A} = \boldsymbol{I}$, $\boldsymbol{A}^T = \boldsymbol{A}^{-1}$. Udowodnimy teraz, że **każda ortogonalna macierz $\boldsymbol{A}_{2\times 2}$ o wyznaczniku równym 1 jest macierzą obrotu**.

Macierz o rozmiarze 2×2 ortogonalną i taką, że $det(A) = 1$ możemy zapisać w postaci:

$$\boldsymbol{A} = \begin{bmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{bmatrix} \qquad \longrightarrow \qquad det(A) = a_{11}a_{22} - a_{12}a_{21} = 1$$

$$\boldsymbol{A}^T\boldsymbol{A} = \begin{bmatrix} a_{11} & a_{21} \\ a_{12} & a_{22} \end{bmatrix} \begin{bmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{bmatrix} = \begin{bmatrix} a_{11}^2 + a_{21}^2 & a_{11}a_{12} + a_{21}a_{22} \\ a_{11}a_{12} + a_{21}a_{22} & a_{12}^2 + a_{22}^2 \end{bmatrix} = \boldsymbol{I}$$

Otrzymujemy układ równań

$$\begin{aligned}
&(1)\quad a_{11}a_{22} - a_{12}a_{21} = 1\\
&(2)\quad a_{11}^2 + a_{21}^2 = 1\\
&(3)\quad a_{12}^2 + a_{22}^2 = 1\\
&(4)\quad a_{11}a_{12} + a_{21}a_{22} = 0
\end{aligned}$$

Równania dodamy stronami: $(2) + (3) - 2 \cdot (1)$

$$\begin{gathered}
a_{11}^2 + a_{21}^2 + a_{12}^2 + a_{22}^2 - 2(a_{11}a_{22} - a_{12}a_{21}) = 1 + 1 - 2 = 0\\
a_{11}^2 - 2a_{11}a_{22} + a_{22}^2 \;+\; a_{12}^2 + 2a_{12}a_{21} + a_{21}^2 = 0\\
(a_{11} - a_{22})^2 + (a_{12} + a_{21})^2 = 0
\end{gathered}$$

Otrzymaliśmy dwa warunki

$$a_{11} - a_{22} = 0 \quad \rightarrow \quad a_{11} = a_{22}\ , \qquad\qquad a_{12} + a_{21} = 0 \quad \rightarrow \quad a_{12} = -a_{21}$$

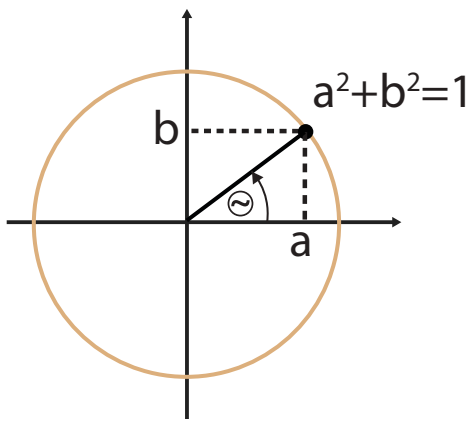
Oznaczmy $a_{11} = a_{22} = a$ oraz $a_{12} = b = -a_{21}$. Otrzymujemy macierz postaci

$$\boldsymbol{A} = \begin{bmatrix} a & b \\ -b & a \end{bmatrix}$$

Wyznacznik tej macierzy $det(\boldsymbol{A}) = a^2 + b^2 = 1$. To oznacza, że punkt na płaszczyźnie $[a, b]^T$ spełnia równanie okręgu o promieniu jednostkowym, rysunek 14.7. Istnieje więc taki kąt Θ, że $a = \cos\Theta$, $b = \sin\Theta$. Otrzymaliśmy macierz obrotu:

$$\boldsymbol{A} = \begin{bmatrix} \cos\Theta & \sin\Theta \\ -\sin\Theta & \cos\Theta \end{bmatrix}$$

Operatory obrotu w $\mathcal{R}^2$ są ortogonalnymi macierzami 2×2 o wyznaczniku jednostkowym. Zależność odwrotna też jest prawdziwa: każda taka macierz jest operatorem obrotu. Ponieważ iloczyn macierzy ortogonalnych jest macierzą ortogonalną oraz wyznacznik iloczynu macierzy jest iloczynem ich wyznaczników, to iloczyn dwóch operatorów obrotu jest także operatorem obrotu.

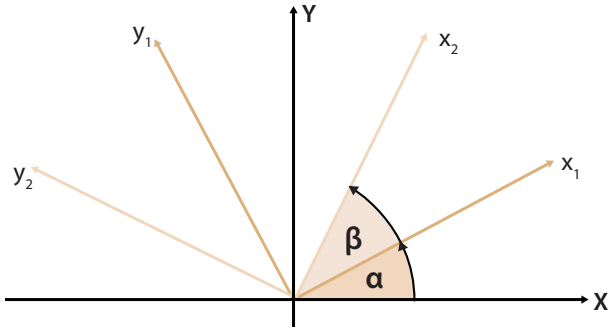


Rys. 14.7: Okrąg jednostkowy

14.5.3. Złożenie obrotów na płaszczyźnie

Sekwencja, czyli złożenie dwóch obrotów: najpierw układ odniesienia obracamy o kąt α, a nastepnie o kąt β. Dowolny, określony wektor $\boldsymbol{v}$, będzie obrócony o kąt α: $\boldsymbol{v_1} = \boldsymbol{A_\alpha v}$, a następnie o kąt β: $\boldsymbol{v_2} = \boldsymbol{A_\beta v_1}$, gdzie

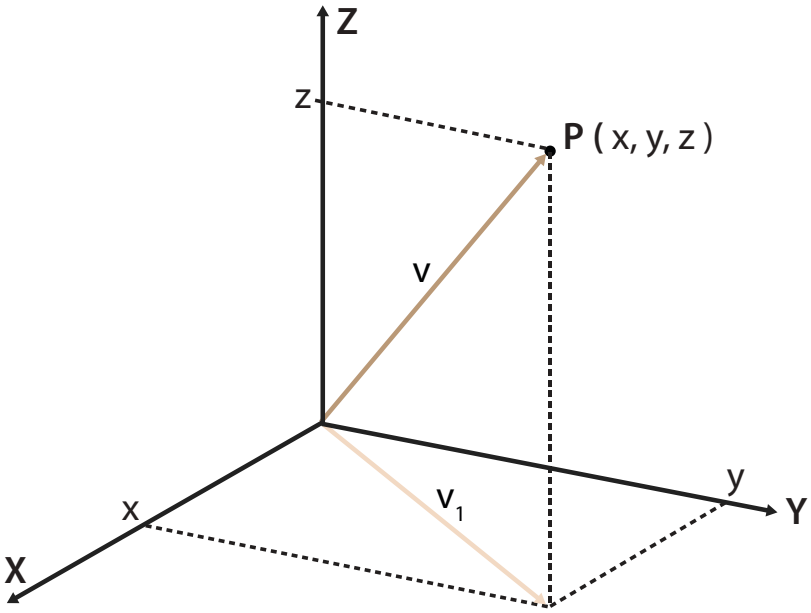
$$\boldsymbol{A_\alpha} = \begin{bmatrix} \cos\alpha & \sin\alpha \\ -\sin\alpha & \cos\alpha \end{bmatrix} \qquad\qquad \boldsymbol{A_\beta} = \begin{bmatrix} \cos\beta & \sin\beta \\ -\sin\beta & \cos\beta \end{bmatrix}$$



Rys. 14.8: Złożenie obrotów na płaszczyźnie

Złożenie tych obrotów (rysunek 14.8) to:

$$\begin{aligned}
\boldsymbol{A}_{\alpha}\boldsymbol{A}_{\beta} &= \begin{bmatrix} \cos\alpha & \sin\alpha \\ -\sin\alpha & \cos\alpha \end{bmatrix} \begin{bmatrix} \cos\beta & \sin\beta \\ -\sin\beta & \cos\beta \end{bmatrix} \\
&= \begin{bmatrix} \cos\beta\cos\alpha - \sin\beta\sin\alpha & \cos\beta\sin\alpha + \sin\beta\cos\alpha \\ -\sin\beta\cos\alpha - \cos\beta\sin\alpha & -\sin\beta\sin\alpha + \cos\beta\cos\alpha \end{bmatrix} \\
&= \begin{bmatrix} \cos(\alpha+\beta) & \sin(\alpha+\beta) \\ -\sin(\alpha+\beta) & \cos(\alpha+\beta) \end{bmatrix} = \boldsymbol{A}_{\alpha+\beta}
\end{aligned}$$



Rys. 14.9: Wektor w przestrzeni $\mathcal{R}^3$

14.5.4. Obroty w przestrzeni $\mathcal{R}^3$

Załóżmy, że punkt $\boldsymbol{P}$ w przestrzeni trójwymiarowej $\mathcal{R}^3$ ma współrzędne $[x, y, z]$. Wektor wodzący tego punktu oznaczymy jako $\boldsymbol{v}$. Obrócimy cały układ odniesienia wokół osi OZ o kąt Ψ. Nowe współrzędne punktu, w obróconym układzie odniesienia, oznaczymy x_1, y_1, z_1. Jest oczywiste, że obrót wokół osi OZ nie zmienia współrzędnej z punktu, mamy więc $z_1 = z$. W celu określenia nowych współrzędnych x_1, y_1 punktu $\boldsymbol{P}$, rozważmy wektor $\boldsymbol{v_1}$ na rysunku 14.9. Wektor $\boldsymbol{v_1}$ jest rzutem wektora $\boldsymbol{v}$ na płaszczyznę OXY. W $\mathcal{R}^2$

współrzędnymi $\boldsymbol{v_1}$ są $[x_1, y_1]$. Obrót wektora $\boldsymbol{v_1}$ w przestrzeni $\mathcal{R}^3$ o kąt Ψ jest równoważny obrotowi o ten sam kąt na płaszczyźnie OXY. Mamy więc równania

$$\begin{aligned} x_1 &= x\cos\Psi + y\sin\Psi \\ y_1 &= -x\sin\Psi + y\cos\Psi \end{aligned}$$

które połączymy z równaniem $z_1 = z$ w jeden układ

$$\begin{aligned} x_1 &= x\cos\Psi + y\sin\Psi + 0\cdot z \\ y_1 &= -x\sin\Psi + y\cos\Psi + 0\cdot z \\ z_1 &= 0\cdot x + 0\cdot y + 1\cdot z \end{aligned}$$

Uzyskaliśmy macierz obrotu 3×3, w przestrzeni $\mathcal{R}^3$, wokół osi OZ o kąt Ψ

$$\boldsymbol{A}_Z = \begin{bmatrix} \cos\Psi & \sin\Psi & 0 \\ -\sin\Psi & \cos\Psi & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

Z powyższego wynika, że jeżeli w przestrzeni $\mathcal{R}^3$ obrót jest wykonywany wokół osi OY, to jedynka powinna być na pozycji $a_{2,2}$ macierzy obrotu, ponieważ tylko współrzędne x oraz z ulegają zmianie. Tak więc, macierz obrotu 3×3, w przestrzeni $\mathcal{R}^3$, wokół osi OY o kąt Θ ma postać

$$\boldsymbol{A}_Y = \begin{bmatrix} \cos\Theta & 0 & -\sin\Theta \\ 0 & 1 & 0 \\ \sin\Theta & 0 & \cos\Theta \end{bmatrix}$$

Analogicznie, macierz obrotu 3×3, w przestrzeni $\mathcal{R}^3$, wokół osi OX o kąt Φ ma postać

$$\boldsymbol{A}_X = \begin{bmatrix} 1 & 0 & 0 \\ 0 & \cos\Phi & \sin\Phi \\ 0 & -\sin\Phi & \cos\Phi \end{bmatrix}$$

14.5.5. Złożenie obrotów w przestrzeni $\mathcal{R}^3$

Jeżeli wykonujemy obrót przestrzeni $\mathcal{R}^3$ wokół osi przechodzącej przez początek układu odniesienia, to wszystkie punkty leżące na tej osi nie zmieniają swego położenia, czyli oś obrotu pozostaje stała. Możemy wyobrazić sobie płaszczyznę, która zawiera oś obrotu: wektory leżące na tej płaszczyźnie obracają się z całą płaszczyzną, ale nie zmieniają swego wzajemnego położenia. W szczególności nie zmieniają się ich długości, ani kąt między nimi, czyli ich iloczyn skalarny jest stały w trakcie obrotu. Mówimy, że **iloczyn skalarny jest niezmiennikiem obrotu**.

> Macierz 3×3 jest operatorem obrotu w przestrzeni $\mathcal{R}^3$ wtedy i tylko wtedy, gdy jest ortogonalna i jej wyznacznik jest równy 1

Dowód Iloczyn skalarny jest niezmiennikiem obrotu, czyli

$$(\boldsymbol{A}\boldsymbol{v_1})^T(\boldsymbol{A}\boldsymbol{v_2}) = \boldsymbol{v_1}^T\boldsymbol{A}^T\boldsymbol{A}\boldsymbol{v_2} = \boldsymbol{v_1}\boldsymbol{v_2} \qquad \longrightarrow \qquad \boldsymbol{A}^T\boldsymbol{A} = \boldsymbol{I}$$

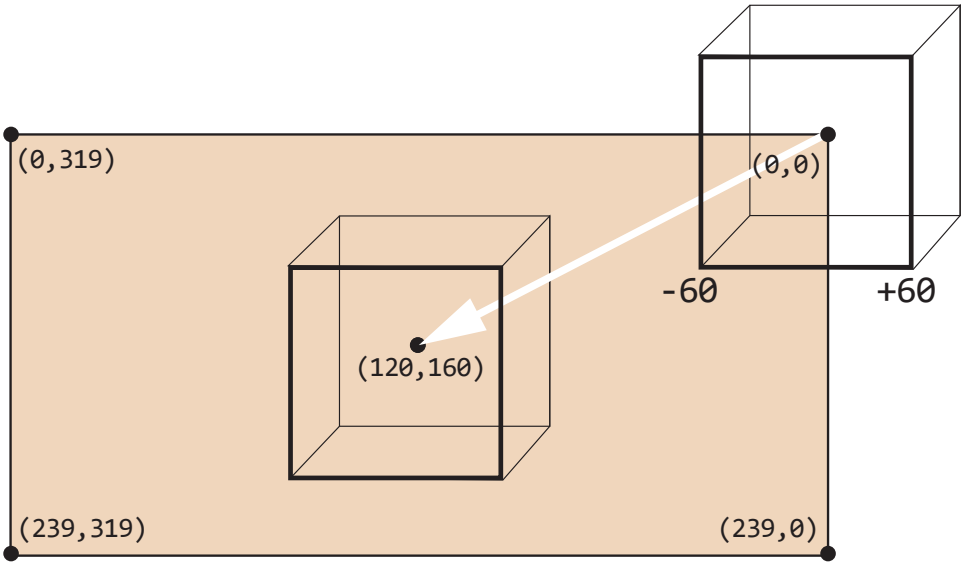
czyli macierz obrotu jest ortogonalna. Wiemy, że $det(\boldsymbol{A}^T) = det(\boldsymbol{A})$. Wyznacznik iloczynu macierzy jest iloczynem ich wyznaczników, czyli

$$det(\boldsymbol{A}^T\boldsymbol{A}) = det(\boldsymbol{A}^T) \cdot det(\boldsymbol{A}) = [det(\boldsymbol{A})]^2 = det(\boldsymbol{I}) = 1$$

To oznacza, że $det(\boldsymbol{A}) = 1$, albo $det(\boldsymbol{A}) = -1$. Ale obrót zachowuje wzajemne położenie osi układu odniesienia. To oznacza, że musi być $det(\boldsymbol{A}) = 1$, gdyż macierz, która ma $det(\boldsymbol{A}) = -1$ zmienia kierunek jednej osi. Przykładem jest macierz

$$\boldsymbol{B} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & -1 \end{bmatrix} \qquad det(\boldsymbol{B}) = -1$$

która zmienia kierunek osi OZ.



Rys. 14.10: Obrót sześcianu bazowego i przesunięcie do środka ekranu

14.5.6. Projekt18_cube3D

Naszym zadaniem jest narysowanie sześcianu, który będzie obracał się wokół własnego środka. W tym celu utworzymy sześcian bazowy, którego środek pokrywa się dokładnie ze środkiem układu odniesienia (rysunek 14.10). Oś obrotu zdefiniujemy określając trzy kąty: Ψ, Θ oraz Φ. Mając ustalone kąty, wyznaczamy trzy macierze obrotu: $\boldsymbol{A_Z}(\Psi)$, $\boldsymbol{A_Y}(\Theta)$ oraz $\boldsymbol{A_X}(\Phi)$. Macierz obrotu całego sześcianu jest iloczynem tych trzech macierzy: $\boldsymbol{A} = \boldsymbol{A_Z}\boldsymbol{A_Y}\boldsymbol{A_X}$.

W kolejnych wykonaniach pętli głównej mnożymy współrzędne wierzchołków sześcianu bazowego przez macierz obrotu $\boldsymbol{A}$. Przesunięcie na środek ekranu polega na dodaniu do współrzędnych x_i wszystkich punktów, liczby 120.0, a do współrzędnych y_i, liczby 160.0. Trójwymiarowy sześcian będziemy rzutowali na płaski ekran, dlatego w czasie rysowania pomijamy współrzędne z_i. Łącząc odcinkami prostymi odpowiednie pary punktów uzyskujemy rysunek konturów sześcianu.

14.6. Projekt19_cube3D_FixedPoint

Kolejny projekt polega na powtórzeniu wszystkich operacji z wykorzystaniem wyłącznie arytmetyki stałopozycyjnej w formacie $Q15.16$. Sześcian jest wyświetlany identycznie jak z zastosowaniem arytmetyki zmiennopozycyjnej. Na początku programu tworzony jest sześcian bazowy i później nigdy nie jest odświeżany. Błędy wynikające z reprezentacji liczb sumują się. W efekcie sześcian wyświetlany z wykorzystaniem arytmetyki $Q15.16$ zmniejsza swoje wymiary, co jest widoczne już po kilku godzinach. Sześcian rysowany z wykorzystaniem reprezentacji zmiennoprzecinkowej *double* nie zmienia kształtu nawet po wielu dniach ciągłego wyświetlania.

Rozdział 15

Bezpośredni dostęp do pamięci DMA

Mechanizm bezpośredniego dostępu do pamięci (*Direct Memory Access*) DMA jest użyteczną techniką przesyłania danych z pominięciem procesora, pomiędzy pamięcią i układami peryferyjnymi oraz z jednego obszaru pamięci do innego.

- Gdy układ peryferyjny jest bardzo wolny, DMA zwalnia procesor z konieczności oczekiwania na gotowość tego układu. Przykładem jest transmisja szeregowa z bardzo małą prędkością bodową. Nadawanie / odbiór dużego bloku danych zajmuje dużo czasu, ale dzięki DMA jest wykonywane w tle. Procesor może realizować inne zadania.
- W przypadku bardzo szybkich transmisji, DMA zapewnia sprawny odbiór i zapisanie w pamięci bloku danych. Także w czasie nadawania DMA jest w stanie dostarczyć szybko strumień danych z pamięci do nadajnika.
- Gdy system rejestruje sygnały analogowe, mechanizm DMA połączony z przetwornikiem A/D umieszcza próbki w pamięci bez angażowania procesora.

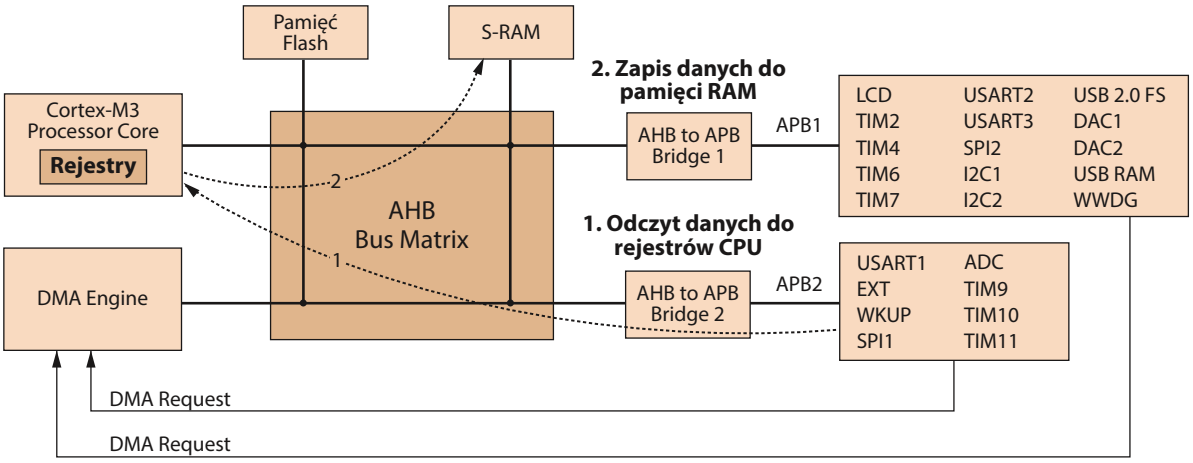
Dzięki technice DMA układ peryferyjny, na przykład port szeregowy USART, nie musi mieć własnej, lokalnej pamięci FIFO, gdyż dane są sprawnie transferowane pomiędzy tym układem a pamięcią systemu.

W mikrokontrolerach STM32 dostępne są układy DMA1 i DMA2. Pierwszy z nich jest 7-kanałowy i występuje we wszystkich mikrokonrolerach STM32. Drugi jest 5-kanałowy i jest dostępny tylko w układach *high-density*, *XL-density* oraz *connectivity*. W płycie ZL27ARM jest zainstalowany procesor STM32F103VB *medium-density* wyposażony w jeden układ DMA1 (rysunek 15.3). Płyta STM32F10E-EVAL zawiera procesor STM32F103ZE *high-density* w którym są dwa układy: DMA1 i DMA2 (rysunek 15.4). Oba kontrolery umożliwiają przesyłanie danych do/z pojedynczych komórek pamięci (oraz rejestrów peryferiali, gdyż są one mapowane do pamięci) oraz transfer całych bloków danych o maksymalnej długości 65'535 elementów. Możliwe są transfery

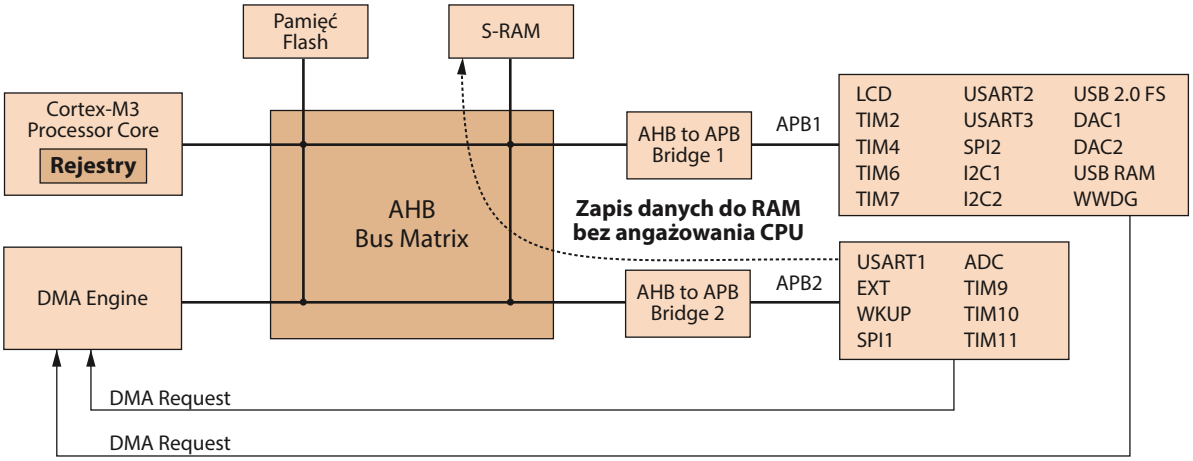
- z pamięci do pamięci
- z pamięci do układu peryferyjnego
- z układu peryferyjnego do pamięci
- z układu peryferyjnego do układu peryferyjnego

15.1. DMA Bus Matrix

Układy DMA są podłączone do zaawansowanej magistrali *advanced high-performance bus* AHB. Wszystkie peryferiale są dołączone do magistral *advanced peripheral bus* APB1 (36 MHz) i APB2 (72 MHz). AHB ma większe pasmo niż APB, dlatego potrzebny jest układ nadzorujący współpracę pomiędzy magistralami. Ten układ to *Bus Matrix*.



Rys. 15.1: Przepisanie bajtu z odbiornika USART1 do pamięci przez procesor



Rys. 15.2: Transfer DMA bajtu z odbiornika USART1 do pamięci

Bus Matrix jest układem arbitrażu magistral wewnętrznych mikrokontrolera. Dostęp poszczególnych układów peryferyjnych i rdzenia do układu *Bus Matrix* jest przydzielany według algorytmu karuzelowego (*round robin*). Polega to na tym, że poszczególne układy kolejno otrzymują od arbitra możliwość dostępu do magistrali. Dzięki temu nie ma niebezpieczeństwa zablokowania magistrali przez któryś z układów. Kiedy przyjdzie kolej na układ DMA, otrzyma on dostęp, ale tylko na jeden cykl. Jest to transfer z podkradaniem (*bus stealing*). Transfer dużego bloku danych, przy wykorzystaniu mechanizmu DMA nie zablokuje więc procesora – co najwyżej nieco spowolni jego pracę.

Porównamy teraz operację odbierania bajtu z kanału transmisji szeregowej bezpośrednio przez procesor z tą samą operacją wykonaną przez układ DMA. Rysunek 15.1 pokazuje pierwszy przypadek (procesor). Bez mechanizmu DMA, procesor musi wykonać dwie operacje wymagające dostępu do magistrali: odczyt bajtu z układu USART do rejestru procesora oraz zapisanie tego bajtu z rejestru do pamięci. W rzeczywistości operacji wymagających dostępu do magistrali jest znacznie więcej, gdyż operacja przepisania bajtu z USART do pamięci wykonywana jest programowo: czyli towarzyszy jej znaczna ilość instrukcji pobieranych z pamięci programu. Ponadto należy rozwiązać problem wykrycia gotowości układu USART - najkosztowniejszy sposób to podczytywanie, bardziej elegancki to obsługa przerwania wystawionego przez USART po odebraniu bajtu.

Rysunek 15.2 to drugi przypadek (DMA). Transfer bajtu z USART do pamięci odbywa się w jednym cyklu dostępu do magistrali, bez ingerencji procesora. Po skonfigurowaniu i załączeniu kanału DMA, bajty odbierane przez USART są przesyłane do kolejnych komórek pamięci. DMA może zgłosić przerwanie po odebraniu bloku danych, które informuje procesor, że dane oczekują w pamięci na przetworzenie. W ten sposób oba procesy: transferu i przetwarzania danych są wykonywane równolegle, co znacznie zwiększa ogólną wydajność systemu.

15.2. Kanały DMA

Połączenia układów peryferyjnych do poszczególnych kanałów układu DMA1 pokazane są w tabeli 15.1, a układu DMA2 w tabeli 15.2. W określonym czasie, każdy kanał DMA jest w stanie obsłużyć tylko jeden układ peryferyjny. Na przykład, do kanału 1, DMA1 są dołączone ADC1, TIM2_CH3 oraz TIM4_CH1. Program może skonfigurować kanał 1 układu DMA1 na obsługę tylko jednego z tych urządzeń.

Tabela 15.1: Kanały układu DMA1

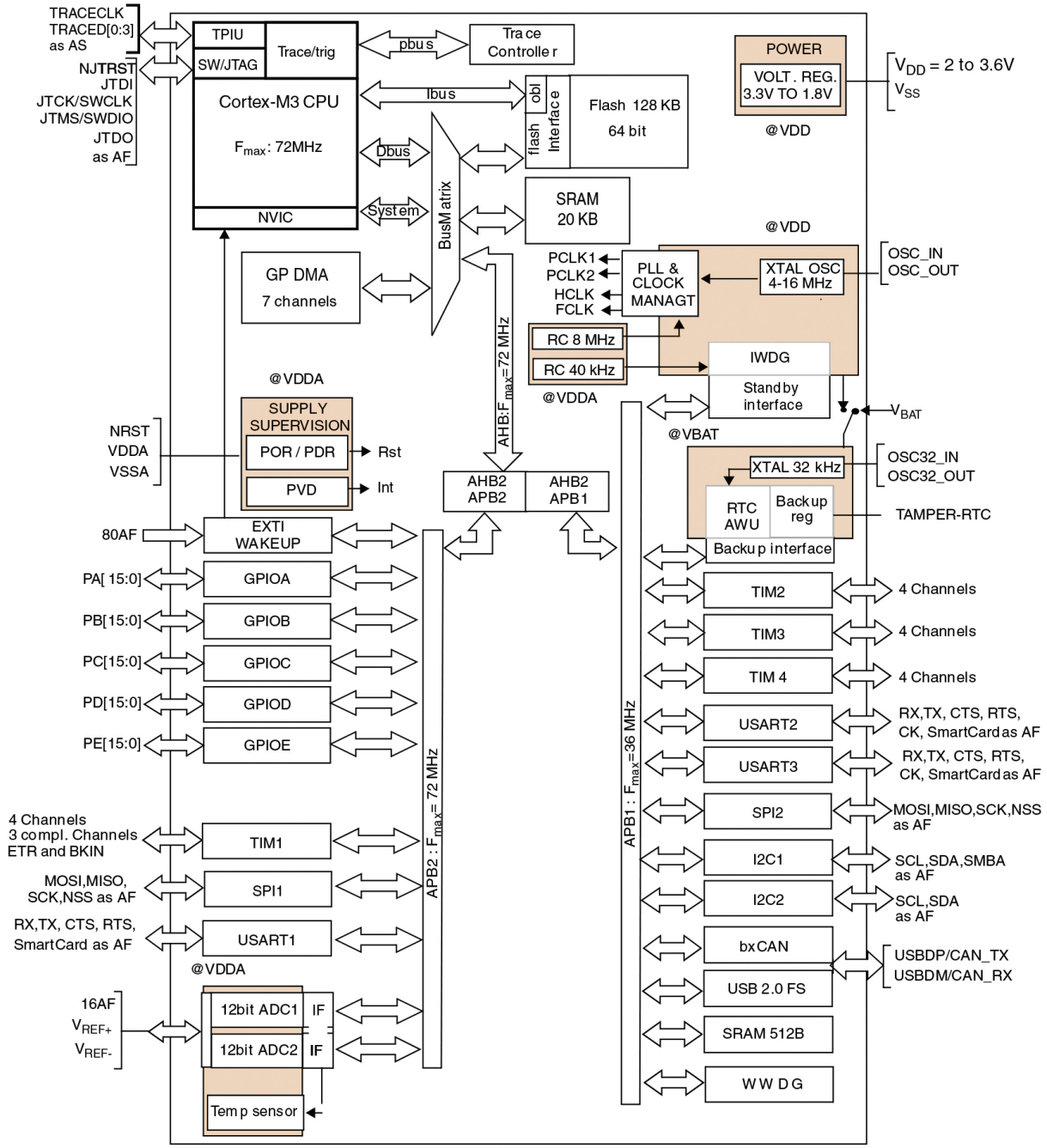
Układ	Kanał 1	Kanał 2	Kanał 3	Kanał 4	Kanał 5	Kanał 6	Kanał 7
ADC1	ADC1	–	–	–	–	–	–
SPI	–	SPI1_RX	SPI1_TX	SPI2_RX	SPI2_TX	–	–
USART	–	USART3_TX	USART3_RX	USART1_TX	USART1_RX	USART2_RX	USART2_TX
I^2C	–	–	–	I2C2_TX	I2C2_RX	I2C1_TX	I2C1_RX
TIM1	–	TIM1_CH1	–	TIM1_CH4 TIM1_TRIG TIM1_COM	TIM1_UP	TIM1_CH3	–
TIM2	TIM2_CH3	TIM2_UP	–	–	TIM2_CH1	–	TIM2_CH2 TIM2_CH4
TIM3	–	TIM3_CH3	TIM3_CH4 TIM3_UP	–	–	TIM3_CH1 TIM3_TRIG	–
TIM4	TIM4_CH1	–	–	TIM4_CH2	TIM4_CH3	–	TIM4_UP

Tabela 15.2: Kanały układu DMA2 (STM32F103ZE)

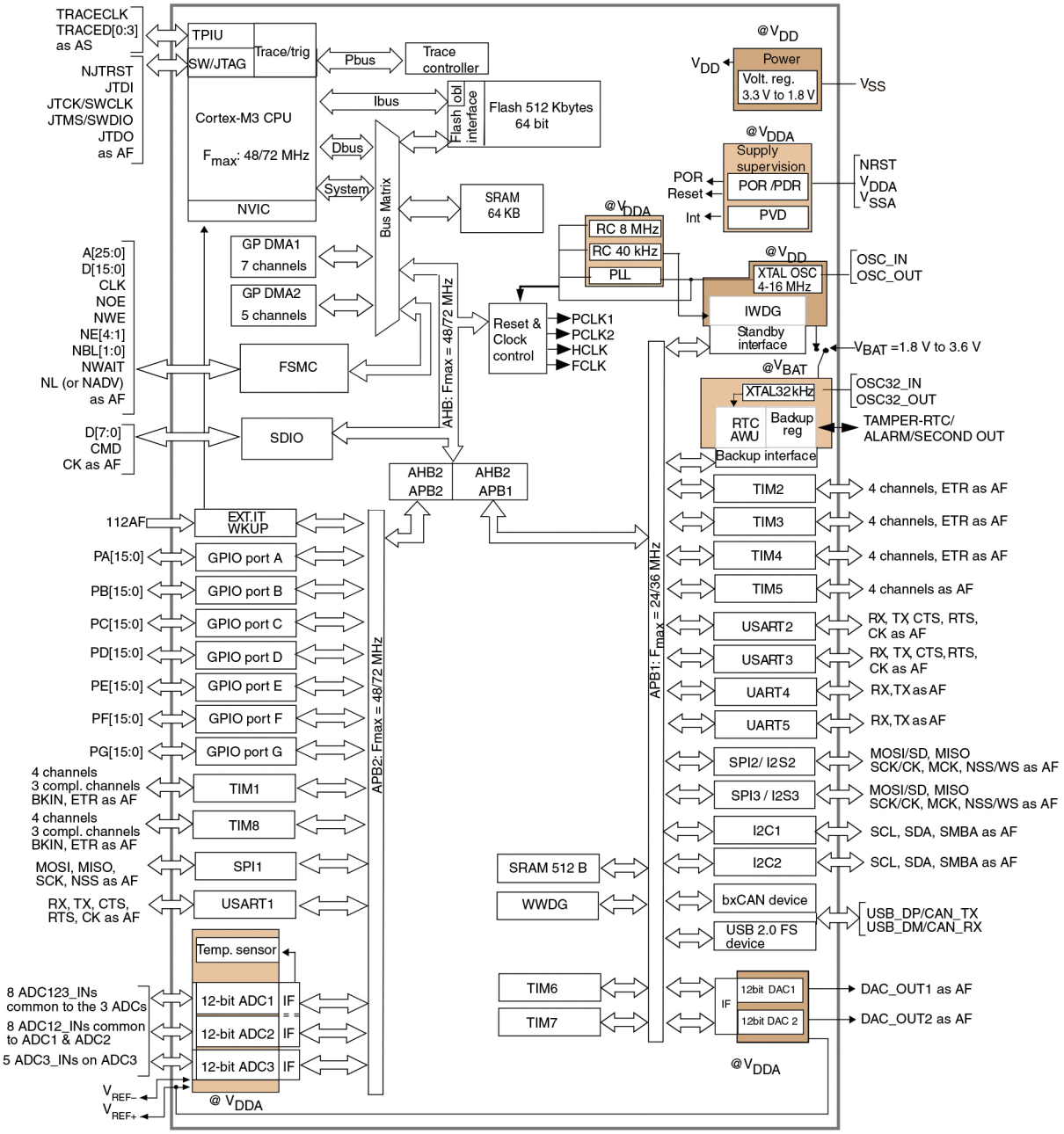
Układ	Kanał 1	Kanał 2	Kanał 3	Kanał 4	Kanał 5
ADC3	–	–	–	–	ADC3
SPI/I2S3	SPI/I2S3_RX	SPI/I2S3_TX	–	–	–
UART4	–	–	UART4_RX	–	UART4_TX
SDIO	–	–	–	SDIO	–
TIM5	TIM5_CH4 TIM5_TRIG	TIM5_CH3 TIM5_UP	–	TIM5_CH2	TIM5_CH1
TIM6/ DAC_Channel1	–	–	–	SDIO	–
TIM7	–	–	–	TIM7_UP/ DAC_Channel2	–
TIM8	TIM8_CH3 TIM8_UP	TIM8_CH4 TIM8_TRIG TIM8_COM	TIM8_CH1	–	TIM8_CH2

Każdemu kanałowi niezależnie, można nadać jeden z czterech priorytetów: *very high, high, medium* oraz *low*. Gdy wykorzystywanych jest wiele kanałów, wysoki priorytet powinien być nadany kanałowi transferującemu duże bloki danych: zapewni to płynność

transferu. Priorytet transferu z odbiornika transmisji szeregowej do pamięci powinien być na tyle duży, aby było zapewnione przeniesienie bieżącego bajtu do pamięci SRAM, przed odebraniem kolejnego, gdyż układy transmisjii szeregowej nie mają podręcznych pamięci. Dotyczy to wszystkich układów akwizycji danych, na przykład przetwornika analogowo–cyfrowego. Jeżeli dwa kanały mają ten sam priorytet programowy, to pierwszy będzie obsłużony ten o mniejszym numerze.



Rys. 15.3: Schemat mikrokontrolera STM32F103VB



Rys. 15.4: Schemat mikrokontrolera STM32F103ZE

15.3. Programowanie DMA

Układ DMA ma dwa rejestry wspólne dla wszystkich kanałów

- DMA *Interrupt Status Register* (`DMA_ISR`) – tylko do odczytu. W tym rejestrze każdy kanał ma przydzielone 4 flagi: `TEIFx` *transfer error flag*, `HTIFx` *half transfer flag*, `TCIFx` *transfer complete flag*, `GIFx` *global interrupt flag*.
- DMA *Interrupt Flag Clear Register* (`DMA_IFCR`) – tylko do zapisu. Wpisanie jedynki kasuje odpowiadającą jej flagę zgłoszenia przerwania.

Każdy kanał może więc zgłosić trzy przerwania: po osiągnięciu połowy transferu i po jego zakończeniu oraz po wykryciu błędu.

Kanały numerowane są od 1 do 7 dla DMA1 i od 1 do 5 dla DMA2. Każdy kanał konfigurowany jest za pomocą czterech rejestrów

- DMA *Channel x Memory Address Register* (`DMA_CMARx`), gdzie `x` to numer kanału. 32-bitowy rejestr początku obszaru pamięci związanej z transferem. DMA może przesyłać dane 8–, 16– oraz 32–bitowe. Adres w rejestrze `DMA_CMARx` muszą być wyrównane odpowiednio, czyli gdy przesyłane są dane 32-bitowe, to dwa najmłodsze bity adresu muszą być zerowe, gdy 16–bitowe, to jeden najmłodszy bit jest zerowy. Należy o tym pamiętać przy przesyłaniu fotografii do wyświetlacza graficznego: obrazek musi być wyrównany do czterech bajtów (transfer jest 32–bitowy).
- DMA *Channel x Peripheral Address Register* (`DMA_CPARx`), 32 bity. Także ten adres musi być wyrównany do rozmiaru transferowanych danych.
- DMA *Channel x Number of Data Register* (`DMA_CNDTRx`). Ilość elementów do przesłania. Rejestr 16–bitowy, można więc przesłać od 0 do 65'535 elementów, na przykład słów 32–bitowych. Jest to licznik przesyłanych elementów, a nie licznik bajtów.
- DMA *Channel x Configuration Register* (`DMA_CCRx`) – konfiguracja kanału.

Ostatni rejestr pozwala na wybór konfiguracji: transfer pamięć ↔ pamięć/peryferial, priorytet kanału, rozmiar elementu tansferu 8/16/32, inkrementacja adresów tak/nie, kierunek transferu, zezwolenie przerwań i zezwolenie kanału, (szczegóły w dokumencie RM0008). Inkrementacja adresu zależy od wybranego rozmiaru elementu: na przykład dla słów 32–bitowych adres będzie zwiększany o 4.

15.4. Projekt20_DMA_M2M

W kolejnych rozdziałach będziemy łączyć DMA z przetwornikiem analogowo-cyfrowym oraz kanałem transmisji szeregowej. Dlatego teraz ustawimy DMA na transfer z jednego obszaru pamięci do drugiego. W pętli głównej będziemy inicjowali transfery znaków naprzemiennie z dwóch różnych obszarów pamięci stałej FLASH do określonego obszaru pamięci S-RAM. Następnie będziemy wyświetlać przepisane znaki. Należy pamiętać o załączeniu taktowania układu DMA w funkcji inicjującej RCC:

```
RCC_AHBPeriphClockCmd(RCC_AHBPeriph_DMA1, ENABLE); //włącz taktowanie DMA
```

Zdefiniujemy bufor na znaki w pamięci S-RAM i dwie stałe tekstowe w pamięci FLASH

```
// zmienne globalne -------------------------------------------
uint8_t txt2LCD[21] = {0};     // bufor funkcji sprintf
const uint8_t txt1[]= "-- pierwszy tekst --/0";
const uint8_t txt2[]= "--- drugi tekst ---/0";
```

Programowanie DMA polega na wypełnieniu zmiennej strukturalnej typu `DMA_InitTypeDef`, a następnie zapisaniu jej za pomocą funkcji `DMA_Init`.

```
DMA_InitTypeDef DMA_InitStructure;

DMA_DeInit(DMA1_Channel1);     // usuń poprzednią konfigurację DMA
```

```
DMA_InitStructure.DMA_PeripheralBaseAddr = (uint32_t) txt2LCD; // adres
DMA_InitStructure.DMA_MemoryBaseAddr = (uint32_t) txt1; // adres
DMA_InitStructure.DMA_DIR = DMA_DIR_PeripheralDST; // kierunek transferu
DMA_InitStructure.DMA_BufferSize = 21; // ilość elementów do przesłania
DMA_InitStructure.DMA_PeripheralInc = DMA_PeripheralInc_Enable;
DMA_InitStructure.DMA_MemoryInc = DMA_MemoryInc_Enable;
DMA_InitStructure.DMA_PeripheralDataSize = DMA_PeripheralDataSize_Byte;
DMA_InitStructure.DMA_MemoryDataSize = DMA_MemoryDataSize_Byte;
DMA_InitStructure.DMA_Mode = DMA_Mode_Normal;// jeden transfer
DMA_InitStructure.DMA_Priority = DMA_Priority_High; // priorytet DMA
DMA_InitStructure.DMA_M2M = DMA_M2M_Enable; // z pamięci do pamięci
DMA_Init(DMA1_Channel1, &DMA_InitStructure); // zapis konfiguracji
DMA_Cmd(DMA1_Channel1, ENABLE); // włącz DMA = start transferu
```

Program inicjuje transfer z `txt1` do `txt2LCD`, wyświetla, opóźnienie 1 sekunda, inicjuje transfer z `txt2`, wyświetla, opóźnienie ... i tak dalej.

Rozdział 16

Przetwornik analogowo–cyfrowy ADC

Przetwornik analogowo – cyfrowy (*analog-to-digital converter*) ADC zamienia stosunek napięcia mierzonego do napięcia odniesienia U_{ref} (*reference voltage*) na liczbę całkowitą. Napięcie odniesienia jest stałe i generowane albo wewnątrz mikrokontrolera, albo dołączone z zewnątrz. Układ zewnętrzny (odpowiednio wykonany) może zapewnić napięcie o większej dokładności i stabilności krótko– i długookresowej. Przewonik ADC nie może przetworzyć napięcia większego od U_{ref}.

Są trzy kluczowe paramery, które określają przetwornik ADC

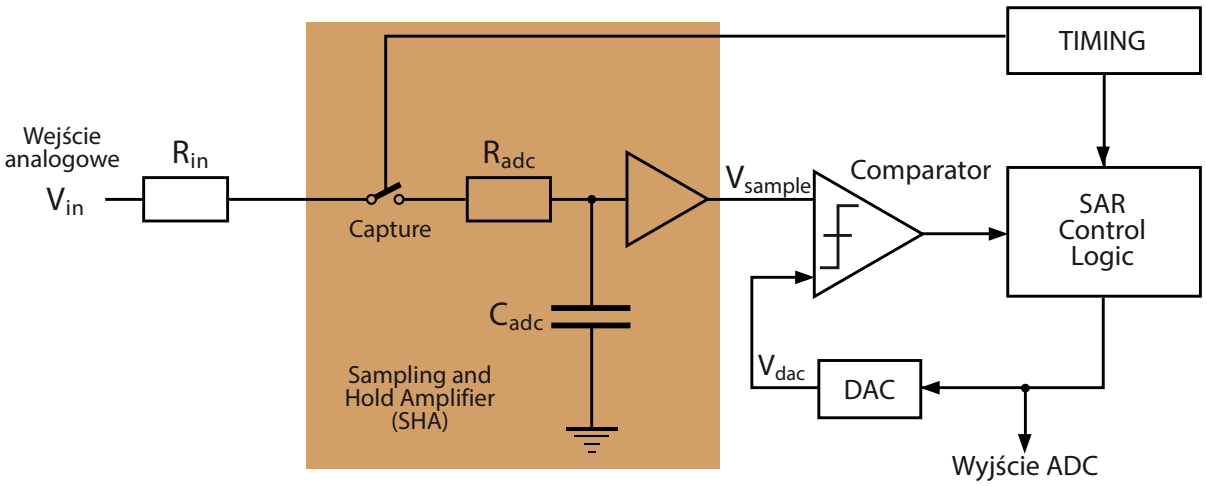
- **Czas przetwarzania**, czyli maksymalny, niezbędny czas do wykonania jednego przetworzenia. Przekłada się na szybkość próbkowania (*sampling rate*), czyli ilość próbek które mogą być przetworzone w jednostce czasu.
- **Rozdzielczość** (*resolution*) to ilość bitów na wyjściu ADC. Możemy także określić rozdzielczość napięciową. Jeżeli przetwornik jest 12-bitowy, to dzieli cały zakres napięcia na $2^{12} = 4096$ równych części. Jeżeli ponadto $U_{ref} = 3.3V$, to rozdzielczość napięciowa, czyli różnica napięć odpowiadająca zmianie na wyjściu przetwornika o jeden bit, jest równa $3.3V/4096 = 0.80566mV$. Rozdzielczości nie należy mylić z dokładnością przetwarzania, na którą składają się rozdzielczość i błędy (na przykład temperaturowe) toru przetwarzania. Dokładność nigdy nie może być lepsza od rozdzielczości.
- **Energia rozpraszana**, czyli tracona w czasie przetwarzania. Parametr szczególnie istotny w urządzeniach zasilanych bateryjnie, jak telefony komórkowe.

16.1. Przetwornik ADC z sukcesywną aproksymacją

Działanie przetwornika składa się z trzech podstawowych etapów. Te etapy to: **próbkowanie**, czyli pobieranie wartości chwilowej sygnału analogowego, **kwantyzacja**, czyli określenie najlepszej cyfrowej reprezentacji pobranej wartości i **kodowanie**, czyli zapisanie wyniku przetwarzania w ustalonym formacie.

W mikrokontrolerach ARM Cortex-M wbudowane są przetworniki 12–bitowe z sukcesywną aproksymacją (*successive-approximation*) SAR. Maksymalna częstotliwość taktowania to 14 MHz. Najkrótszy czas przetwarzania to $1\mu s$, gdy mikrokontroler taktowany jest 56 MHz i $1.17\mu s$ gdy 72 MHz. Schemat przetwornika SAR pokazany jest na rysunku 16.1. Przetwornik składa się z dwóch bloków: układu próbkowania (*Sampling and Hold Amplifier*) SHA oraz układu kwantyzacji.

Przetwornik analogowo-cyfrowy jest drogim układem, dlatego do jego wejścia dołączony jest multiplekser, który umożliwia przetwarzanie sygnałów analogowych pochodzących z różnych źródeł. Multiplekser umożliwia dołączenie 16 kanałów zewnętrznych i dwóch wewnętrznych (czujnik temperatury i napięcie U_{REFINT}) do przetwornika ADC.



Rys. 16.1: Schemat przetwornika ADC z sukcesywną aproksymacją

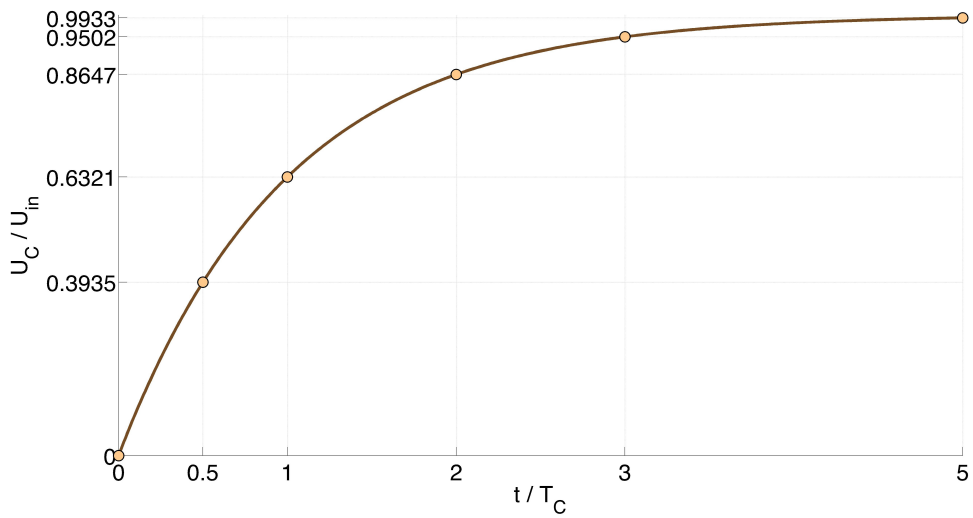
16.1.1. Próbkowanie

Układ próbkowania przetwornika SAR (rysunek 16.1) zawiera opornik wejściowy R_{in}, obwód całkujący $R_{adc}C_{adc}$ i wzmacniacz operacyjny. Zadaniem układu SAR jest pobranie (*sample*) wartości chwilowej napięcia U_{in} i zapamiętanie (*hold*) jej w kondensatorze C_{adc} na czas kwantyzacji. Wzmacniacz operacyjny jest tylko konwerterem impedancji: wysoka impedancja wejścia wzmacniacza nie obciąża układu całkującego i chroni kondensator przed rozładowaniem. Wyjściem układu SAR jest napięcie U_{sample}.

Cykl przetwarzania zaczyna się od zamknięcia (zwarcia) przełącznika *Capture*. Napięcie na kondensatorze C_{adc} rośnie wykładniczo, zgodnie ze wzorem

$$U_C(t) = U_{in} \cdot \left(1 - e^{-\frac{t}{T_C}}\right) \tag{16.1}$$

Gdzie T_C to stała czasowa obwodu całkującego $T_C = (R_{in} + R_{adc}) \cdot C_{adc}$. To oznacza, że napięcie wejściowe nie może być przetwarzane natychmiast. Na przykład jeżeli przełącznik *Capture* jest zamknięty przez czas $3 \cdot T_C$, to napięcie na kondensatorze C_{adc} będzie równe 95.02% napięcia U_{in}, jak pokazano na rysunku 16.2.



Rys. 16.2: Zmiana ilorazu napięć U_{sample}/U_{in} w układzie całkującym

Zgodnie z danymi producenta (dokument *STM32F103x8/B Datasheet* , tablica 46) maksymalne wartości układu całkującego to: $R_{adc} = 1k\Omega$ oraz $C_{adc} = 8pF$. To daje stałą czasową $T_C = 10^3 \cdot 8 \cdot 10^{-12} = 8 \cdot 10^{-9} = 8$ ns. Jeżeli chcemy uzyskać dużą dokładność przetwarzania, to przetwornik należy tak skonfigurować, aby przełącznik *Capture* był zwarty przez dostatecznie długi czas. Okres czasu, w którym przełącznik pozostaje zwarty będziemy nazywać czasem próbkowania (*sampling time*).

Czas przetwarzania będziemy określać w cyklach sygnału `ADCCLK` taktującego przetwornik ADC. Maksymalna częstotliwość tego sygnału (dla procesorów STM32F103) jest równa 14 MHz. ADC dołączony jest do magistrali APB2 (rysunki 15.3, 15.4) o maksymalnej częstotliwości taktowania 72 MHz. Ta częstotliwość może być podzielona w dzielniku układu ADC przez 1, 2, 4, 6 lub 8. To oznacza, że przy maksymalnej częstotliwości magistrali APB2, nie zdołamy uzyskać 14 MHz. Najbliższa niższa częstotliwość (podział przez 6) to 72/6 = 12 MHz. Jeżeli chcemy wykorzystać pełne możliwości przetwornika ADC, to możemy obniżyć częstotliwość taktowania całego systemu do 56 MHz. Wtedy zastosujemy podział przez 4 i uzyskamy 56/4 = 14 MHz. Uzyskaliśmy maksymalną możliwą szybkość przetwarzania, ale kosztem spowolnienia całego systemu.

Nie musimy korzystać z pełnej rozdzielczości przetwornika (12 bitów). W konfiguracji możemy wybrać długość słowa wyjściowego jako 6, 8, 10 lub 12 bitów. Całkowity czas przetwarzania zależy od długości słowa i jest tym mniejszy, im mniejsza jest rozdzielczość (ilość bitów na wyjściu) przetwornika. W przypadku rozdzielczości równej 12 bitów, całkowity czas przetwarzania T jest równy

$$\mathrm{T} = \textit{sampling time} + 12.5 \text{ taktu}$$

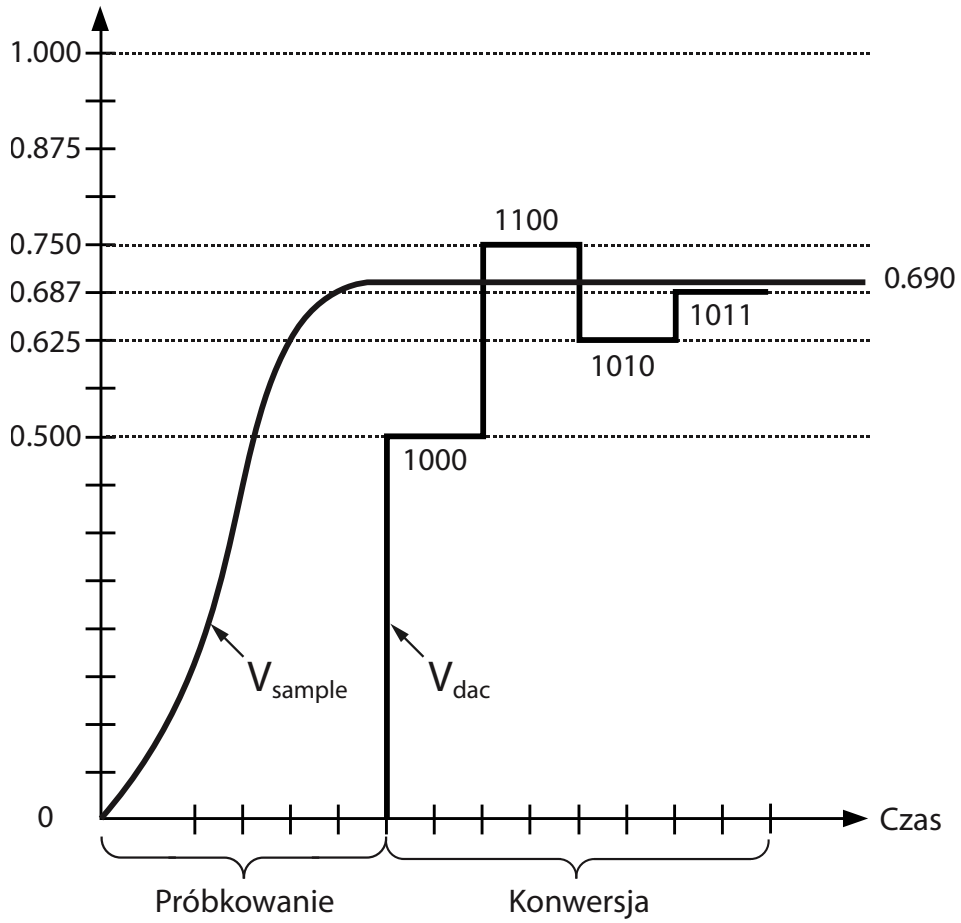
gdzie 12.5 taktu to faza kwantyzacji. Czas próbkowania (*sampling time*) jest wybierany programowo. Do wyboru są wartości równe 1.5, 7.5, 13.5, 28.5, 41.5, 55.5, 71.5 i 239.5 taktów sygnału `ADCCLK`. Każdy kanał może mieć wybrany inny czas próbkowania. Poniżej są dwa przykłady wyliczania czasu T

Przykład 1: `ADCCLK` = 12 MHz, *sampling time* = 1.5 → T = 1.5+12.5 = 14 taktów
14 / 12 000 000 = 1.17 μs

Przykład 2: `ADCCLK` = 14 MHz, *sampling time* = 1.5 → T = 1.5+12.5 = 14 taktów
14 / 14 000 000 = 1 μs

16.1.2. Kwantyzacja

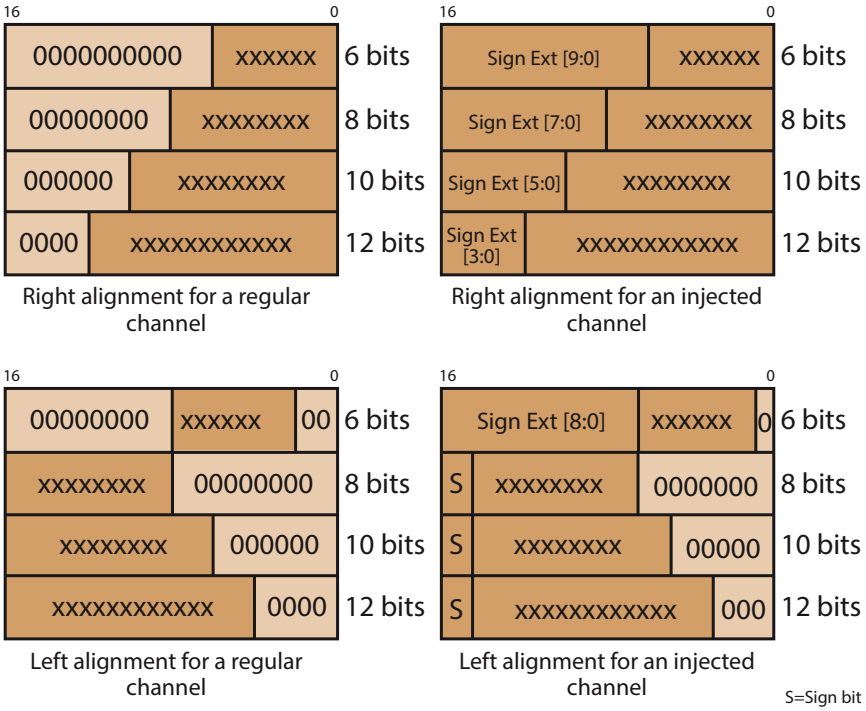
Układ przetwarzania cyfrowego zawiera logikę sterującą, która ustawia/zeruje kolejne bity słowa wyjściowego przetwornika. To słowo steruje wewnętrznym przetwornikiem cyfrowo–analogowym (*digital-to-analog converter*) DAC. Napięcie generowane na wyjściu DAC jest porównywane z napięciem mierzonym przez komparator. W zależności od stanu wyjścia komparatora, układ sterujący zmienia kolejne bity słowa wyjściowego, czyli wyniku przetwarzania. Przetwornik używa algorytmu znanego w optymalizacji jako *dychotomous search* - szukanie z podziałem. Układ sterowania zmienia kolejne bity wyjściowe zaczynając od najstarszego tak, aby napięcie U_{dac} na wyjściu przetwornika cyfrowo–analogowego jak najlepiej odwzorowywało napięcie wejściowe U_{in}.



Rys. 16.3: Przykład 4-bitowego przetwarzania z sukcesywną aproksymacją

1. Przetwarzanie zaczyna się od ustawienia najstarszego bitu słowa wyjściowego, pozostałe bity są wyzerowane. Na wyjściu DAC pojawia się połowa napięcia odniesienia $U_{dac} = U_{ref}/2$.
2. Jeżeli napięcie wejściowe U_{in} jest większe od $U_{ref}/2$ to układ sterujący pozostawia najstarszy bit ustawiony, w przeciwnym razie (gdy $U_{in} < U_{ref}/2$) najstarszy bit jest zerowany. To kończy ustalanie wartości najstarszego bitu.
3. Układ sterujący przechodzi do kolejnego bitu słowa wyjściowego: jest on wstępnie ustawiany. Napięcie na wyjściu DAC jest równe albo $(1/4) \cdot U_{ref}$, albo $(3/4) \cdot U_{ref}$, w zależności od (już ustalonego) stanu najstarszego bitu.
4. Napięcie U_{dac} jest porównywane z napięciem wejściowym i w zależności od wyniku ustalany jest stan bitu.
5. Procedura jest powtarzana dla wszystkich kolejnych bitów słowa wyściowego.

Ilość porównań jest więc równa długości słowa na wyjściu przetwornika. Przykład 4-bitowego przetwarzania z sukcesywną aproksymacją pokazany jest na rysunku 16.3. Bity słowa wyjściowego oznaczymy $b_3b_2b_1b_0$. Zakładamy $U_{ref} = 1$ i $U_{in} = 0.7$. Po fazie próbkowania, gdy nastąpiło ustalenie poziomu napięcia w układzie SHA, rozpoczyna się konwersja. Wstępnie ustawiany jest najstarszy bit słowa wyjściowego $b_3 = 1$. Napięcie na wyjściu DAC jest równe 0.5. Komparator wskazuje, że $U_{in} > 0.5$ dlatego najstarszy bit pozostaje jedynką. Sterowanie ustawia bit $b_2 = 1$. Napięcie na wyjściu DAC jest równe 0.75. Komparator wskazuje, że $U_{in} < 0.75$ dlatego bit b_2 zostaje wyzerowany. Teraz ustawiany jest bit $b_1 = 1$, napięcie na wyjściu DAC $U_{dac} = 0.625$ jest mniejsze od U_{in}, bit b_1 pozostaje



Rys. 16.4: Kodowanie wyników w rejestrach ADC

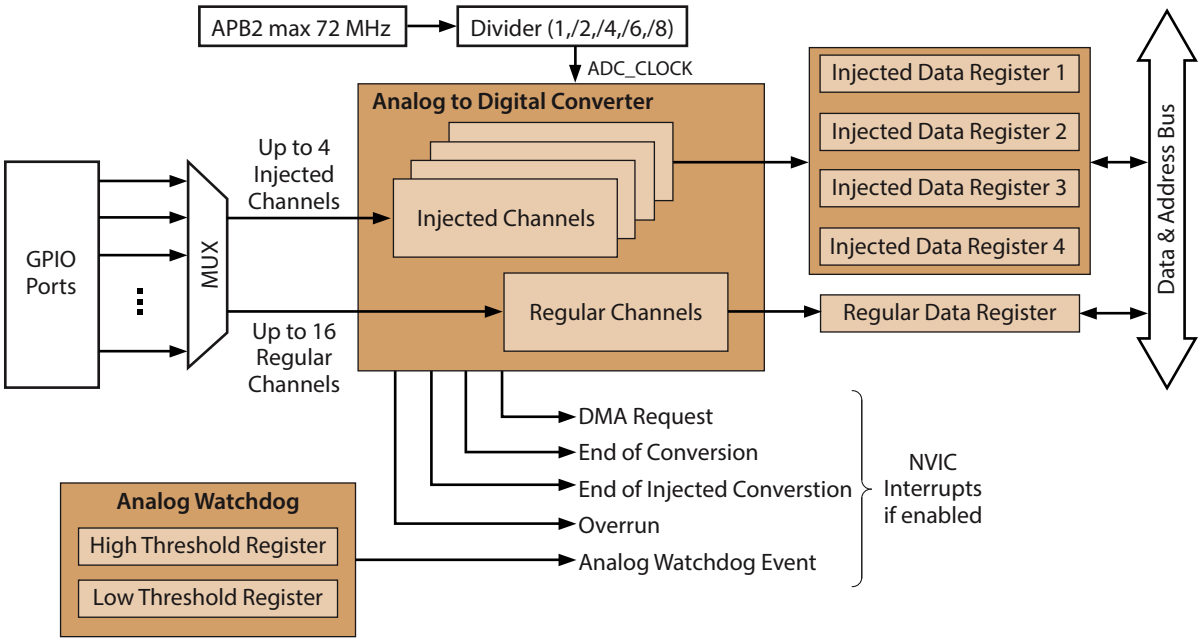
jedynką. Najmłodszy bit $b_0 = 1 \quad \rightarrow \quad U_{dac} = 0.6875 < U_{in}$, czyli b_0 pozostaje jedynką. Wynikiem przetwarzania jest słowo 1011.

16.1.3. Kodowanie

Poszczególne kanały, czyli przetwarzane sygnały można przypisać do jednej z dwóch grup: podstawowej (*regular*) i wstrzykiwanej (*injected*) - rysunek 16.5. Kodowanie wyników jest nieco inne w obu przypadkach. Programowo można zmieniać rozdzielczość przetwornika ADC. Do wyboru są: 6, 8, 10 i 12 bitów. Natomiast rejestry wyjściowe (do odczytu wyniku przetwarzania) są 16–bitowe. Rysunek 16.4 pokazuje różne możliwe formaty danych wyjściowych.

16.2. Kanały wejściowe ADC

Kanały zewnętrzne (16) i wewnętrzne (2), czyli przetwarzane sygnały, można przypisać do jednej z dwóch grup. Te grupy to: **podstawowa** (*regular group*) i **wstrzykiwana** (*injected group*). Do grupy podstawowej można przypisać maksymalnie 16 kanałów, do wstrzykiwanej 4. Grupa składa się z sekwencji przetworzeń, które mogą być wykonywane w dowolnej kolejności. Na przykład możliwe jest przetwarzanie w kolejności: Ch3, Ch8, Ch2, Ch2, Ch0, Ch2, Ch2, Cha15 (przykład zaczerpnięty z dokumentu RM0008). Przetwarzanie grupy wstrzykiwanej ma wyższy priorytet niż podstawowej, dlatego sygnały krytyczne, z punktu widzenia systemu, należy przetwarzać w grupie wstrzykiwanej. Jeżeli wyzwolenie przetwarzania grupy wstrzykiwanej nastąpi w trakcie przetwarzania grupy podstawowej, to będzie ono przerwane. Po zakończeniu przetwarzania grupy wstrzykiwanej, przetwornik powróci do grupy podstawowej.



Rys. 16.5: Schemat przetwornika analogowo - cyfrowego ADC

Rysunek 16.5 zawiera schemat jednego przetwornika ADC. W grupie wstrzykiwanej każdy kanał ma oddzielny rejestr na wynik przetwarzania. Natomiast w grupie podstawowej jest jeden wspólny rejestr dla wszystkich 16 kanałów. Zastosowanie mechanizmu bezpośredniego dostępu do pamięci DMA umożliwia rozdzielenie wyników do poszczególnych kanałów.

16.3. Wyzwalanie ADC i okres próbkowania

W teorii przetwarzania sygnałów (*Digital Signal Processing*) DSP, jednym z najważniejszych parametrów jest okres próbkowania (*sampling time*) T_s. Oznacza to, że próbki muszą być pobierane w równych odstępach czasu. Odwrotność czasu próbkowania T_s, czyli odstępu pomiędzy kolejnymi próbkami, to częstotliwość próbkowania $f_s = 1/T_s$.

Stałą częstotliwość próbkowania zapewnia wyzwalanie przetwarzania za pomocą licznika. Na przykład przetwarzanie w grupie *injected* może być wyzwalane za pomocą licznika `TIM2_CH1`, natomiast w grupie podstawowej za pomocą licznika `TIM1_CH1`. Szczegóły (które kanały liczników umożliwiają jakie wyzwalnie) podane są w dokumencie RM0008.

Dodatkowym elementem układów ADC jest układ monitorujący napięcie w jednym lub kilku kanałach analogowych (*Analog Watchdog*). W przypadku przekroczenia zadanych progów napięcia, zgłasza on zdarzenie, rysunek 16.5.

Mikrokontroler STM32F103VB zawiera dwa układy ADC, a STM32F103ZE – trzy, patrz tabela 13.1. W najprostrzym przypadku ich działanie może być niezależne. Ale dostępny jest także tryb pracy jednoczesnej (*simultaneous mode*) oraz tryb pracy naprzemiennej (*interleaved mode*). W trybie jednoczesnym, przetwornik ADC1 jest nadrzędny w stosunku do ADC2. W momencie wyzwolenia ADC1, oba przetworniki rozpoczynają przetwarzanie jednocześnie. Tryb ten należy stosować przy realizacji regulatora MIMO.

W trybie naprzemiennym, ADC1 jest również nadrzędny w stosunku do ADC2, ale przetwarzanie w ADC2 jest opóźnione w stosunku do ADC1 o 7 (*fast interleaved mode*) lub o 14 (*slow interleaved mode*) taktów sygnału taktującego układy. Umożliwia to podwojenie

częstotliwości próbkowania. Jeżeli w obu przetwornikach ustawimy *sampling time* na 1.5 taktu, to przetwarzanie trwa 14 taktów. Jeżeli teraz, drugi przetwornik rozpoczyna po siedmiu taktach, czyli w połowie czasu przetwarzania pierwszego i oba próbkują ten sam sygnał, to uzyskujemy podwojenie czestotliwości próbkowania. W przypadku `ADCCLK` = 14 MHz możemy uzyskać $f_s = 2$ MHz.

16.4. Autokalibracja

Każdy przetwornik ADC ma wbudowaną procedurę wewnętrznej kalibracji. Kalibracja znacząco redukuje niedokładności przetwarzania wywołane zmianą parametrów elementów wewnętrznych przetwornika. W trakcie kalibracji zapamiętywana jest wartość korekcyjna (liczba), która jest używana do korekcji błędów w każdym następnym cyklu przetwarzania. Autokalibracja jest wyzwalana przez ustawienie bitu `CAL` w rejestrze `ADC_CR2`.

16.5. Przetwarzanie ADC z wykorzystaniem przerwań SysTick

Do projektu regulatora PID potrzebne jest nam najprostrze przetwarzanie analogowo - cyfrowe wyzwalane na początku okresu interwencji regulatora. Cykl pracy regulatora cyfrowego PID skład się z trzech faz:

1. Wystawienie sygnału o wartości obliczonej w poprzednim cyklu na wyjściu regulatora (przetwornik analogowo-cyfrowy DAC)
2. Wyzwolenie przetwarzania analogowo-cyfrowego ADC na wejściu regulatora
3. Po zakończeniu przetwarzania, gdy znana jest reprezentacja cyfrowa wejścia, nastepuje faza obliczeń zakończona zapamiętaniem cyfrowej reprezentacji sygnału wyjściowego – do wystawienia w następnym cyklu interwencji.

Tak działający regulator PID wprowadza opóźnienie równe (dokładnie) długości cyklu interwencji. Do synchronizacji, czyli wyznaczania początków okresów interwencji, najłatwiej jest wykorzystać zegar systemowy *SysTick*. Wykorzystanie przerwań systemowych zapewnia synchronizację pomiędzy wejściem i wyjściem regulatora: wzajemne położenie obu sygnałów nie zależy ani od czasu przetwarzania, ani od czasu trwania obliczeń. Oczywiście, okres interwencji musi być dłuższy od największej możliwej sumy obu tych czasów.

Jeżeli nasz regulator PID będzie współpracować z układem nadrzędnym, to musimy przewidzieć dodatkowy czas na transmisję szeregową. Regulator może wysyłać informacje o alarmach i cyfrowe wartości sygnałów na wejściu i wyjściu (do rejestracji), a otrzymywać wartość zadaną i swoje nastawy.

16.5.1. Projekt21_ADC1

Kopiujemy cały katalog *Projekt15_StepperMotor* i nadajemy mu nazwę *Projekt21_ADC1*. Otwieramy nasz nowy projekt w środowisku Keil μVision. W pliku *stm32f10x_conf.h* usuwamy komentarz z linii

```
#include "stm32f10_adc.h";
```

Z katalogu *Libraries\STM32F10x_StdPeriph_Driver\src* dodajemy plik *stm32f10x_adc.c*

W pliku *main.c* definiujemy zmienną globalną

```
unsigned int ADC1_DR;     // wartość binarna odczytana z ADC
```

W konfiguracji systemu należy dodać inicjalizację ADC i wyzwolenie pierwszego przetwarzania. Odczekamy na koniec przetwarzania i dopiero wtedy załączymy przerwania systemowe.

```
ADC_Config();
ADC_SoftwareStartConvCmd(ADC1, ENABLE);      // rozpocznij przetwarzanie ADC
while (!ADC_GetFlagStatus(ADC1,ADC_FLAG_EOC));  // czekamy na zakończenie
SysTick_Config_Div8(900000ul);                  // przerwanie co 0.1 sekundy
```

W pętli głównej należy wyliczyć i wyświetlić napięcie na podstawie wartości binarnej (zmienna globalna `ADC1_DR`). Pełny zakres binarny jest od zera do $2^{12} - 1 = 4095$, co odpowiada napięciu $3.3V$. Stąd współczynnik proporcjonalności jest równy

$$\frac{3.3}{4095} = 0.00080586$$

```
voltage = ADC1_DR * 0.00080586;         // przeliczone na napięcie
```

Kolejna czynność to konfiguracja układu taktowania systemu. Do funkcji `RCC_Config(void)` należy dopisać

```
// konfiguracja sygnałów taktujacych używanych peryferii
RCC_ADCCLKConfig(RCC_PCLK2_Div6);  // ADCCLK = PCLK2/6 = 72 MHz /6 = 12 MHz
RCC_APB2PeriphClockCmd(RCC_APB2Periph_ADC1, ENABLE);// włącz taktowanie ADC1
RCC_APB2PeriphClockCmd(RCC_APB2Periph_GPIOF, ENABLE); // port GPIO F : LEDy
RCC_APB2PeriphClockCmd(RCC_APB2Periph_GPIOC, ENABLE); //PC4 = potencjometr
```

Port `PC4` musimy zaprogramować jako wejście analogowe, do funkcji `GPIO_Config()` należy dopisać

```
GPIO_InitStructure.GPIO_Pin = GPIO_Pin_4;
GPIO_InitStructure.GPIO_Mode = GPIO_Mode_AIN;     // wejście analogowe
GPIO_InitStructure.GPIO_Speed = GPIO_Speed_50MHz;
GPIO_Init(GPIOC, &GPIO_InitStructure);
```

W pliku *stm32f10x_it.c* napiszemy obsługę przerwania systemowego od SysTick. Najpierw odczytamy wynik przetwarzania zainicjowanego w poprzednim okresie interwencji. Następnie zainicjujemy przetwarzanie w bieżącym okresie interwencji (przyszłego regulatora). Odczytane dane są opóźnione o 0.1 sekundy.

```
/*** @brief This function handles SysTick Handler. ***/
extern unsigned int ADC1_DR;     // wartość binarna odczytana z ADC
void SysTick_Handler(void){
   ADC1_DR = ADC_GetConversionValue(ADC1); // odczyt ADC
   ADC_SoftwareStartConvCmd(ADC1, ENABLE); // rozpocznij przetwarzanie ADC
}
```

Pozostaje napisać funkcję inicjującą przetwornik ADC1 (w pliku *main.c*).

```
// -------------- konfiguracja przetwornika ADC1 --------------
void ADC_Config(void){
   ADC_InitTypeDef ADC_InitStructure;
   ADC_InitStructure.ADC_Mode = ADC_Mode_Independent; // praca niezależna
   ADC_InitStructure.ADC_ScanConvMode = DISABLE; // pomiar jednego kanału
   ADC_InitStructure.ADC_ContinuousConvMode = DISABLE; // tryb jednokrotny
   ADC_InitStructure.ADC_ExternalTrigConv = ADC_ExternalTrigConv_None;
   ADC_InitStructure.ADC_DataAlign = ADC_DataAlign_Right; // wyrównanie
   ADC_InitStructure.ADC_NbrOfChannel = 1; // liczba używanych kanałów =1
   ADC_Init(ADC1, &ADC_InitStructure);    // incjalizacja przetwornika
   ADC_RegularChannelConfig(ADC1,ADC_Channel_14,1,ADC_SampleTime_1Cycles5);

           // kanal 14 - GPIOC4 - potencjometr na płytce
           // grupa podstawowa, czas próbkowania 1,5 + 12,5 = 14 cykli
   ADC_Cmd(ADC1, ENABLE);         // włącz ADC1
   ADC_ResetCalibration(ADC1);    // reset rejestrów kalibracyjnych ADC1
   while(ADC_GetResetCalibrationStatus(ADC1));    // czekamy na reset
   ADC_StartCalibration(ADC1);    // kalibracja ADC1
   while(ADC_GetCalibrationStatus(ADC1)); // czekamy na koniec kalibracji
}
```

16.6. Pętla prądowa 4 - 20 mA

Pętla prądowa 4 -20 mA jest standardem powszechnie stosowanym w przemysłowych przetwornikach pomiarowych. Pozwala na zasilanie oraz jednoczesne przesyłanie informacji na znaczne odległości, z wykorzystaniem przewodu dwużyłowego. Zapewnia dużą dokładność i wysoką odporność na zakłócenia. Opis pętli prądowej zaczniemy od wyjaśnienia sposobu działania źródeł napięciowego i prądowego.

16.6.1. Źródło napięciowe DC: stałe napięcie na zaciskach

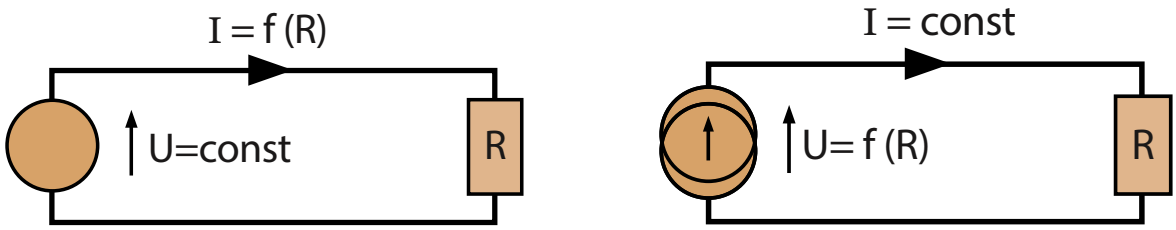
Źródło napięciowe prądu stałego (*Direct Current*) DC utrzymuje stałe napięcie na zaciskach, rysunek 16.6a. Natomiast prąd w zasilanym obwodzie zależy od obciążenia. Przybliżeniem źródła napięciowego jest akumulator samochodowy o napięciu znamionowym 12 Volt.

$$I(R) = \frac{U}{R}, \qquad U = const$$

Moc pobierana ze źródła napięciowego zależy od oporności obciążenia

$$P(R) = U \cdot I(R) = \frac{U^2}{R}$$

Gdy oporność jest nieskończona (źródło rozwarte), to moc pobierana jest zerowa. Gdy oporność jest zerowa (źródło zwarte), to chcemy pobrać nieskończenie wielką moc (prąd rośnie do nieskończoności, gdyż źródło próbuje utrzymać swoje znamionowe napięcie na zaciskach), co najczęściej prowadzi do uszkodzenia źródła.



(a) napięciowe (b) prądowe

Rys. 16.6: Źródło napięciowe i prądowe

16.6.2. Źródło prądowe DC: stały prąd w obwodzie

Źródło prądowe DC wymusza w obwodzie swój prąd znamionowy, rysunek 16.6b. Napięcie na źródle jest zmienne i zależy od obciążenia, czyli oporności obwodu.

$$U(R) = I \cdot R, \qquad I = const$$

Moc pobierana ze źródła prądowego zależy od oporności obciążenia

$$P(R) = U(R) \cdot I = I^2 \cdot R \tag{16.2}$$

Zgodnie z powyższym wzorem, gdy oporność dołączonego obwodu jest zerowa (źródło zwarte), to moc pobierana jest zerowa, gdyż zerowe napięcie wystarcza do wymuszenia ustalonego prądu. Gdy oporność jest nieskończona (źródło rozwarte), to napięcie na źródle rośnie do nieskończoności, gdyż źródło próbuje wymusić zadany prąd w nieistniejącym obwodzie. Wtedy, zgodnie ze wzorem (16.2), próbujemy pobrać nieskończenie wielką moc, co najczęściej prowadzi do uszkodzenia źródła.

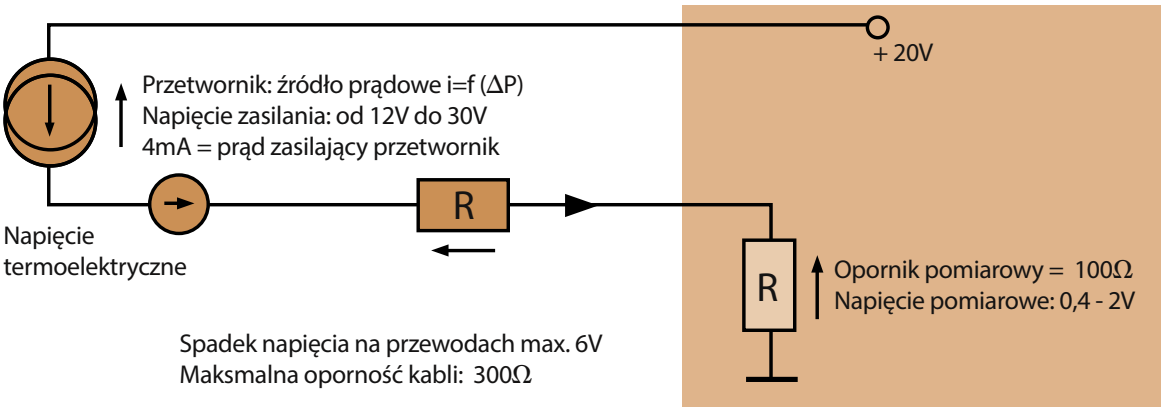
16.6.3. Parametryzowane źródło prądowe

Jest to źródło prądowe DC, którego prąd znamionowy zależy od pewnego parametru, na przykład od zmierzonej wartości temperatury. Jest stosowane na wyjściu przetworników pomiarowych. W przemyśle najczęściej mierzone są: temperatura, ciśnienie absolutne (*absolute pressure*), nadciśnienie (*gauge pressure*) i różnica ciśnień (*differential pressure*).

Rozpatrzmy przetwornik nadciśnienia o zakresie pomiarowym 0 - 600 kPa i wyjściu prądowym 4 - 20 mA. Ciśnieniu 0 kPa odpowiada prąd na wyjściu 4 mA, a ciśnieniu 600 kPa – prąd 20 mA. Zależność prądu i mierzonego ciśnienia jest liniowa. Na podstawie prądu możemy obliczyć ciśnienie zgodnie ze wzorem

$$P[kPa] = 37.5 \cdot I - 150 \tag{16.3}$$

Zgodnie z (16.3), połowa zakresu, czyli prąd 12 mA, odpowiada ciśnieniu 300 kPa. Przekroczenie zakresu 4 - 20 mA jest stanem alarmowym i oznacza przekroczenie zakresu pomiarowego przetwornika: w dół gdy prąd jest mniejszy od 4 mA i w górę gdy większy od 20 mA. W takim przypadku nie obowiązuje wzór (16.3) – sygnalizowany jest tylko fakt przekroczenia.



Rys. 16.7: Przykładowy układ pętli prądowej

16.6.4. Układ pomiarowy pętli prądowej

Układ pomiarowy pętli prądowej pokazano na rysunku 16.7. Cały obwód zasilany jest z niestabilizowanego źródła napięciowego 20V. Przetwornik ciśnienia dołączono do regulatora za pomocą dwóch przewodów. Zgodnie z wymaganiami producenta spadek napięcia na przetworniku powinien być z zakresie od 12 do 30 Volt. To napięcie połączone z prądem w pętli (minimum 4 mA) tworzy energię wykorzystywaną do zasilania przetwornika. W przykładzie pokazanym na rysunku 16.7 minimalna energia dostarczona do przetwornika to $P = 12[V] \cdot 4[mA] = 48[mW]$. Przetwornik zużywa znacznie mniej, dlatego możliwe jest wymuszenie prądu, na przykład 3 mA w celu zasygnalizowania przekroczenia zakresu pomiarowego.

Ważnym elementem pętli prądowej jest opornik pomiarowy. W przykładzie z rysunku 16.7 ma on wartość 100Ω. Zgodnie z prawem Ohma, napięcie generowane na tym oporniku zależy tylko od jego wartości i od prądu w pętli. Nie zależy od napięcia zasilającego, długości przewodów, ani od napięć termoelektrycznych w pętli. Są dwa zjawiska, które mogą istotnie zmienić wartość oporności, czyli pogorszyć dokładność całej pętli. Jedno to temperatura samego opornika, drugie to procesy starzenia. Na temperaturę opornika wpływa temperatura otoczenia oraz ilość energii wydzielanej w wyniku przepływu prądu. Na temperaturę otoczenia raczej nie mamy wpływu (choć oporniki pomiarowe układów o najwyższej dokładności są umieszczane w temostatach). Maksymalna energia wydzielana w oporniku pomiarowym z rysunku 16.7 jest równa: $P_{Rmax} = 2[V] \cdot 20[mA] = 40[mW]$, więc jest dość znaczna. Dlatego należy zastosować opornik o znacznie wiekszej mocy, na przykład 1 [W] i zapewnić dobre odprowadzanie ciepła. Istotnym parametrem jest **temperaturowy współczynnik rezystancji** (*Temperature Coefficient of Resistance*) TCR - określa on zmiany rezystancji pod wpływem temperatury. Czym mniejsza wartość TCR, tym bardziej stabilny rezystor. Temperaturowy współczynnik rezystancji określa wzór:

$$TCR = \frac{\Delta R}{R \cdot \Delta T} \cdot 10^6$$

i jest wyrażony w ppm/°C czyli *parts per milion* na stopień Celsjusza. Typowy opornik ma 200 ppm/°C, opornik pomiarowy to około 3 ppm/°C

Koncentracja nośników prądu jest to ilość nośników przypadająca na jednostkę objętości materiału i jest określana jako gęstość nośników prądu. Różnica koncentracji nośników prądu na złączu dwóch przewodników wywołuje różnicę potencjałów nazywaną napięciem

termoelektrycznym. **Napięcie termoelektryczne** powstaje więc na styku dwóch różnych przewodników (na przykład przewód miedziany wkręcony do stalowych zacisków) i jego wartość zależy od temperatury styku, gdyż temperatura wpływa na gęstość nośników prądu. W oparciu o to zjawisko buduje się termopary, które umożliwiają pomiar wysokich temperatur (np. termopara typu J ma zakres pomiarowy do 1200°C). W pętli prądowej wszelkie napięcia termoelektryczne nie mają wpływu na dokładność przesyłania informacji, gdyż są niwelowane przez źródło prądowe na wyjściu przetwornika.

Podobnie niwelowana jest **oporność przewodów**. W przykładzie pokazanym na rysunku 16.7 napięcie zasilania to 20 Volt. Maksymalne napięcie na oporniku pomiarowym to $U_R = 20[mA] \cdot 100[\Omega] = 2[V]$. Minimalne napięcie zasilania przetwornika to 12 Volt. Jeżeli maksymalną oporność przewodów oznaczymy R_{Pmax}, to prawdziwe jest równanie:

$$20[V] = 12[V] + 2[V] + R_{Pmax}[\Omega] \cdot 0.020[A] \qquad \longrightarrow \qquad R_P < \frac{6[V]}{0.020[A]} = 300[\Omega]$$

Oczywiście należy pozostawić znaczny margines na napięcia termoelektryczne i stany alarmowe (prąd zwiększony ponad 20 mA). Dlatego możemy przyjąć, że oporność przewodów nie powinna przekroczyć 100Ω, co i tak jest bardzo dużą wartością.

Zalety pętli prądowej

- Zasilanie i informacja zwrotna na dwóch przewodach
- Dokładność przesyłania informacji nie zależy od długości (oporności) przewodów ani od napięć termoelektrycznych w pętli
- Duża odporność na zakłócenia elektromagnetyczne dzięki małym impedancjom urządzeń dołączonych na obu końcach przewodów

Źródła błędów

- Błędy przetwornika
- Niestabilność temperaturowa opornika pomiarowego
- Niestabilność długookresowa opornika pomiarowego
- Upływ prądu pomiędzy przewodami spowodowany złą jakością izolacji: na przykład na skutek wchłaniania wilgoci (dotyczy szczególnie kabli doziemnych).

16.7. Projekt22_4-20mA przemysłowy pomiar temperatury

W projekcie wykorzystamy przetwornik temperatury z wyjściem prądowym 4-20 mA, typu MacT II - CT3, produkowany w firmie PLUM w Ignatkach. Dokumentację przetwornika można pobrać ze strony producenta www.plum.pl. Zakładka *Produkty > Gaz*, nastąpi przełączenie na plummac.com, zakładka *Produkty > Gaz > Przetworniki pomiarowe P i T > MacT II*.

Cyfrowy przetwornik temperatury MacT II jest przeznaczony do pomiaru temperatury dowolnych mediów w fazie ciekłej lub gazowej (za wyjątkiem tlenu). Czujnikiem temperatury jest termometr oporowy Pt100 klasy A, podłączony czteroprzewodowo przez złącze Tuchel (czujnik typu CT3 lub CT5). Posiada wyjście prądowe 4 - 20 mA. Przetwornik wyposażony jest także w łącze transmisji szeregowej (standard RS-GAZ2), które umożliwia odczyt mierzonej wartości temperatury oraz konfigurację przetwornika. Wszystkie sygnały wyprowadzone są na złącze HAN4A. Przetwornik należy zasilić napięciem od 9 do 30 V (zalecane 20V).

Rys. 16.8: Przetwornik temperatury MacT II–CT3 z czujnikiem CT5

Zastosujemy opornik drutowy $150\Omega, 3W, \pm 1\%$ o współczynniku temperaturowym 20 ppm/°C. Zakres napięć na oporniku pomiarowym to 0.6 do 3 V. Pętlę zasilimy napięciem 24 V. Układ należy połączyć zgodnie ze schematem 16.7 i tabelą poniżej

płyta	**port**	**kanał ADC1**
ZL27ARM	PB0	IN8
STM3210E-EVAL	PC2	IN10

Pomiary będą wyzwalane w przerwaniu systemowym Systick. Naszym celem jest zbudowanie rejestratora temperatury. Za pomocą kanału DMA wyniki pomiarów będziemy zapisywać w pamięci. Korzystając z transmisji szeregowej, zarejestrowaną temperaturę będziemy mogli przesłać do systemu zbierania danych (rozdział 18).

Przetwornik wbudowany do mikrokontrolera wykazuje dość znaczny szum pomiarowy (masy analogowa i cyfrowa nie są rozdzielone). Ten szum objawia się jako denerwujące, częste zmiany wskazań. Dlatego zadeklarujemy bufor na sto odczytów z przetwornika ADC. Pomiary będą wyzwalane co 0.1 sekundy, czyli bufor obejmie 10 sekund. Kanał DMA zaprogramujemy tak, aby po zakończeniu każdego przetwarzania wynik był wpisywany do kolejnej komórki pamięci w buforze. W pętli głównej będziemy wyliczać średnią z tych stu pomiarów, którą przeliczymy na prąd, a następnie na temperaturę. W pliku *main.c* zaczniemy od deklaracji zmiennych globalnych.

```
unsigned char txt2LCD[32] = 0;     // bufor funkcji sprintf
uint16_t PC2_adc[100]=0;           // bufor na wartości binarne z ADC
```

Następnie dołączymy prototyp funkcji konfigurującej DMA, którą wywołamy na początku funkcji *main()*. Nie możemy zapomnieć o załączeniu taktowania układu DMA

w *RCC_Config*. W pętli głównej wyliczymy średnią ze stu odczytów, ale żeby nie utracić dokładności, obliczenia będą wykonywane na liczbach *float*.

```
current =0.0;
for(i=0; i<100; i++){
  current += (float)PC2_adc[i] * 0.00537241;  // przeliczone na prąd [mA]
}
current /= 100.0;
T = 5.625*current - 52.5;
```

Napięcie referencyjne $U_{ref} = 3.3V$ odpowiada wartości binarnej `0x0FFF` = 4095. Jeżeli pomnożymy liczbę binarną odczytaną z ADC przez 3.3/4095, to uzyskamy napięcie na opornku pomiarowym, wyrażone w Voltach. Prąd płynący przez ten opornik ($R = 150\Omega$) jest więc równy $i[mA] = 1000 \cdot U[V]/150[\Omega]$. Ostatecznie, wartość binarną odczytaną z ADC należy pomnożyć przez

$$(3.3/4095) \cdot 1000/150 = 0.00537241$$

Otrzymamy prąd wyrażony w miliAmperach. Jeżeli włączymy precyzyjny amperomierz w pętlę prądową, to porównując wskazania, można skorygować ten współczynnik tak, aby uzyskać jak najmniejszą rozbieżność. Zakres pomiarowy przetwornika temperatury (-30°C , +60°C) prowadzi do wzoru $T = 5.625 \cdot i - 52.5$. Układ DMA zaprogramujemy na cykliczny transfer stu słów 16–bitowych z ADC do bufora w pamięci

```
// ----------------- konfiguracja kanału DMA ------------------
void DMA_Config(void){
   #define ADC1_DR_Address 0x4001244C;    //adres rejestru ADC1->DR
   DMA_InitTypeDef DMA_InitStructure;
   DMA_DeInit(DMA1_Channel1);     // usuń poprzednią konfigurację DMA
   DMA_InitStructure.DMA_PeripheralBaseAddr = (uint32_t) ADC1_DR_Address;
   DMA_InitStructure.DMA_MemoryBaseAddr = (uint32_t) &PC2_adc;
   DMA_InitStructure.DMA_DIR = DMA_DIR_PeripheralSRC; // kierunek transferu
   DMA_InitStructure.DMA_BufferSize = 100; // ilość elementów do przesłania
   DMA_InitStructure.DMA_PeripheralInc = DMA_PeripheralInc_Disable;
   DMA_InitStructure.DMA_MemoryInc = DMA_MemoryInc_Enable;
   DMA_InitStructure.DMA_PeripheralDataSize=DMA_PeripheralDataSize_HalfWord;
   DMA_InitStructure.DMA_MemoryDataSize = DMA_MemoryDataSize_HalfWord;
   DMA_InitStructure.DMA_Mode = DMA_Mode_Circular;     // cyklicznie
   DMA_InitStructure.DMA_Priority = DMA_Priority_High; // priorytet DMA
   DMA_InitStructure.DMA_M2M = DMA_M2M_Disable; // z portu do pamięci
   DMA_Init(DMA1_Channel1, &DMA_InitStructure); // zapis konfiguracji
   DMA_Cmd(DMA1_Channel1, ENABLE);             // włącz kanał 1 DMA1
}
```

Funkcja inicjalizacji przetwornika ADC1 *ADC_Config* pozostaje bez zmian, jedynie na jej końcu należy dodać instrukcję, która włączy DMA dla ADC1:

```
   DMA_Cmd(DMA1_Channel1, ENABLE);)
```

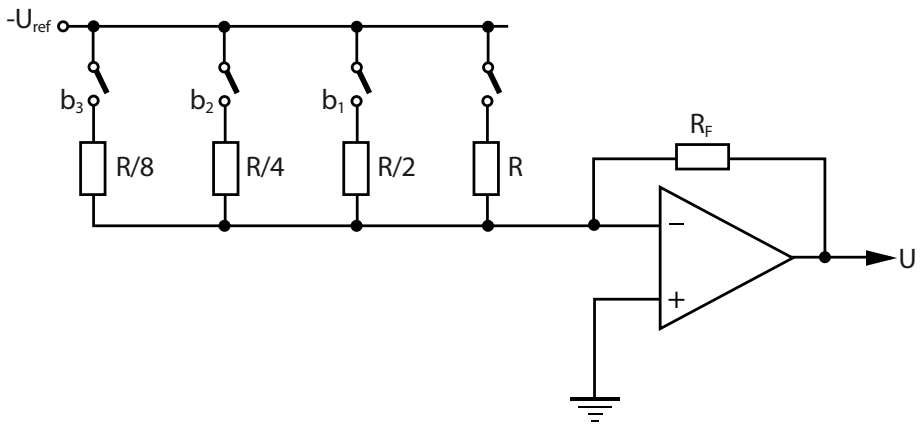
W obsłudze przerwania od SysTick należy wpisać:

```
   ADC_SoftwareStartConvCmd(ADC1, ENABLE); // rozpocznij przetwarzanie ADC1
```

Rozdział 17

Przetwornik cyfrowo–analogowy DAC

Przetwornik cyfrowo–analogowy (*Digital-to-Analog Converter*) DAC zamienia liczbę całkowitą na napięcie analogowe. Na przykład odtwarzacze MP3 wykorzystują przetworniki DAC do zamiany zapisu cyfrowego muzyki na sygnał analogowy przesyłany do słuchawek.



Rys. 17.1: Schemat 4–bitowego przetwornika DAC

17.1. Budowa przetwornika DAC

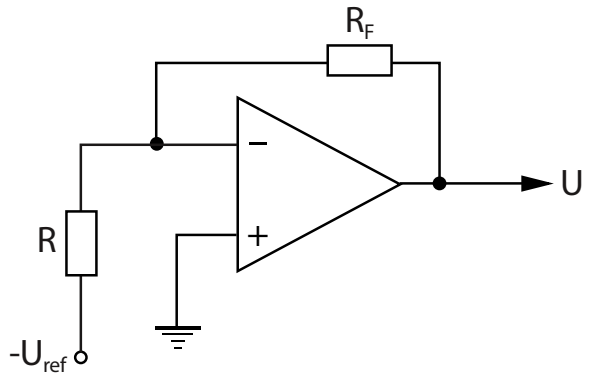
Rysunek 17.1 pokazuje budowę 4–bitowego przetwornika DAC. Liczbę binarną przeznaczoną do przetworzenia oznaczymy $b_3b_2b_1b_0$. Jeżeli bit b_i jest zerem, to odpowiadający mu przełącznik jest otwarty, gdy jest jedynką – przełącznik jest zamknięty (zwarty). Napiszemy równania dla uproszczonego układu, pokazanego na rysunku 17.2. Zakładamy, że wzmacniacz operacyjny ma nieskończone (bardzo duże) wzmocnienie, które oznaczymy jako K. Napięcie na wyjściu U otrzymamy mnożąc napięcie na wejściu odwracającym przez minus K. Napięcie odniesienia U_{ref} jest ujemne.

$$U = -K\Big[(U + U_{ref})\frac{R}{R + R_F} - U_{ref}\Big]$$

Obie strony dzielimy przez K. Zakładając, że $K \to \infty$, wyrazy z K w mianowniku możemy pominąć

$$\frac{U}{K} = -\Big[(U + U_{ref})\frac{R}{R + R_F} - U_{ref}\Big] \quad \to \quad (U + U_{ref})\frac{R}{R + R_F} = U_{ref}$$

$$(U + U_{ref})R = U_{ref}(R + R_F) \quad \to \quad UR + U_{ref}R = U_{ref}R + U_{ref}R_F \quad \to \quad U = \frac{R_F}{R}U_{ref}$$



Rys. 17.2: Dzielnik napięcia na wzmacniaczu operacyjnym

Układ z rysunku 17.2 jest więc dzielnikiem rezystancyjnym napięcia odniesienia U_{ref}. Jeżeli do opornika R dołączymy równolegle opornik $R/2$, to uzyskamy

$$R||R/2 = \frac{R \cdot R/2}{R + R/2} = \frac{R}{3} \quad \rightarrow \quad U = \frac{R_F}{R/3} U_{ref} = 3\frac{R_F}{R} U_{ref} = (1+2)\frac{R_F}{R} U_{ref}$$

Widzimy, że opornik $R/2$ ma wagę 2. Analogicznie opornik $R/4$ ma wagę 4 i $R/8$ ma wagę 8. Możemy teraz opisać działanie układu z rysunku 17.1 wzorem

$$U = U_{ref}\frac{R_F}{R}\left[8b_3 + 4b_2 + 2b_1 + b_0\right] = U_{ref}\frac{R_F}{R}\left[2^3 b_3 + 2^2 b_2 + 2^1 b_1 + 2^0 b_0\right]$$

Jak widać napięcie U na wyjściu przetwornika DAC liniowo zależy od wartości $b_3 b_2 b_1 b_0$.

Ponieważ wartość kolejnego rezystora podwaja się z każdym kolejnym bitem, to powiększenie rozdzielczości wymaga poszerzenia zakresu wartości rezystancji. Na przykład, dla przetwornika 12–bitowego, stosunek skrajnych rezystancji jest jak 1:2048, z tolerancją wykonania 0.025%. Ponadto konieczne jest zapewnienie jednakowych współczynników temperaturowych TCR wszystkich oporników, co przy takiej rozpiętości rezystancji jest zadaniem bardzo trudnym. Co więcej, stosowanie dużych rezystorów pogarsza własności szumowe. Dlatego w praktyce stosuje się drabinki w układzie $R - 2R$, jak pokazano na rysunku 17.3. Jak łatwo zauważyć, w układzie z drabinką $R - 2R$ potrzebne są dwa precyzyjne rezystory na każdy przetwarzany bit. Rozpiętość ich wartości jest niewielka. Powierzchnia układu nie rośnie wykładniczo z ilością bitów. Te wszystkie zalety powodują, że w przetwornikach DAC najczęściej stosowane są drabinki $R - 2R$.

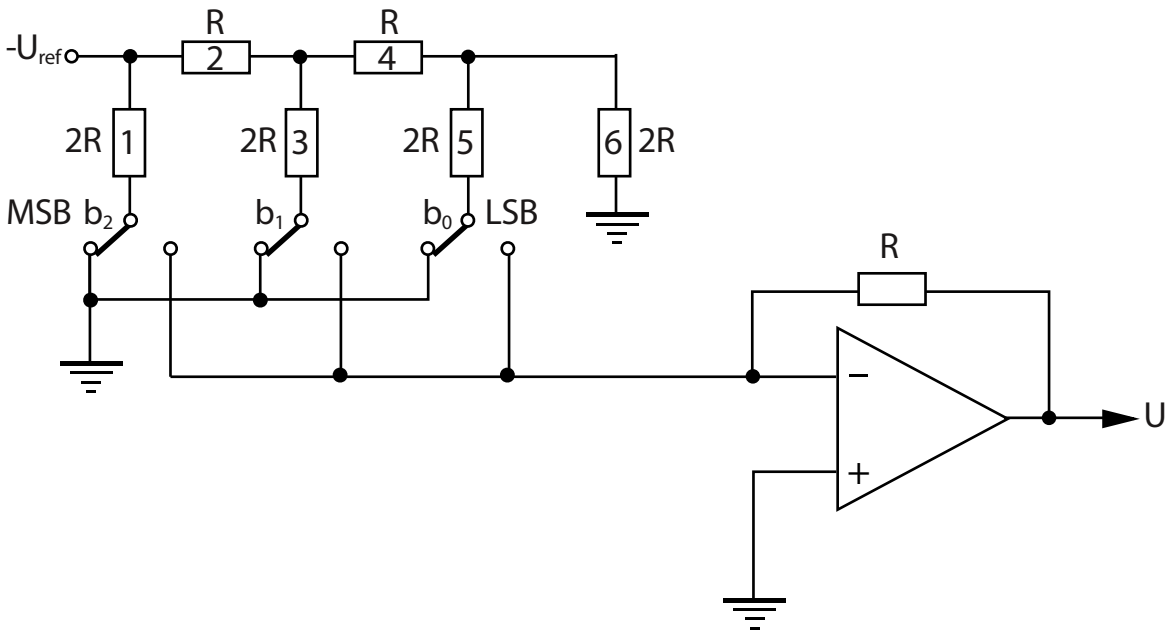
Oporniki na rysunku 17.3 zostały ponumerowane, co ułatwi nam analizę. Wzmacniacz operacyjny ma nieskończone wzmocnienie. Wejście nieodwracające (+) jest uziemione. Wzmacniane jest napięcie różnicowe, czyli napięcie na wyjściu wzmacniacza jest równe $U = K \cdot (U_{IN+} - U_{IN-}) = -K \cdot U_{IN-}$. Gdy obie strony tej równości podzielimy przez K, to pamiętając, że $K \rightarrow \infty$ uzyskamy $U_{IN-} \rightarrow 0$. W równaniach opisujących napięcie wyjściowe będziemy więc zakładać, że $U_{IN-} = 0$.

Zaczniemy od kodu wejściowego $b_2 b_1 b_0 = 100$. Układ połączeń pokazany jest na rysunku 17.4. Oporność zastępczą oporników $2, 3, 4, 5, 6$ możemy obliczyć następująco

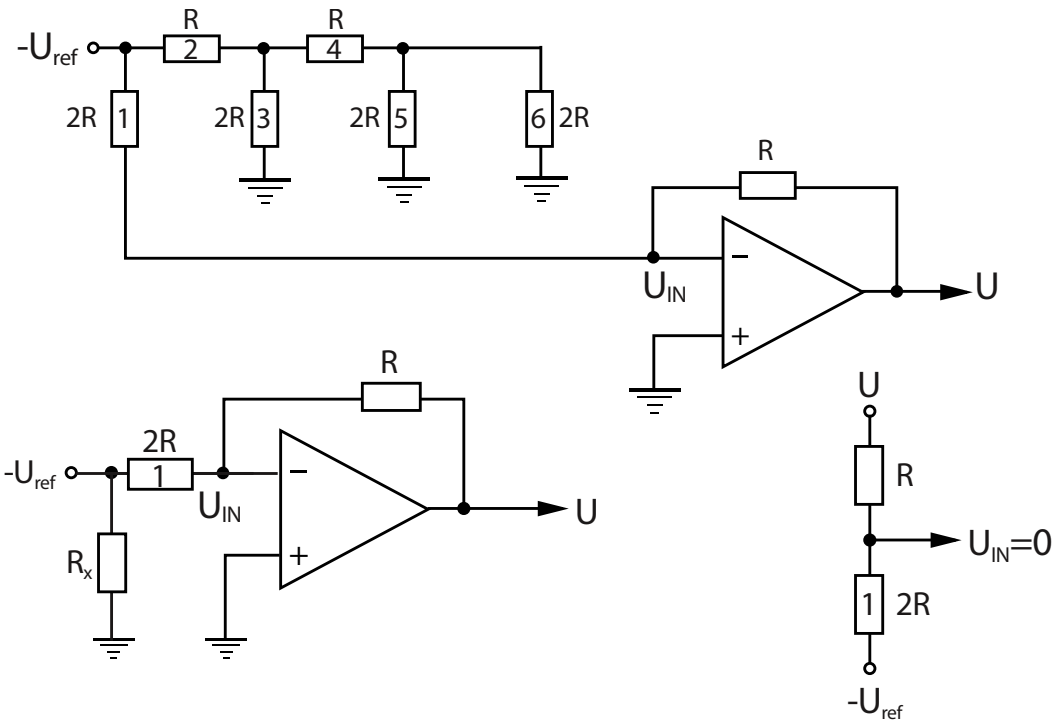
$$5||6 = R \quad \rightarrow \quad 5||6+4 = 2R \quad \rightarrow \quad (5||6+4)||3 = R \quad \rightarrow \quad R_x = (5||6+4)||3+2 = 2R$$

Jednak oporność R_x tylko obciąża źródło napięcia odniesienia U_{ref} i nie ma wpływu na napięcie wyjściowe przetwornika DAC. Schemat z rysunku 17.4 możemy opisać równaniem

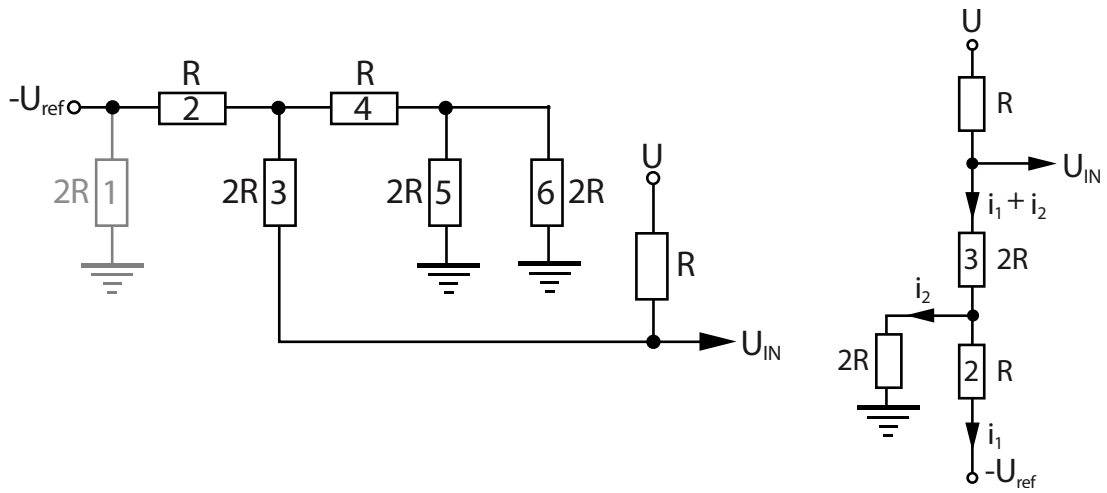
$$U_{IN-} = 0 = -U_{ref} + (U + U_{ref})\frac{2R}{3R} \quad \rightarrow \quad -3U_{ref} + 2U + 2U_{ref} = 0 \quad \rightarrow \quad U = \frac{1}{2}U_{ref}$$



Rys. 17.3: Schemat 3–bitowego przetwornika DAC z drabinką R – 2R



Rys. 17.4: Przetwornik DAC z drabinką R – 2R, wejście $b_2b_1b_0 = 100$



Rys. 17.5: Przetwornik DAC z drabinką R – 2R, wejście $b_2b_1b_0 = 010$

Widzimy, że waga bitu b_2 jest równa $0.5 = 2^{-1}$. Z powyższych równań płynie bardzo ważny wniosek: nastąpiło uproszczenie $2R/3R = 2/3$. To oznacza, że ważny jest jedynie stosunek rezystancji, a nie ich wartości bezwzględne. Tak będzie we wszystkich następnych wzorach. Na parametry przetwarzania (zakres, rozdzielczość, błąd przetwarzania) ma wpływ jedynie dokładność zachowania stosunku wartości pary 2R i R, a nie konkretne wartości – a więc dwa układy przetworników mogą mieć zupełnie inne wartości R np. w jednym $R = 10.00k\Omega$, a w drugim $R = 10.50k\Omega$, ale jeśli spełniony jest żądany stosunek $2R/R = 2$, to obydwa przetworniki mają identyczne parametry przetwarzania. Właściwość ta jest bardzo korzystna dla technologii układów scalonych, gdyż wykonanie identycznych wartości rezystancji w jednej strukturze jest łatwe, ale wykonanie rezystorów o identycznych wartościach w kilku różnych egzemplarzach jest już kłopotliwe.

Sprawdzimy teraz wagę bitu b_1. W tym celu kod wejściowy zamienimy na $b_2b_1b_0 = 010$. Schemat układu pokazany jest na rysunku 17.5. Oporniki $4, 5, 6$ zastąpimy jednym o oporności $5||6 + 4 = 2R$. Opornik 1 nie ma wpływu na U, więc go pominiemy. Równania opisujące rozpływ prądów:

$$U = R(i_1 + i_2) \tag{17.1}$$

$$U_{ref} + 2Ri_2 = Ri_1 \tag{17.2}$$

$$U_{ref} = 2R(i_1 + i_2) + Ri_1 = 3Ri_1 + 2Ri_2 \tag{17.3}$$

Równanie (17.2) wstawimy do (17.3)

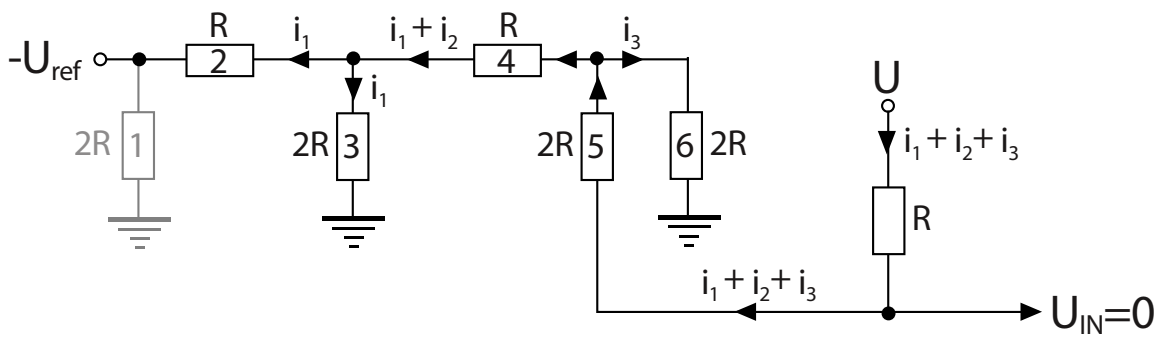
$$U_{ref} = 3(U_{ref} + 2Ri_2) + 2Ri_2 = 3U_{ref} + 8Ri_2 \quad \rightarrow \quad Ri_2 = -\frac{U_{ref}}{4} \tag{17.4}$$

Równanie (17.4) wstawimy do (17.2)

$$Ri_1 = U_{ref} + 2\left(-\frac{U_{ref}}{4}\right) = U_{ref}\left(1 - \frac{1}{2}\right) = \frac{U_{ref}}{2} \tag{17.5}$$

Na końcu, równania (17.4) i (17.5) wstawimy do (17.1)

$$U = Ri_1 + Ri_2 = \frac{U_{ref}}{2} - \frac{U_{ref}}{4} = \frac{U_{ref}}{4}$$



Rys. 17.6: Przetwornik DAC z drabinką R – 2R, wejście $b_2b_1b_0 = 001$

Zgodnie z oczekiwaniem, waga bitu b_1 jest równa 2^{-2}. Pozostał nam bit b_0. Kod wejściowy: $b_2b_1b_0 = 001$. Schemat układu pokazany jest na rysunku 17.6. Opornik 1 nie ma wpływu na U, więc go pominiemy. Równania opisujące rozpływ prądów:

$$U = R(i_1 + i_2 + i_3) \tag{17.6}$$

$$2Ri_2 = Ri_1 - U_{ref} \quad \rightarrow \quad Ri_1 = U_{ref} + 2Ri_2 \tag{17.7}$$

$$U_{IN} = 0 = 2Ri_3 + 2R(i_1 + i_2 + i_3) = 2Ri_1 + 2Ri_2 + 4Ri_3 \quad \rightarrow \quad i_3 = \frac{-i_1 - i_2}{2} \tag{17.8}$$

$$2Ri_3 = R(i_1 + i_2) + Ri_1 - U_{ref} \quad \rightarrow \quad U_{ref} - 2Ri_1 - Ri_2 + 2Ri_3 = 0 \tag{17.9}$$

Równanie (17.8) wstawimy do (17.9)

$$U_{ref} - 2Ri_1 - Ri_2 + 2R\frac{-i_1 - i_2}{2} = U_{ref} - 3Ri_1 - 2Ri_2 = 0 \tag{17.10}$$

Teraz do (17.10) wstawimy (17.7)

$$U_{ref} - 3(U_{ref} + 2Ri_2) - 2Ri_2 = -2U_{ref} - 8Ri_2 = 0 \quad \rightarrow \quad Ri_2 = -\frac{U_{ref}}{4} \tag{17.11}$$

(17.11) wstawimy do (17.7)

$$Ri_1 = U_{ref} + 2Ri_2 = U_{ref} - 2\frac{U_{ref}}{4} = \frac{U_{ref}}{2} \tag{17.12}$$

Wyliczymy Ri_3 korzystając z (17.8), (17.11) i (17.12)

$$Ri_3 = R\frac{-i_1 - i_2}{2} = -\frac{R}{2}\left(\frac{U_{ref}}{2} - \frac{U_{ref}}{4}\right) = -\frac{U_{ref}}{8} \tag{17.13}$$

Końcowe podstawienie, to (17.11), (17.12) i (17.13) do (17.6)

$$U = Ri_1 + Ri_2 + Ri_3 = U_{ref}\left(\frac{1}{2} - \frac{1}{4} - \frac{1}{8}\right) = \frac{U_{ref}}{8} \tag{17.14}$$

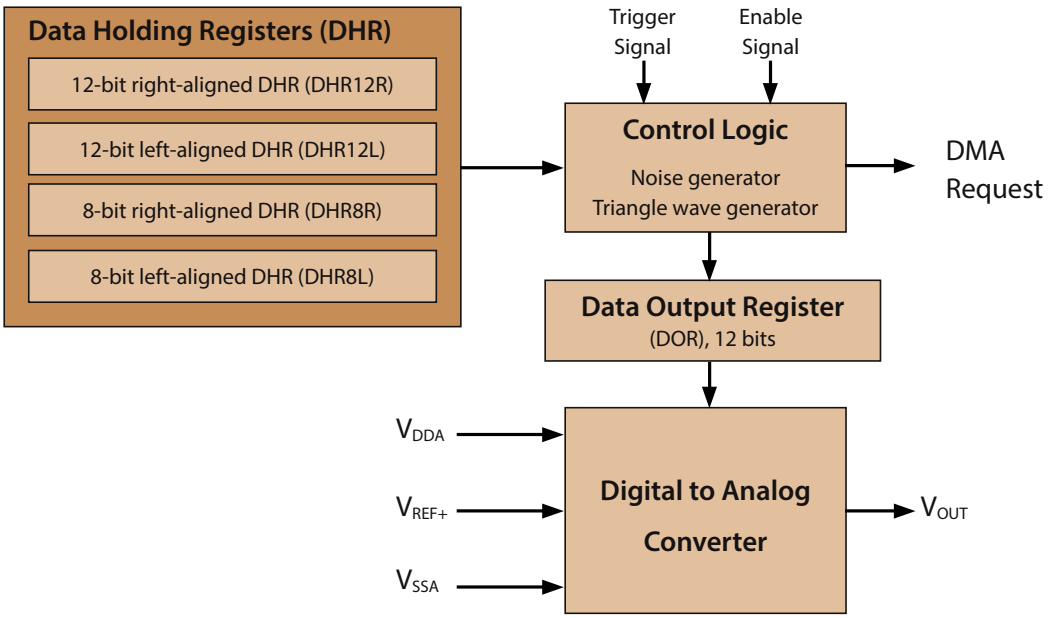
Zgodnie z oczekiwaniem, waga bitu b_0 jest równa 2^{-3}. Możemy teraz utworzyć wzór opisujący działanie przetwornika DAC z rysunku 17.3.

$$U = U_{ref}(b_2 2^{-1} + b_1 2^{-2} + b_0 2^{-3}) = U_{ref} 2^{-3}(b_2 2^2 + b_1 2^1 + b_0 2^0)$$

Tak jak poprzednio, napięcie U na wyjściu przetwornika DAC zależy liniowo od $b_2b_1b_0$.

Najważniejsze cechy przetwornika DAC to: rozdzielczość, czas ustalania odpowiedzi i zakłócenia dynamiczne.

- **Rozdzielczość** (*resolution*) to najmniejsza zmiana napięcia, jaką może odwzorować przetwornik. Jeżeli napięcie referencyjne $U_{ref} = 3.3V$ i przetwornik jest 12–bitowy, to rozdzielczość napięciowa jest równa $\Delta U = 3.3/4095$ (liczymy od $0V$ do $3.3V \leftrightarrow 4095 =$ `0xFFF`), czyli $\Delta U = 8.0586mV$. Najczęściej rozdzielczość wyraża się jako ilość przetwarzanych bitów. Przetworniki DAC w mikrokontrolerach Cortex-M3 są 12–bitowe.
- **Czas ustalania odpowiedzi** (*setting time*) jest to czas od zapisania liczby do rejestru DAC, do ustalenia się napięcia wyjściowego z dokładnością do rozdzielczości napięciowej, czyli wartości napięcia odpowiadającej LSB. W przypadku przetwornika 12–bitowego jest to 0.025% zakresu. Wartość tego błędu jest nazywana *error band*. Czas ustalania jest ograniczony przez pojemności rozproszone w układzie DAC, głównie są to pojemności przełączników.
- **Zakłócenia dynamiczne** (*glitches*), zwane popularnie *gliczami*. W sytuacji idealnej, gdy następuje zmiana przetwarzanej liczby, napięcie wyjściowe powinno monotonicznie dążyć do nowej wartości. W rzeczywistości przełączniki działają z różnymi prędkościami i może nastąpić krótkotrwały stan nieustalony, gdy część przełączników osiągnęła nowy stan, a inne pozostają w poprzednim stanie. To powoduje krótkotrwaly skok napięcia wyjściowego do wartości, która nie odpowiada ani poprzedniej ani nowej wartości przetwarzanej. Takie krótkotrwałe zmiany napięcia są nazywane gliczami. Glicze wystąpiły już w wyniku odbić styków, gdy podłączaliśmy klawiaturę do płyty ZL27ARM.



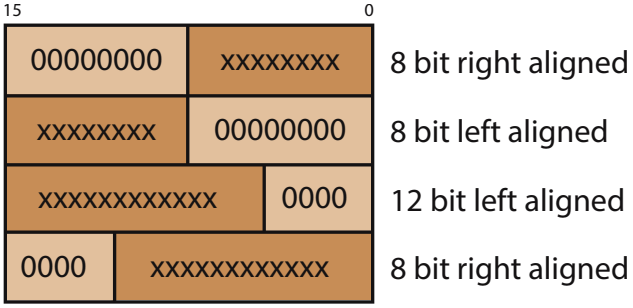
Rys. 17.7: Przetwornik cyfrowo–analogowy DAC

17.2. DAC w procesorach STM32F103

Zgodnie z tabelą 13.1, procesor STM32F103VB nie zawiera przetworników DAC. Procesor STM32F103ZE ma wbudowane dwa przetworniki DAC, każdy ma jedno wyjście: DAC1 jest wyprowadzony na port PA4, DAC2 na port PA5. Dlatego wszystkie projekty wymagające przetwarzania A/D bedą realizowane na płycie STM3210E-EVAL.

Każdy DAC może być skonfigurowany jako 8–bitowy, albo 12–bitowy. Mogą pracować niezależnie lub synchronicznie. Praca synchroniczna jest konieczna, gdy na przykład chcemy zrealizować odtwarzacz muzyki stereo. Jak pokazano na rysunku 17.7 układ DAC zawiera rejestry danych (*Data Holding Registers*) DHR, logikę sterującą (*Control Logic*), rejestr wyjściowy (*Data Output Register*) DOR i przetwornik DAC. Zasilany jest z napięcia części analogowej V_{DDA} i masy analogowej V_{SSA}. Wymaga ponadto napięcia odniesienia V_{REF}. W procesorach Cortex-M stosuje się jedno, wspólne napięcie odniesienia V_{REF} dla wszystkich układów ADC i DAC. Wewnętrzne źródło ma wartość $V_{REF} = 3.3V$. Zakres wyjścia przetwornika DAC, to od $0V$ do V_{REF}.

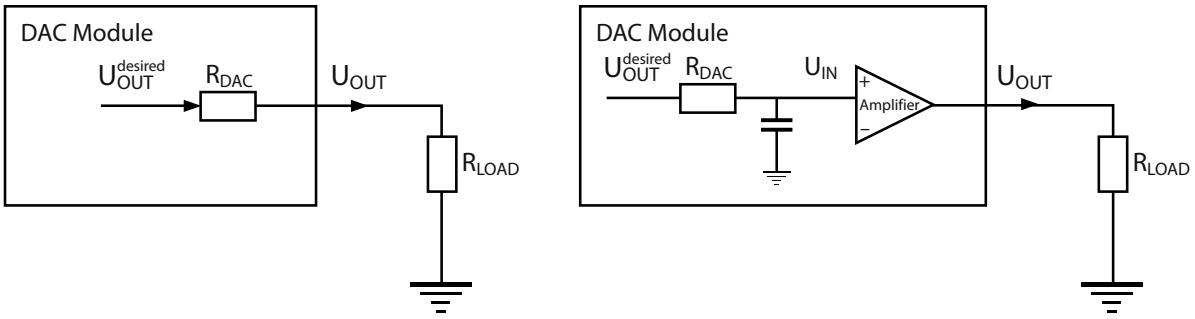
Logika sterująca może dodać szum biały lub przebieg piłozębny do wyjścia przetwornika DAC. Formaty danych w rejestrach DHR pokazane są na rysunku 17.8.



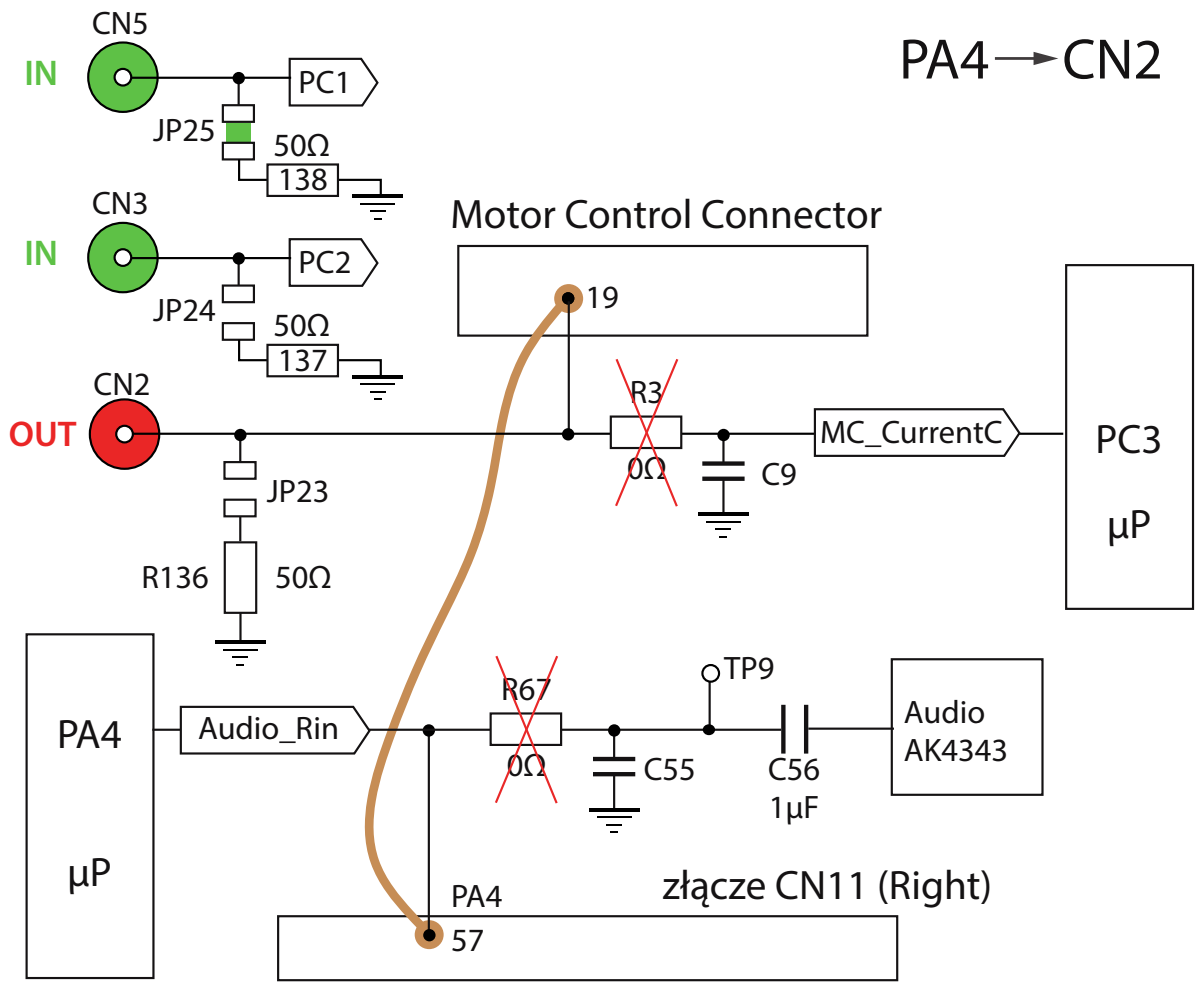
Rys. 17.8: Formaty danych w rejestrach DHR

17.3. Wzmacniacz wyjściowy

Do wyjścia przetwornika DAC można (programowo) dołączyć wzmacniacz, który zwiększa wydajność prądową układu. Obie konfiguracje wyjścia DAC pokazane są na rysunku 17.9.



Rys. 17.9: Wzmacniacz wyjściowy przetwornika DAC



Rys. 17.10: Modyfikacja płyty STM3210E-EVAL

Wzmacniacz wyjściowy jest dołączany przez wyzerowanie bitu `DAC_CR_BOFF` w rejestrze sterującym `DAC_CR`. Oporność wyjściowa przetwornika DAC, gdy wyłączony jest wzmacniacz, jest równa $R_{DAC} = 15\ k\Omega$ - lewa strona rysunku 17.9. To oznacza, że z obciążeniem wyjścia R_{LOAD} tworzy ona dzielnik potencjometryczny napięcia. Jeżeli dołączymy $R_{LOAD} = 15\ k\Omega$, na wyjściu uzyskamy tylko połowę napięcia. Dopiero obciążenie $R_{LOAD} = 1.5\ M\Omega$ powoduje błąd 1%. Dlatego w praktyce, zawsze należy dołączać wzmacniacz wyjściowy przez wyzerowanie bitu `DAC_CR_BOFF` - prawa strona rysunku 17.9. Z dołączonym wzmacniaczem, zakres napięć wyjściowych jest równy od $0.2\ V$ do $V_{DDA} - 0.2\ V$ i jest najszerszy z możliwych do osiągnięcia.

17.4. Projekt23_DAC_sin

W płycie STM3210E-EVAL zainstalowano trzy gniazda koncentryczne BNC. Wszystkie są wejściami przetworników ADC. Wykonamy proste modyfikacje (rysunek 17.10). Wylutujemy dwie zworki (oporniki 0 Ω): R3 i R67. W ten sposób przerwiemy połączenie gniazda CN2 z portem PC3 oraz gniazda PA4 z układem *audio*. Ponadto należy zewrzeć *jumper* JP25, w ten sposób dołączymy terminal (zakończenie linji długiej) R138 50 Ω do gniazda CN5. Pin 19 złącza *Motor Control Connector* należy połączyć przewodem z pinem 57 (PA4) złącza CN11. Na tak zmodyfikiwanej płycie zbudujemy obiekt regulacji

oraz regulator PID. Gniazdo CN3 to będzie wejście, a CN2 to wyjście obiektu/regulatora. Przez gniazdo CN5 będziemy dołączali zakłócenia, na przykład z generatora szumów. Gdy nie dołączymy nic, opornik R138 50 Ω wymusi zero, czyli brak zakłóceń.

W bieżącym projekcie wykonamy konfigurację przetwornika DAC i napiszemy program generacji sygnału $CN2 = sin(t)$. Synchronizację podstawy czasu zapewni przerwanie systemowe SysTick.

```
// --------- konfiguracja przetwornika DAC channel1 -----------
void DAC_Config(void){
   DAC_InitTypeDef DAC_InitStructure;
   DAC_InitStructure.DAC_Trigger = DAC_Trigger_None;
   DAC_InitStructure.DAC_WaveGeneration = DAC_WaveGeneration_None;
   DAC_InitStructure.DAC_LFSRUnmask_TriangleAmplitude = DAC_LFSRUnmask_Bit0;
   DAC_InitStructure.DAC_OutputBuffer = DAC_OutputBuffer_Enable;
   DAC_Init(DAC_Channel_1, &DAC_InitStructure);
   DAC_Cmd(DAC_Channel_1, ENABLE);     // włącz DAC Channel1
}
```

Załączyliśmy wzmacniacz wyjściowy. W płycie STM3210E-EVAL napięcie zasilania części analogowej jest równe napięciu referencyjnemu $V_{DDA} = V_{REF} = 3.3V$. To oznacza, że na wyjściu przetwornika DAC dysponujemy zakresem napięć od 0.2 V do 3.1 V. Generując sygnał wyjściowy musimy ograniczyć go do tego zakresu. W przeciwnym razie nastąpi zniekształcenie sygnału, przez przycięcie napięć mniejszych od 0.2 V i większych od 3.1 V.

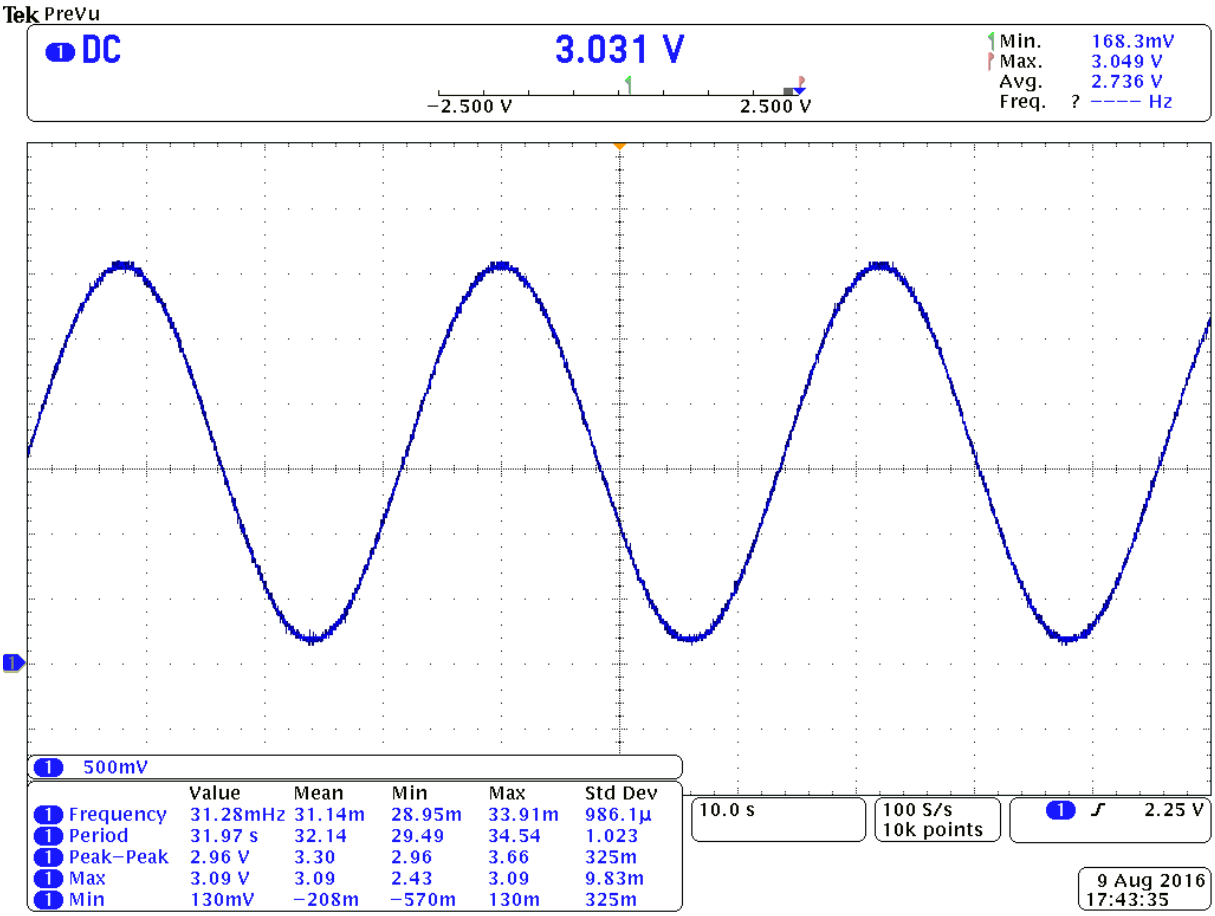
Wartości binarnej `0x0FFF` = 4095 odpowiada napięcie 3.3 V. To oznacza, że napięciu 3.1 V odpowiada wartość binarna

$$\frac{3.1}{3.3}4095 = 3846.8 \approx 3846 \qquad \rightarrow \qquad 3846 - 2048 = 1798$$

Środek zakresu to 2048. Zakres od środka w obie strony: $2048 - 1798 = 250$ (0.2 V) i $2048 + 1798 = 3846$ (3.1 V).

W osi poziomej, na wyświetlaczu mamy 320 pikseli. Podstawę czasu naszej sinusojdy wyliczymy tak, aby na ekranie zmieścił się jeden okres. To oznacza, że liczbie 320 odpowiada 2Π, współczynnik jest więc równy: 2Π/320 = 0.0196349541. Program obsługi przerwania SysTick:

```
extern unsigned int ADC1_DR, ADC2_DR;     // wartości binarne z ADC1/2
extern unsigned int SysTickCounter;      // licznik do wyświetlania
void SysTick_Handler(void){
   float x, y;
   // przetworniki A/D
   ADC1_DR = ADC_GetConversionValue(ADC1); // wartość binarna z ADC1
   ADC2_DR = ADC_GetConversionValue(ADC2); // wartość binarna z ADC2
   ADC_SoftwareStartConvCmd(ADC1, ENABLE); // rozpocznij przetwarzanie ADC1
   ADC_SoftwareStartConvCmd(ADC2, ENABLE); // rozpocznij przetwarzanie ADC2
   // przetwornik D/A - generacja sinusa
   if (SysTickCounter >= 320)
   SysTickCounter = 0;
   x = SysTickCounter * 0.0196349541; // podstawa czasu
   y = 1798.0 * sin(x)+ 2048.0;       // sinus od 250 (0.2V) do 3846 (3.1V)
   DAC_SetChannel1Data(DAC_Align_12b_R, (uint16_t) y);
   SysTickCounter++;
}
```



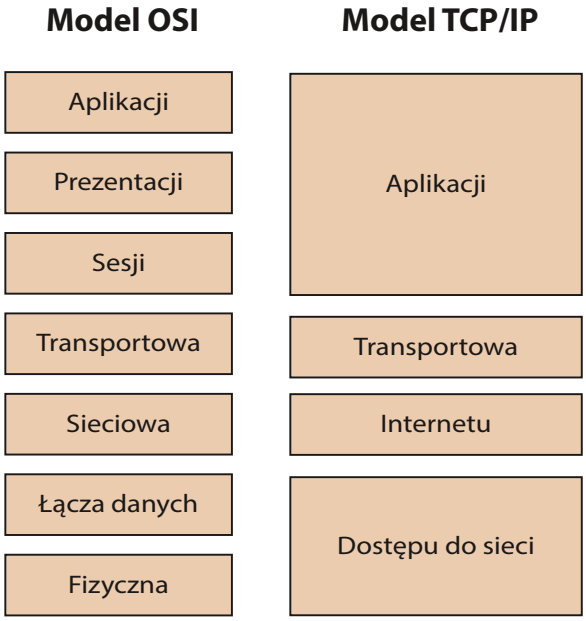
Rys. 17.11: Oscylogram sygnału wyjściowego, gniazdo CN2

Rozdział 18

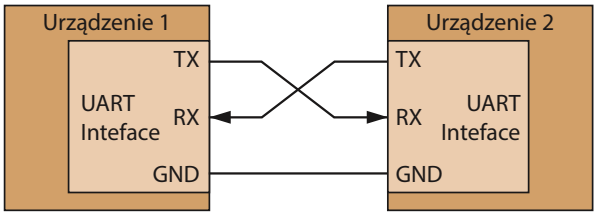
Transmisja Szeregowa

W tym rozdziale omówimy trzy podstawowe standardy transmisji szeregowej: *Universal Synchronous /Asynchronous Receiver-Transmitter* USART, *Inter-Integrated Circuit* I^2C oraz *Serial Peripheral Interface* SPI. W dawnych komputerach osobistych używany był port drukarki LPT (Centronics IEEE 1284), gdzie jednocześnie przesyłanych jest osiem bitów danych. W przeciwieństwie do tego, port szeregowy ma tylko jedną linię danych (ewentualnie dwie – po jednej w każdym kierunku), którą bajty są przesyłane bit po bicie. W porównaniu z portem równoległym, transmisja szeregowa oferuje mniejszą przepustowość, ale wymaga przewodów o znacznie mniejszej liczbie żył i umożliwia zestawianie połączeń na większe odległości.

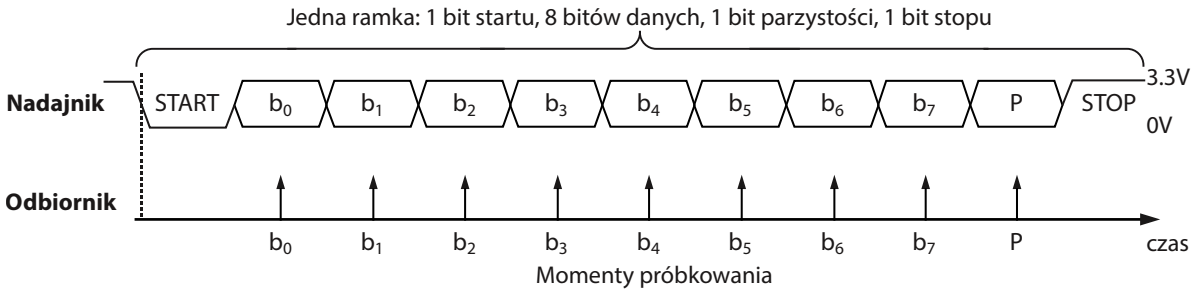
Międzynarodowa Organizacja Normalizacyjna (*International Organization for Standardization*) na początku lat osiemdziesiątych przyjęła specyfikację *Open Systems Interconnection Reference Model* OSI, która do polskich norm została zaadaptowana w 1995 roku. ISO OSI RM definiuje model odniesienia (wzorzec) dla większości rodzin protokołów komunikacyjnych. Podstawowym założeniem modelu jest podział systemów sieciowych na 7 warstw (*layers*) współpracujących ze sobą w ściśle określony sposób. Został on przyjęty przez ISO w 1984 roku. Nas będą interesowały tylko dwie najniższe warstwy modelu OSI, czyli fizyczna i łącza danych. Ilość portów USART/USB/CAN/SDIO w procesorach STM32F103VB (płyta ZL27ARM) i STM32F103ZE (płyta STM3210E-EVAL) podane są w tabeli 13.1.



Rys. 18.1: Porównanie warstw modelu OSI i modelu TCP/IP



Rys. 18.2: Połączenie dwóch układów UART



Rys. 18.3: Ramka w transmisji UART

18.1. USART Universal Synchronous/Asynchronous Receiver-Transmitter

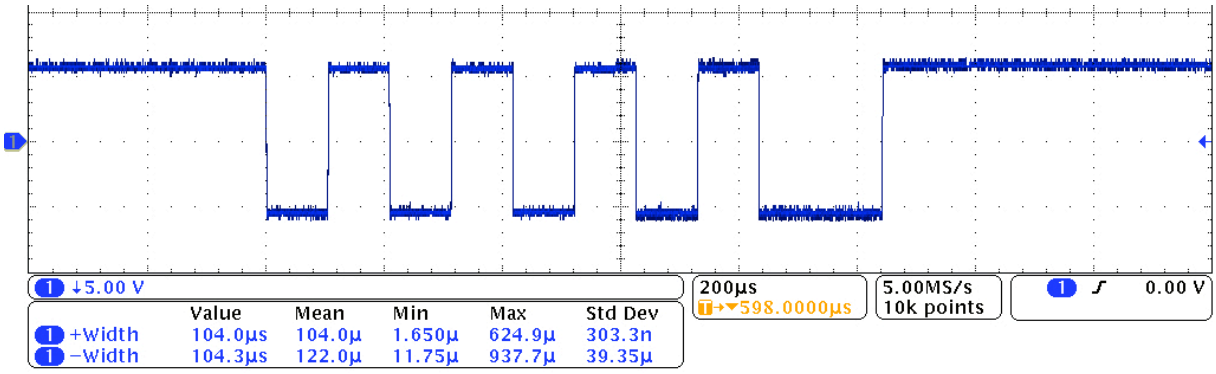
Kanał transmisji szeregowej z możliwością transmisji synchronicznej i asynchronicznej to USART (*Universal Synchronous/Asynchronous Receiver-Transmitter*). Jeżeli możliwa jest wyłącznie transmisja asynchroniczna, to taki kanał oznaczamy skrótem UART. Podstawowe parametry transmisji (prędkość bodowa, bit parzystości, ilość bitów stopu, transmisja asynchroniczna /synchroniczna) kanału USART są programowalne.

Tryb asynchroniczny nie wymaga przekazywania do odbiornika sygnału taktującego. Nadajnik i odbiornik muszą być nastawione na taką samą prędkość bodową przed rozpoczęciem transmisji. **Prędkość bodowa** (*baud rate*) to ilość bitów przesyłanych w ciągu sekundy. Wyrażamy ją w bodach. Na przykład 300 bodów (bitów na sekundę) oznacza ,że każdy bit trwa 1/300 sekundy. Najmniejszą stosowaną jest 110 bodów – dawne dalekopisy i telegrafy. Tradycyjne wartości to: 300, 600, 1200, 2400, 4800, 9600, 19200, 38400, 57600 oraz 115200 bodów.

Transmisję szeregową należy zestawić w sposób pokazany na rysunku 18.2. Wyjście nadajnika `Tx` pierwszego urządzenia, jest połączone z wejściem odbiornika `Rx` w drugim urządzeniu i odwrotnie.

18.1.1. Ramka

Kanał UART przesyła dane w ramkach (*frames*) – rysunek 18.3. Jest to najmniejsza, możliwa do przesłania ilość informacji. Poziom wysoki linii oznacza stan spoczynku. Ramkę rozpoczyna zbocze opadające – bit startu zawsze ma poziom niski, czyli przeciwny do stanu spoczynku linii. Pierwsze zbocze opadające służy do synchronizacji całej ramki w odbiorniku. Po czasie połowy długości trwania bitu, próbkowany jest stan linii. W rzeczywistości rejestrowane są trzy próbki (bit dzielony jest na 16 równych odcinków) – środkowa i dwie boczne. O wartości bitu decydują dwie jednakowe próbki, na przykład `0,1,0` traktowane jest jako zero. Jeżeli bit startu nie jest zerem, odbiornik uznaje, że



Rys. 18.4: Oscylogram: RS-232, 9600, `0x55/even/1stop`

było to zakłócenie i wraca do oczekiwania na zbocze opadające. Jeżeli bit startu jest zerem, odczytywanych jest osiem kolejnych bitów informacji, bit parzystości (jeśli jest) i bit stopu, którego stan jest równy stanowi spoczynku linii. Brak bitu stopu lub błędny bit parzystości oznacza wadliwą transmisję, i odczytany bajt jest odrzucany.

Synchronizacja następuje z każdym zboczem opadającym bitu startu nowej ramki. Dzięki temu dopuszczalna jest pewna rozbieżność w prędkościach bodowych nadajnika i odbiornika – taka żeby momenty próbkowania mieściły się w okresach czasu przewidzianych dla każdego bitu ramki.

Bity danych przesyłane są od najmłodszego do najstarszego. Ilość bitów danych jest programowalna: może to być 7 bitów, lub pełny bajt, czyli 8 bitów. Siedem bitów jest zaszłością historyczną i związane jest z pierwszym, powszechnie stosowanym protokołem transmisji: **Intel HEX Format**, w którym stosowano wyłącznie 7–bitowe znaki ASCII7. Po bitach danych, a przed bitem stopu, jest miejsce na bit parzystości. Możliwe opcje to:

- *EVEN* – bity danych wraz z bitem parzystości zawierają parzystą ilość jedynek
- *ODD* – bity danych wraz z bitem parzystości zawierają nieparzystą ilość jedynek
- brak bitu parzystości

Ramkę kończy znacznik stopu, możliwe długości stopu to 0.5, 1, 1.5 albo 2 bity.

Oscylogram transmisji szeregowej bajtu `0x55 = 0101 0101` pokazany jest na rysunku 18.4. Standard napięciowy RS-232, $\pm 6V$. Prędkość bodowa 9600, czas trwania jednego bitu $t = 1/9600 = 104.17\mu s$. Bit parzystości *even* i jeden bit stopu. Sekwencja bitów:

stan spoczynku linii, START, dane od LSB, parzystość, STOP, stan spoczynku linii

1 1 1 1 1 1 1 1 0 1 0 1 0 1 0 1 0 0 1 1 1 1 1 1 1 1

Bajt danych to `0x55 = 0101 0101`. Zapisany w odwrotnej kolejności: `1010 1010`. Bajt danych zawiera cztery jedynki, a więc parzystą ilość, dlatego bit parzystości jest zerowy.

18.1.2. Prędkość bodowa

Tylko USART1 jest taktowany z PCLK2 (72 MHz max), pozostałe są taktowane z PCLK1 (36 MHz max). Dlatego USART1 umożliwia transmisję z maksymalną prędkością

4.5 Mega bitów na sekundę. Dla pozastałych układów USART maksymalna prędkość to 2.25 Mbps. Prędkość bodowa układu USART jest określona zgodnie ze wzorem

$$Baud_rate = \frac{f_{PCLK}}{16 \cdot USARTDIV}$$

gdzie f_{PCLK} to częstotliwość taktowania linii PCLK2 dla USART1 oraz PCLK1 dla pozostałych układów USART. Natomiast USARTDIV to podzielnik w formacie $Q12.4$, czyli podzielnik jest liczbą stałopozycyjną, której część całkowita ma 12 bitów, a część ułamkowa 4 bity. Załóżmy, że częstotliwość PCLK2 to 72 MHz i chcemy uzyskać 9600 bodów w USART1. Na wyjściu stałego podzielnika [:16] mamy częstotliwość 72/16 = 4.5 MHz. Wymagany podzielnik USARTDIV = $4'500'000/9'600 = 468.75$. Część całkowita 468 =`0x1D4`, część ułamkowa to $0.75 = 3/4 = 12/16$ czyli `0xC`. Ostatecznie do rejestru `USART_BRR` należy wpisać wartość USARTDIV = `0x1D4C`. W tabeli poniżej podano podzielniki dla kilku popularnych prędkości bodowych.

Tabela 18.1: Podzielniki dla częstotliwości taktowania 36 i 72 MHz

prędkość bodowa	f_{PCLK} = 36 MHz			f_{PCLK} = 72 MHz		
	real rate	**podzielnik**	**błąd**	**real rate**	**podzielnik**	**błąd**
2'400	2'400	937.5	0 %	2'400	1875	0 %
9'600	9'600	234.375	0 %	9'600	468.75	0 %
19'200	19'200	117.1875	0 %	19'200	234.375	0 %
57'600	57'600	39.0625	0 %	57'600	78.125	0 %
115'200	115'384	19.5	0.15 %	115'200	39.0625	0 %

18.1.3. Warstwa fizyczna

Na warstwę fizyczną składa się format ramki (jak na rysunku 18.3), prędkość bodowa oraz standard elektryczny. Najbardziej popularne standardy elektryczne to RS232, RS485 i RS422. W tabeli 18.2 podano maksymalny dystans, maksymalną prędkość i maksymalną ilość układów, które można połączyć w jednej pętli.

Tabela 18.2: Standardy elektryczne transmisji USART

standard	**sygnał elektryczny**	**dystans**	**prędkość**	**ilość układów**
RS-232	sygnał napięciowy: 1 to $(+5, +15V)$, 0 to $(-5, -15V)$	15 m	115'200 bps	1 nadajnik, 1 odbiornik
RS-422	różnicowy: $-6V \leftrightarrow +6V$	1200 m	10 Mbps	1 nadajnik, 10 odbiorników
RS-485	różnicowy: $-7V \leftrightarrow +12V$	1200 m	10 Mbps	32 nadajniki, 32 odbiorniki

RS-422/485: maksymalna odległość na jaką może odbywać się transmisja to 1200 m, ale przy takim dystansie prędkość transmisji spada do 90 kbps. Transfer 10 Mbps można uzyskać na dystansie nie większym niż 12 m.

```
:10310000000102030405060708090A0B0C0D0E0F47
:10311000101112131415161718191A1B1C1D1E1F37
:10312000202122232425262728292A2B2C2D2E2F27
:00000001FF
```

znak rozpoczynający
ilość bajtów
adres
typ rekordu
dane
suma kontrolna

Rys. 18.5: Przykład transmisji w formacie Intel HEX

18.2. Warstwa łącza danych

18.2.1. INTEL HEX Format

Intel HEX Format to pierwszy protokół transmisji zdefiniowany przez firmę Intel w latach siedemdziesiątych XX wieku. Wykorzystuje wyłącznie 7–bitowe znaki ASCII7. Umożliwia przesyłanie danych binarnych, kodowanych HEX, tekstowo, czyli z wykorzystaniem znaków `0,1,...9,A,B,...,F` oraz dwukropka. Dane przesyłane są w postaci rekordów. Każdy rekord składa się z sześciu pól

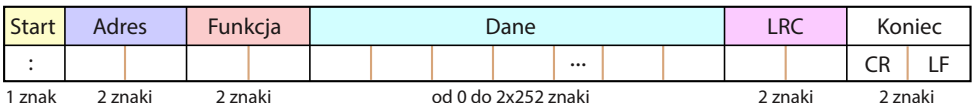
1. Rekord rozpoczyna się dwukropkiem (jeden znak ASCII)
2. Liczba bajtów: dwie cyfry hexadecymalne, określające długość rekordu danych
3. Adres: cztery cyfry hexadecymalne, czyli 16 bitowy adres, od którego należy zapisywać kolejno bajty danych. Umożliwia zaadresowanie do 64K pamięci.
4. Typ rekordu: dwie cyfry hexadecymalne w przedziale od 00 do 05
5. Dane: sekwencja danych, gdzie bajt zapisany jest jako dwa znaki HEX
6. Suma kontrolna rekordu: dwie cyfry hexadecymalne. Stosowana jest suma LRC (*Longitudinal Redundancy Check*). Najmłodszy bajt sumy całego rekordu (bez dwukropka) łącznie z bajtem sumy kontrolnej, jest zerem.

Istnieje sześć różnych typów rekordów, ale najważniejsze są dwa:
00 – rekord zawierający dane oraz 16 bitowy adres,
01– koniec bloku.

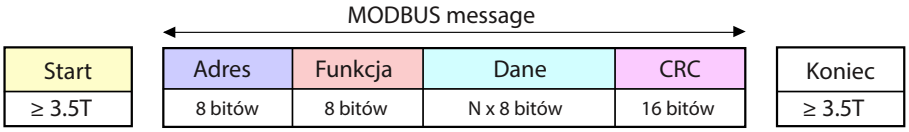
Pełna specyfikacja podana jest w dokumencie *Intel Hexadecimal Object File Format Specification 1988 (PDF).*

18.2.2. Modbus

Modbus to otwarty protokół komunikacyjny stworzony przez firmę Modicon w roku 1979. Istnieje w dwóch odmianach: znakowej ASCII i binarnej RTU.
Modbus ASCII. Bajty są wysyłane szesnastkowo, po dwa znaki ASCII. Dane w ramce są zabezpieczone sumą kontrolną, liczoną tak samo jak w Intel HEX Format. Ramki zaczynają się dwukropkiem i kończą się znakiem nowej linii CR/LF (`0x0A,0x0D`). Ramka pokazana jest na rysunku 18.6



Rys. 18.6: Ramka Modbus ASCII



Rys. 18.7: Ramka Modbus RTU

Modbus RTU. Bajty w ramce są wysyłane binarnie jako znaki ośmiobitowe. Ramka zabezpieczona jest sumą kontrolną CRC. Znaki w ramce muszą być przesyłane w sposób ciągły bez przerw między znakami (maksymalnie 1.5T gdzie T oznacza czas transmisji jednego znaku). Każda ramka poprzedzona jest odstępem (ciszą na linii) większym niż 3.5T. Ramka pokazana jest na rysunku 18.7. Dokładna specyfikacja protokołu zawarta jest w dokumencie *MODBUS over serial line specification and implementation guide V1.02*, który można pobrać ze strony Modbus.org.

18.2.3. Master (klient) – slave (serwer danych)

Protokół Modbas jest typu *Master – Slave*. Do magistrali transmisyjnej jest dołączony tylko jeden *Master*, zwany też klientem oraz pewna ilość serwerów danych (*slaves*). Tylko *Master* inicjuje transmisję – wysyła polecenia i zapytania. Układy *slave* nie wysyłają danych bez pytania od *Master* oraz nie wymieniają informacji pomiędzy sobą. *Master* może wysyłać polecenia w dwóch trybach:

- **Tryb unikatowy** (*unicast mode*), gdy *Master* adresuje jeden, wybrany serwer danych. W tym trybie *Master* wysyła polecenie, a *slave* na nie odpowiada. Wszystkie układy *slave* muszą mieć unikalne adresy: od 1 do 247. Adresy od 248 do 255 są zarezerwowane i nie należy ich używać do adresowania serwerów danych. Układ *Master* nie ma adresu. Układ *slave* odsyła swój adres.
- **Tryb rozgłoszeniowy** (*broadcast mode*). *Master* wysyła polecenie do wszystkich serwerów danych jednocześnie, ale żaden nie odpowiada transmisją. Przykładem

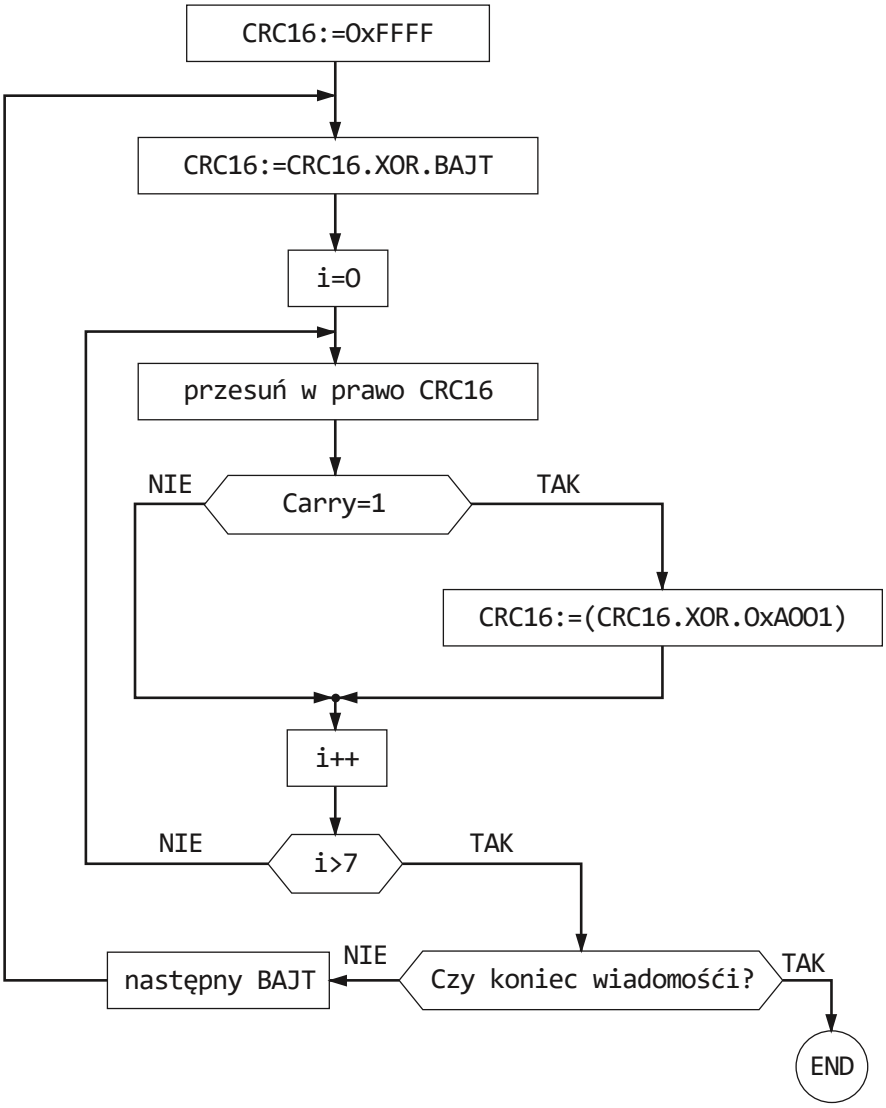
Tabela 18.3: Podstawowe cechy protokołów Modbus

	Modbus ASCII		**Modbus RTU**	
Znaki	ASCII: 0,1,2,3,4,5,6,7,8,9,A,B,C,D,E,F		8–bitowe 0 . . . 255	
Kontrola błędów	LRC Longitudinal Redundancy Check		CRC Cyclic Redundancy Check	
Start ramki	: dwukropek		cisza minimum 3.5T	
Koniec ramki	znaki CR,LF		cisza minimum 3.5T	
Przerwy (max)	1 sec		1.5 T	
Ilość bitów	7		8	
Parzystość	even/odd	brak	even/odd	brak
Bity stopu	1	2	1	2

trybu rozgłoszeniowego jest wysłanie nowego czasu (synchronizacja zegarów). Dla trybu rozgłoszeniowego jest zarezerwowany adres zerowy .

18.2.4. CRC w Modbus RTU

Pole CRC (*Cyclic Redundancy Check*) w ramce Modbus RTU (rysunek 18.7) zawiera dwa bajty, czyli 16 bitów. CRC jest wyliczane przez nadajnik i dołączane do ramki. Odbiornik wylicza swoje CRC dla odebranej ramki i porównuje z odebranym CRC. Jeżeli jest różnica, to oznacza, że wystąpiły zakłócenia i transmisja powinna być powtórzona. Mikrokontrolery Cortex-M3 są wyposażone w układ obliczający CRC, ale nie będziemy mogli z niego skorzystać, gdyż wykorzystuje on współcześnie stosowany klucz `0x4C11DB7`. Protokół Modbus stworzono w czasach, gdy nie były znane szczegółowe analizy skuteczności różnych kluczy. W Modbus RTU stosowany jest klucz `0xA001` i CRC startowane jest od `0xFFFF`, a nie od zera. Algorytm obliczeń pokazany jest na rysunku 18.8.



Rys. 18.8: Algorytm liczenia CRC Modbus RTU

Mamy do przesłania pewną paczkę informacji, czyli pewną ilość uszeregowanych bajtów. Obliczenie CRC zaczynamy od wypełnienia 16-bitowego rejestru samymi jedynkami.

Pobieramy pierwszych osiem bitów z paczki, uzupełniamy zerami z lewej strony i wykonujemy działanie logiczne XOR na obu 16-bitowych ciągach. Wynik przesuwamy o jeden w prawo. Jeżeli wysuwany bit (najmniej znaczący) jest jedynką, to wykonujemy XOR przesuniętego wyniku z kluczem, czyli z `0xA001`. Wysunięcie zera oznacza brak działania. Przesunięcia (oraz XOR przy każdej jedynce) wykonujemy łącznie osiem razy.

Pobieramy kolejny bajt z paczki, uzupełniamy zerami z lewej i wykonujemy osiem przesunięć z XOR przy każdej jedynce, ale zaczynamy z wartością CRC otrzymaną z poprzedniej sekwencji. Gdy opisaną sekwencję wykonamy dla wszystkich bajtów z paczki, to uzyskamy CRC dla całej paczki.

```
// ----------------- obliczamy CRC Modbus RTU -----------------
uint16_t ModRTU_CRC(uint8_t bufor[], int len){
   uint16_t crc = 0xFFFF;
   volatile unsigned int k, i;
   for (k = 0; k < len; k++){
     crc ^= (uint16_t)bufor[k];   // XOR kolejnego bajtu z CRC
     for (i = 8; i != 0; i--){    // pętla po wszystkich ośmiu bitach
       if ((crc & 0x0001) != 0){  // jeżeli LSB jest jedynką, to ...
         crc >>= 1;               // ... przesuń w prawo, XOR 0xA001
         crc ^= 0xA001;
       }
       else                     // jesli LSB jest zerem, to ...
         crc >>= 1;             // ... tylko przesuń
     }
   }
   return crc; //UWAGA: w Modbus RTU, CRC jest wysyłane: młodszy, starszy
}
```

18.3. Projekt24_1_Modbus_Master

Uruchomimy transmisję pomiędzy jednym klientem (*Master*) i jednym serwerem danych (*slave*). Serwer ma adres `0x01` i dwa rejestry. Rejestry mają po 4 bajty długości i zawierają liczby typu `float`. Numery rejestrów:

- `0x0000` = bieżąca, zmierzona wartość prądu w [mA],
- `0x0001` = bieżąca, zmierzona wartość temperatury w [°C],

Zgodnie z tabelą 18.3, gdy stosujemy bit parzystości, to musi być dokładnie jeden bit stopu.

Warstwa łącza danych:

- Modbus RTU

Warstwa fizyczna:

- RS-232, USART1, złącze DB9F
- 9600 bodów
- 8 bitów danych
- bit parzystości
- jeden bit stopu

Zastosujemy funkcję 0x03. Jest to polecenie przesłania zawartości określonej liczby rejestrów, ułożonych w sposób ciągły, od podanego numeru. Gdy poprosimy o dwa rejestry od adresu 0x0000, to otrzymamy zawartości rejestrów 0x0000,0x0001. Formaty ramek funkcji 0x03 pokazane są na rysunku 18.9

Kod funkcji 0x03 - odczytaj N rejestrów (N =20 max)

Pytanie

adres slave	kod funkcji	adres startowy	liczba rejestrów	CRC 16
1 bajt	03	2 bajty	2 bajty	2 bajty

Odpowiedź

adres slave	kod funkcji	liczba bajtów	dane 1	...	CRC
1 bajt	03	1 bajt			2 bajty

Rys. 18.9: Formaty ramek funkcji 0x03

W pliku *main.c* zdefiniujemy zmienne globalne

```
// zmienne globalne ---------------------------------------------
unsigned char txt2LCD[32] = {0}; // bufor funkcji sprintf
uint8_t Tx_message[32]={0};      // bufor na znaki do nadania
uint8_t Rx_message[32]={0};      // bufor na znaki odebrane
unsigned int Tx_num = 0;         // licznik ramek nadanych
unsigned int Rx_num = 0;         // licznik ramek odebranych
```

Z tabeli 15.1 możemy odczytać, że kanał czwarty DMA1 obsługuje nadawanie USART1 (USART1_Tx). Napiszemy funkcje konfigurujące kanał 4 w DMA1 oraz USART1.

```
// ----------------- konfiguracja kanału DMA ------------------
void DMA_Config(void){
   #define USART1_DR_Address 0x40013804;      //adres rejestru USART1_DR
   DMA_InitTypeDef DMA_InitStructure;

   DMA_DeInit(DMA1_Channel4);  // usuń poprzednią konfigurację DMA
   DMA_InitStructure.DMA_PeripheralBaseAddr = (uint32_t) USART1_DR_Address;
   DMA_InitStructure.DMA_MemoryBaseAddr = (uint32_t) Tx_message;
   DMA_InitStructure.DMA_DIR = DMA_DIR_PeripheralDST; // kierunek transferu
   DMA_InitStructure.DMA_BufferSize = 8; // liczba elementów do przesłania
   DMA_InitStructure.DMA_PeripheralInc = DMA_PeripheralInc_Disable;
   DMA_InitStructure.DMA_MemoryInc = DMA_MemoryInc_Enable;
   DMA_InitStructure.DMA_PeripheralDataSize = DMA_PeripheralDataSize_Byte;
   DMA_InitStructure.DMA_MemoryDataSize = DMA_MemoryDataSize_Byte;
   DMA_InitStructure.DMA_Mode = DMA_Mode_Normal;  // jeden transfer
   DMA_InitStructure.DMA_Priority = DMA_Priority_High;
   DMA_InitStructure.DMA_M2M = DMA_M2M_Disable;
   DMA_Init(DMA1_Channel4, &DMA_InitStructure);  // zapis konfiguracji
}
```

Będziemy nadawać polecenie o długości $1 + 1 + 2 + 2 + 2 = 8$ bajtów (rysunek 18.9). Co sekundę będzie inicjowany transfer DMA z pamięci `Tx_message` do rejestru danych układu USART. Adres pamięci będzie inkrementowany, adres rejestru `USART1_DR` będzie stały. W powyższej funkcji nie ma polecenia `ENABLE`, czyli zezwolenia kanału czwartego, jest tylko zapis parametrów transferu do DMA.

```
// ------------------- konfiguracja USART1 --------------------
void USART_Config(void){
   USART_InitTypeDef USART_InitStructure;
   USART_InitStructure.USART_BaudRate = 9600;
   USART_InitStructure.USART_WordLength = USART_WordLength_9b;
   USART_InitStructure.USART_StopBits = USART_StopBits_1;
   USART_InitStructure.USART_Parity = USART_Parity_Even;
USART_InitStructure.USART_HardwareFlowControl=USART_HardwareFlowControl_None;
   USART_InitStructure.USART_Mode = USART_Mode_Rx | USART_Mode_Tx;
   USART_Init(USART1, &USART_InitStructure);
   USART_Cmd(USART1, ENABLE);
}
```

W oprogramowaniu bibliotecznym przyjęto konwencję, że długość słowa transmisyjnego jest liczona razem z bitem parzystości, dlatego wybraliśmy `USART_WordLength_9b`. Konfigurację płyty musimy uzupełnić o:

```
USART_Config();
DMA_Config();
USART_DMACmd(USART1, USART_DMAReq_Tx, ENABLE); // transfer DMA -> USART1_Tx
```

Teraz utworzymy ramkę do nadania i załączymy przerwania systemowe SysTick

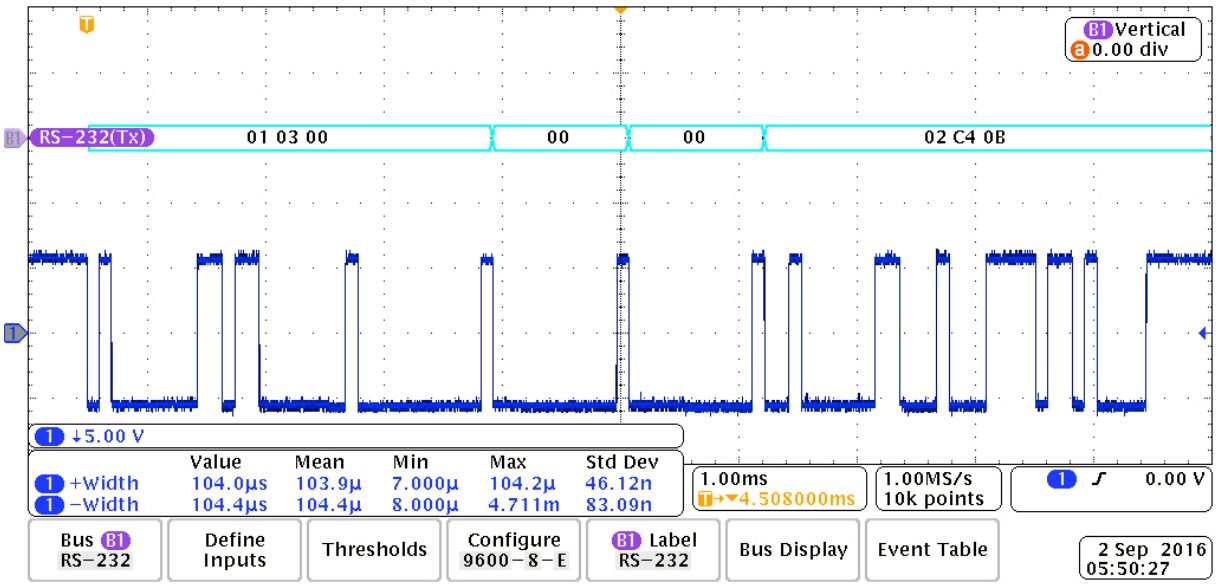
```
char * wskaznik;

Tx_message[0] = 0x01;  // adres slave
Tx_message[1] = 0x03;  // funkcja
Tx_message[2] = 0x00;  // adres pierwszego rejestru
Tx_message[3] = 0x00;
Tx_message[4] = 0x00;  // ilość rejestrów
Tx_message[5] = 0x02;
crc = ModRTU_CRC(Tx_message, 6);
wskaznik = (char *) &crc;
Tx_message[6] = wskaznik[0];
Tx_message[7] = wskaznik[1];
SysTick_Config_Div8(9000000ul);  // przerwanie co 1 sekundę
```

Pozostało napisanie programu obsługi przerwania systemowego. Przerwanie systemowe będzie wyzwalane co sekundę i przez zezwolenie kanału czwartego w DMA1, będzie inicjowało nadawanie. W konfiguracji DMA wybraliśmy opcję pojedynczego transferu. To oznacza, że zostanie nadana jedna ramka. Po zakończeniu transferu, zainicjowany adres rejestru danych w układzie USART, oraz zainicjowany adres pamięci z początkiem ramki do nadania, pozostają niezmienione. Natomiast licznik bajtów do nadania w rejestrze

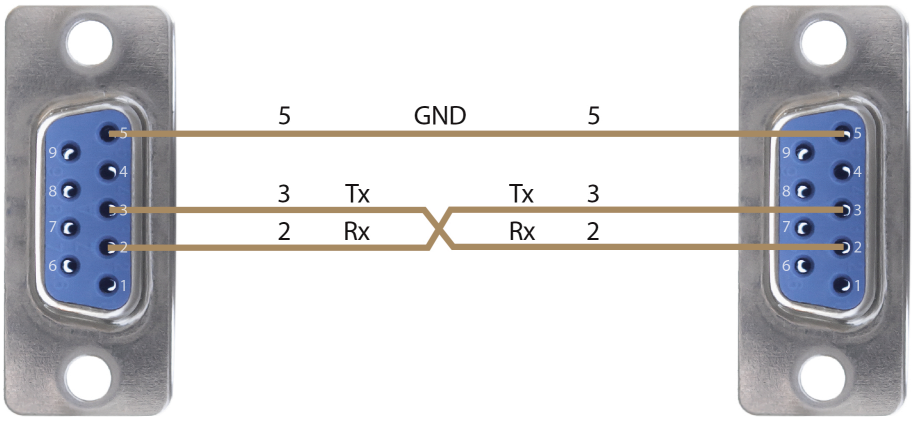
CNDTR4 jest dekrementowany w trakcie transferu aż do osiągnięcia wartości zerowej. W kolejnym wywołaniu obsługi przerwania, do CNDTR4 musimy wpisać ilość bajtów do wysłania, ale ten wpis jest możliwy tylko, gdy kanał jest zabroniony. Dlatego wykonamy sekwencję: zabronimy kanał czwarty, a następnie wpiszemy ilość bajtów do wysłania (w naszym przypadku 0x08). Zezwolenie kanału czwartego jest początkiem transmisji.

```
/* This function handles SysTick Handler ****************************/
extern unsigned int Tx_num;            // licznik ramek nadanych
void SysTick_Handler(void){
   DMA1 -> IFCR = 0x00007000;          // kasujemy zgłoszenia przerwań
   DMA_Cmd(DMA1_Channel4, DISABLE);    // wyłącz kanał 4 /DMA1
   DMA1_Channel4 -> CNDTR = 0x08;      // ilość bajtów do przesłania
   DMA_Cmd(DMA1_Channel4, ENABLE);     // włącz kanał 4 /DMA1 = transmisja
   Tx_num ++;
}
```

Wynik działania programu pokazany jest na rysunku 18.10. Co sekundę nadawane jest polecenie: 01 03 0000 0002 C40B



Rys. 18.10: Oscylogram polecenia wysyłanego przez *Mastera*



Rys. 18.11: Kabel RS-232 z przeplotem

18.4. Projekt25_Modbus_slave

Kolejny projekt to program serwera danych, który zrealizujemy na drugiej płycie STM3210E-EVAL. Zaczniemy od wykonania kabla RS-232, **z przeplotem**, jak pokazano na rysunku 18.11. Przeplot oznacza, że połączone są pary nadajnik–odbiornik. Kablem łączymy porty *CN12 USART1* w obu płytach.

Transmisję szeregową można zrealizować wykorzystując parę DMA–USART. Programowo zezwalamy (`ENABLE`) kanał DMA i cała ramka jest wysyłana sprzętowo bez udziału procesora. Jest to bardzo efektywny sposób. Jednak nie zawsze dysponujemy układem DMA. Wtedy nadawanie (odbiór) bajtów do (z) kanału USART można zorganizować z wykorzystaniem przerwań ("na przerwaniach"). Każde zwolnienie rejestru danych nadajnika generuje przerwanie, w którym należy wpisać następny bajt do nadania. Gdy odebrany bajt oczekuje na odczytanie go z rejestru odbiornika, generowane jest kolejne przerwanie. Programy dołączone do książki mają cel dydaktyczny, dlatego w tym projekcie zarówno nadawanie jak i odbiór zrobimy na przerwaniach. Oczywiście można mieszać ze sobą oba sposoby: na przykład nadawanie DMA i odbiór na przerwaniach lub odwrotnie.

Wskaźniki na bufory bajtów nadawanych i odbieranych musimy zadeklarować jako zmienne globalne, gdyż zmienne lokalne programu obsługi przerwania znikają z chwilą zakończenia jego wykonywania. W chwili odebrania kolejnego bajtu, ustawimy licznik `SysTick_count`, który będzie dekrementowany w przerwaniu `SysTick`. Zmiana wartości tego licznika z `1` na `0` oznacza wykrycie przerwy czasowej końca ramki odbieranej – wtedy ramka zostanie rozpakowana, sprawdzona, utworzymy odpowiedź i zainicjujemy nadawanie.

```
// zmienne globalne -------------------------------------------
unsigned char txt2LCD[32] = {0};// bufor funkcji sprintf
uint16_t PC2_adc[100]={0};       // bufor na wartości binarne z ADC
uint8_t Tx_message[32]={0};      // bufor na znaki do nadania
uint8_t Rx_message[32]={0};      // bufor na znaki odebrane
uint32_t Tx_index = 0;           // wskaźnik na miejsce w Tx_message
uint32_t Rx_index = 0;           // wskaźnik na miejsce w Rx_message
unsigned int Tx_num = 0;         // licznik ramek nadanych
unsigned int Rx_num = 0;         // licznik ramek odebranych
unsigned int SysTick_count = 0;  // licznik do odmierzania przerwy > 3.5T
float current, T;
```

W konfiguracji NVIC musimy zezwolić na przerwania od USART1

```
// ----------- konfiguracja kontrolera przerwan NVIC ----------
void NVIC_Config(void) {
   NVIC_InitTypeDef NVIC_InitStructure;
   NVIC_SetVectorTable(NVIC_VectTab_FLASH, 0x0);
   NVIC_PriorityGroupConfig(NVIC_PriorityGroup_1);
   NVIC_InitStructure.NVIC_IRQChannel = USART1_IRQn; //Przerwanie od USART1
   NVIC_InitStructure.NVIC_IRQChannelPreemptionPriority = 1;
   NVIC_InitStructure.NVIC_IRQChannelSubPriority = 0;
   NVIC_InitStructure.NVIC_IRQChannelCmd = ENABLE;
   NVIC_Init(&NVIC_InitStructure);
}
```

Konfiguracja układu USART1 jest identyczna jak poprzednio, musimy tylko dodać zezwolenie na generację przerwań od zapełnionego odbiornika.

```
USART_ITConfig(USART1, USART_IT_RXNE, ENABLE);  // Receiver Not Empty
```

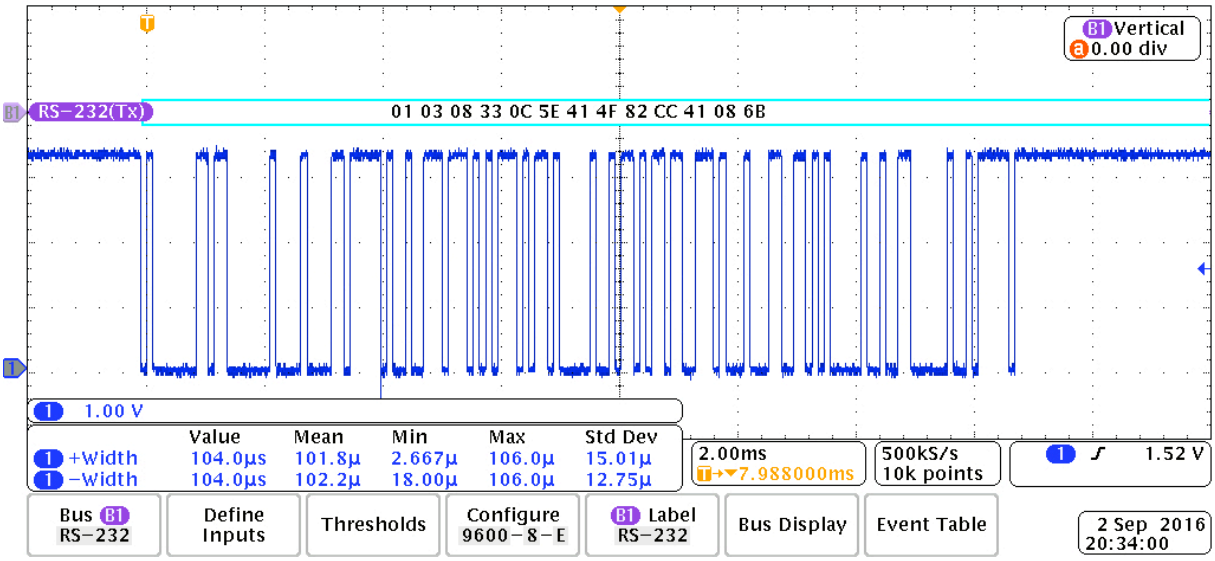
W pętli głównej możemy wyświetlać liczniki ilości ramek nadanych /odebranych, ramki transmisyjne oraz przesyłane liczby `float`. Przerwania `SysTick` zaprogramujemy na 0.01 sekundy.

```
void SysTick_Handler(void){
   unsigned int crc, index;
   char * wskaznik;

   ADC_SoftwareStartConvCmd(ADC1, ENABLE); // rozpocznij przetwarzanie ADC1
   if(SysTick_count > 0){
     SysTick_count--;
     if(SysTick_count == 0){    // wykryta przerwa po zakonczeniu Rx
       Rx_num++;
       index = Rx_index - 1; Rx_index = 0;
       if((Rx_message[0] == 0x01) & (Rx_message[1] == 0x03)){  // czy 0x0103
         crc = ModRTU_CRC(Rx_message, index - 1); // sprawdzimy CRC
         wskaznik = (char *) &crc;
         if((wskaznik[0] == Rx_message[index - 1]) & (wskaznik[1]
           == Rx_message[index])){
           // teraz należy sprawdzić, numer rejestru, ilość rejestrów, ...
           // tworzymy odpowiedź
           Tx_message[0] = Rx_message[0]; // adres
           Tx_message[1] = Rx_message[1]; // funkcja
           Tx_message[2] = 0x08; // ilość bajtów danych
           wskaznik = (char *) &current;
           Tx_message[3] = wskaznik[0];
           Tx_message[4] = wskaznik[1];
           Tx_message[5] = wskaznik[2];
           Tx_message[6] = wskaznik[3];
           wskaznik = (char *) &T;
           Tx_message[7] = wskaznik[0];
           Tx_message[8] = wskaznik[1];
           Tx_message[9] = wskaznik[2];
           Tx_message[10] = wskaznik[3];
           crc = ModRTU_CRC(Tx_message, 11);
           wskaznik = (char *) &crc;
           Tx_message[11] = wskaznik[0];
           Tx_message[12] = wskaznik[1];
           Tx_num++;
           Tx_index = 0;
           USART_ITConfig(USART1, USART_IT_TXE, ENABLE); // start nadawania
       }
     }
   }
}
```

Najważniejsza część programu to obsługa przerwań od USART

```
/* Function Name : USART1_IRQHandler    */
void USART1_IRQHandler(void){
   if(USART_GetITStatus(USART1, USART_IT_RXNE) != RESET){
     Rx_message[Rx_index] = USART_ReceiveData(USART1);
     SysTick_count = 8;        // wartość początkowa do odliczania
     Rx_index++;
     if(Rx_index > 31)
       Rx_index = 31;            // koniec bufora
   }
   if(USART_GetITStatus(USART1, USART_IT_TXE) != RESET){
     USART_SendData(USART1, Tx_message[Tx_index++]); // wyślij kolejny znak
     if(Tx_index > 12)                        // mamy do wysłania 13 bajtów
        USART_ITConfig(USART1, USART_IT_TXE, DISABLE); // koniec transmisji
   }
}
```

Oscylogram przykładowej ramki wysyłanej przez serwer danych pokazany jest na rysunku:



Rys. 18.12: Przykładowa ramka odpowiedzi, wysyłanej przez *Slave*

18.5. Projekt24_2_Modbus_Master

Uzupełnimy program *Master*a o część odbierającą odpowiedź. Wykorzystamy transfer DMA. Z tabeli 15.1 możemy odczytać, że kanał czwarty w DMA1 obsługuje nadawanie USART1 (`USART1_Tx`), natomiast kanał piąty obsługuje odbiór danych z USART1 (`USART1_Rx`). Teraz konfiguracja płyty oprócz zezwolenia na transfer `DMA -> USART1_Tx`, wymaga także zezwolenia na transfer `USART1_Rx -> DMA`

```
USART_DMACmd(USART1, USART_DMAReq_Rx, ENABLE); // transfer USART1_Rx -> DMA
```

W funkcji `DMA_Config()` musimy dopisać konfigurację kanału piątego:

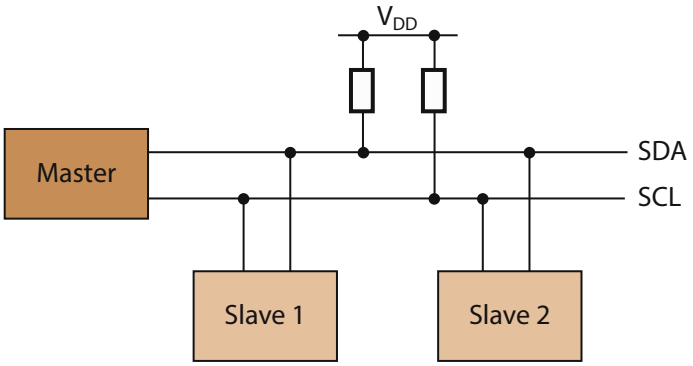
```
DMA_DeInit(DMA1_Channel5);  // usuń poprzednią konfigurację DMA
DMA_InitStructure.DMA_PeripheralBaseAddr = (uint32_t) USART1_DR_Address;
DMA_InitStructure.DMA_MemoryBaseAddr = (uint32_t) Rx_message;
DMA_InitStructure.DMA_DIR = DMA_DIR_PeripheralSRC; // kierunek transferu
DMA_InitStructure.DMA_BufferSize = 13; // liczba elementów do przesłania
DMA_InitStructure.DMA_PeripheralInc = DMA_PeripheralInc_Disable;
DMA_InitStructure.DMA_MemoryInc = DMA_MemoryInc_Enable;
DMA_InitStructure.DMA_PeripheralDataSize = DMA_PeripheralDataSize_Byte;
DMA_InitStructure.DMA_MemoryDataSize = DMA_MemoryDataSize_Byte;
DMA_InitStructure.DMA_Mode = DMA_Mode_Normal;  // jeden transfer
DMA_InitStructure.DMA_Priority = DMA_Priority_High;
DMA_InitStructure.DMA_M2M = DMA_M2M_Disable;
DMA_Init(DMA1_Channel4, &DMA_InitStructure);  // zapis konfiguracji
DMA_Cmd(DMA1_Channel5, ENABLE);  // włącz DMA, kanal 5
DMA_ITConfig(DMA1_Channel5, DMA_IT_TC, ENABLE);  // przerwanie
```

Włączyliśmy kanał, czyli układ oczekuje na transmisję. Włączyliśmy także generację przerwania po odebraniu całej ramki. Konfiguracja USART pozostaje bez zmian, natomiast do konfiguracji NVIC należy dopisać przerwanie od końca transmisji w kanale piątym DMA.

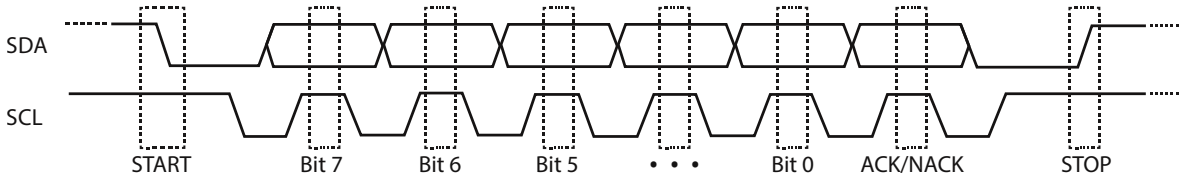
```
// ----------- konfiguracja kontrolera przerwan NVIC ----------
void NVIC_Config(void){
   NVIC_InitTypeDef NVIC_InitStructure;
   NVIC_SetVectorTable(NVIC_VectTab_FLASH, 0x0);
   NVIC_PriorityGroupConfig(NVIC_PriorityGroup_1);
   // Przerwanie od DMA1_Channel5_IRQn
   NVIC_InitStructure.NVIC_IRQChannel = DMA1_Channel5_IRQn;
   NVIC_InitStructure.NVIC_IRQChannelPreemptionPriority = 1;
   NVIC_InitStructure.NVIC_IRQChannelSubPriority = 0;
   NVIC_InitStructure.NVIC_IRQChannelCmd = ENABLE;
   NVIC_Init(&NVIC_InitStructure);
}
```

Master wysyłając pytanie do układów *slave* wie ile bajtów będzie zawierała odpowiedź, dlatego może odpowiednio skonfigurować kanał piąty DMA. Jeżeli jednak w linii odbiorczej wystąpi zakłócenie, które zostanie odebrane jako znak, lub kilka znaków, to po odebraniu łącznie zadanej liczby bajtów układ DMA zgłosi przerwanie. Odebrana ramka nie będzie poprawna, a ponadto po obsłużeniu przerwania, DMA będzie odbierało następne znaki z bieżącej ramki, które zostaną zakwalifikowane do kolejnej ramki. W projekcie nie zastosowaliśmy układu czasowego do wykrywania przerw jako znaczników końca ramki, ale problem rozwiążemy w bardzo prosty sposób: kanał piąty będziemy ustawiali na odbiór kolejnej ramki dopiero po nadaniu pytania. W ten sposób uzyskamy synchronizację pomiędzy ramkami nadawanymi i odbieranymi, a ewentualne zakłócenia na linii odbiorczej nie będą miały wpływu na komunikację pomiędzy płytami. Program obsługi przerwania systemowego SysTick uzupełnimy o ustawienie kanału piątego DMA na odbiór właściwej liczby znaków.

```
/* This function handles SysTick Handler ******************************/
extern unsigned int Tx_num;          // licznik ramek nadanych
void SysTick_Handler(void){
   DMA1 -> IFCR = 0x00007000;        // kasujemy zgłoszenia przerwań
   DMA_Cmd(DMA1_Channel4, DISABLE);  // wyłącz kanał 4 /DMA1
   DMA1_Channel4 -> CNDTR = 8;       // ilość bajtów do przesłania
   DMA_Cmd(DMA1_Channel4, ENABLE);   // włącz kanał 4 /DMA1 = transmisja
   Tx_num ++;
   DMA1 -> IFCR = 0x00070000;        // kasujemy zgłoszenia przerwań
   DMA_Cmd(DMA1_Channel5, DISABLE);  // wyłącz kanał 5 /DMA1
   DMA1_Channel4 -> CNDTR = 13;      // ilość bajtów do przesłania
   DMA_Cmd(DMA1_Channel5, ENABLE);   // włącz kanał 5 /DMA1 = transmisja
}

/*********************************************************************
* Function Name : DMA1_Channel5_IRQHandler
* Description : This function handles DMA1 Channel 5 interrupt request.
*********************************************************************/
extern uint8_t Rx_message[32];  // bufor na znaki odebrane
extern unsigned int Rx_num;    // licznik ramek odebranych
extern float current, T;
uint16_t ModRTU_CRC(uint8_t bufor[], int len);

void DMA1_Channel5_IRQHandler(void){
   unsigned int crc;
   char * wskaznik;
   DMA1 -> IFCR = 0x00070000;           // kasujemy zgłoszenie przerwania
   if((Rx_message[0]==0x01)&(Rx_message[1]==0x03)&(Rx_message[2]==0x08)){
      crc = ModRTU_CRC(Rx_message, 11); // sprawdzimy CRC
      wskaznik = (char *) &crc;
      if((wskaznik[0]==Rx_message[11]) & (wskaznik[1]==Rx_message[12])){
         Rx_num++;                      // licznik ramek odebranych
         wskaznik = (char *) &current;
         wskaznik[0] = Rx_message[3];
         wskaznik[1] = Rx_message[4];
         wskaznik[2] = Rx_message[5];
         wskaznik[3] = Rx_message[6];
         wskaznik = (char *) &T;
         wskaznik[0] = Rx_message[7];
         wskaznik[1] = Rx_message[8];
         wskaznik[2] = Rx_message[9];
         wskaznik[3] = Rx_message[10];
      }
   }
}
```



Rys. 18.13: Przykład magistrali I^2C: jeden *Master* i dwa *Slave*



Rys. 18.14: Wykres czasowy transmisji I^2C

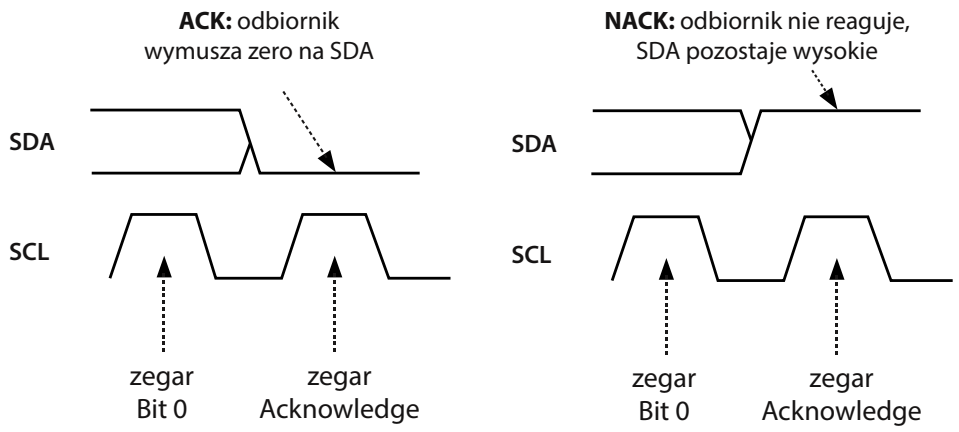
18.6. I²C Inter-Integrated Circuit

I²C to *Inter-Integrated Circuit*, czyli magistrala łącząca układy scalone, najczęściej w obrębie jednej płyty. Jest to transmisja szeregowa synchroniczna *half-duplex*, czyli dwukierunkowa, ale bez możliwości jednoczesnej transmisji w obu kierunkach. Magistrala składa się z dwóch linii:

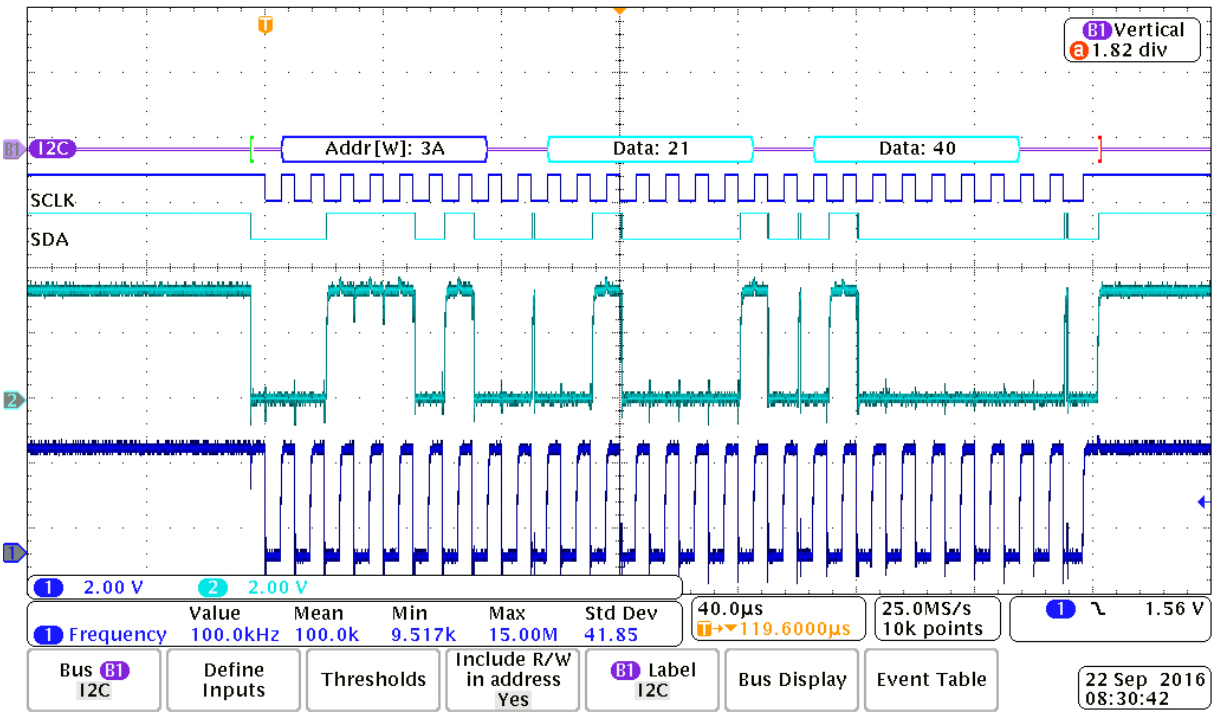
- SDA (*Serial Data Line*) – dane
- SCL (*Serial Clock Line*) – sygnał zegarowy

Układy scalone działają w systemie z otwartym kolektorem (drenem). Obie linie magistrali są połączone z napięciem zasilania przez oporniki podciągające. Układy scalone mają możliwość wymuszania tylko stanów niskich. W najprostrzym przypadku do magistrali dołączony jest jeden *Master* i kilka układów *slave*, jak na rysunku 18.13. Tylko *Master* inicjuje transmisję i on generuje sygnał zegarowy SCL. Standard I²C definiuje zasady arbitrażu, które umożliwiają dołączenie kilku układów *Master* do tej samej magistrali.

W spoczynku obie linie są w stanie wysokim. *Master* wymusza stan niski na linii SDA. Jest to znacznik początku ramki (START). Po wykryciu znacznika START, wszystkie układy przestawiają się na odbiór. Jako pierwszy, wysyłany jest adres: w dalszej transmisji będzie uczestniczył tylko układ o zgodnym adresie. Adres jest siedmio-bitowy, natomiast ostatni, ósmy bit to kierunek transmisji $R/\bar{W}$. Stan niski tego bitu oznacza zapis do układu *slave*, stan wysoki to jego odczyt. Po znaczniku START, nadajnik wymusza sygnał zegarowy na linii SCL i wartości kolejnych bitów danych na linii SDA. Bity są odczytywane w odbiorniku w stanie wysokim sygnału zegarowego, jak pokazano na rysunku 18.14. Po przesłaniu ośmiu bitów, nadajnik wystawia kolejny impuls zegarowy na SCL, ale linię danych SDA pozostawia w stanie wysokim. Odbiornik opuszcza SDA na poziom niski, co oznacza potwierdzenie odebrania bajtu i gotowość przyjęcia następnego. Jest to znacznik ACK (rysunek 18.15). Brak reakcji, gdy linia SDA pozostanie wysoka, oznacza,



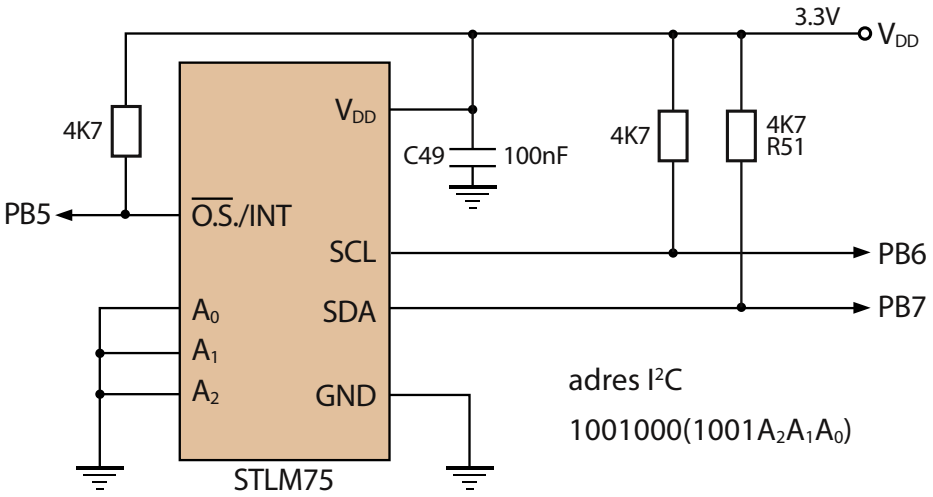
Rys. 18.15: Porównanie komunikatów ACK i NACK



Rys. 18.16: Przykład zapisu dwóch bajtów do akcelerometru LIS35

że żaden odbiornik nie nawiązał transmisji z nadajnikiem. Jest to znacznik NACK. Gdy nadajnik odbierze znacznik ACK, może kontynuować transmisję, nadając kolejny znak, albo ją zakończyć, wystawiając znacznik STOP. Jest to zmiana SDA ze stanu niskiego (po ACK), na wysoki, w czasie, gdy sygnał zegarowy SCL pozostaje wysoki (rysunek 18.15). Zauważmy, że gdy zmiany na linii danych występują przy niskim stanie linii zegarowej SCL, to mamy bity danych. Zmiany SDA w czasie gdy SCL jest wysokie, oznacza znaczniki START lub STOP.

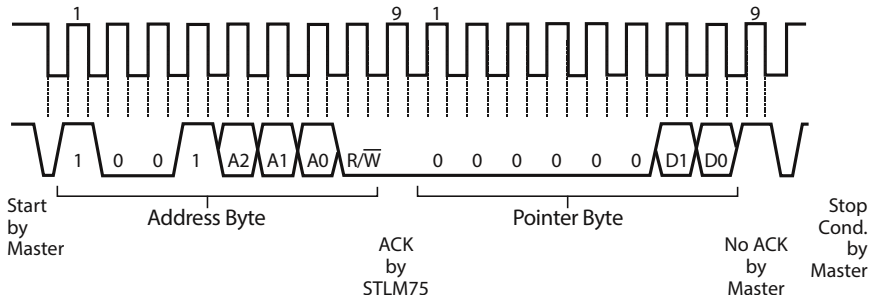
Rysunek 18.16 pokazuje przykład transmisji I^{2}C procesora z akcelerometrem LIS35 firmy STM. Adres układu LIS35 to `0x3A`. Najmłodszy bit adresu jest zerem, co oznacza, że jest to ramka zapisu. Bajt `0x21` jest numerem rejestru, kolejny `0x40` to bajt zapisywany. Widać znaczniki START i STOP.



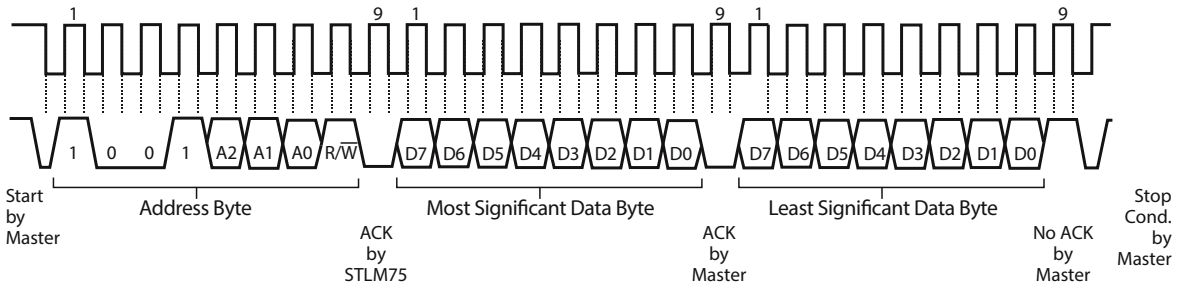
Rys. 18.17: Schemat aplikacyjny układu STLM75

18.6.1. Czujnik temperatury STLM75

W projekcie wykorzystamy płytę STM3210E-EVAL, która ma zamontowany termometr cyfrowy z wyjściem I²C, typu STLM75 firmy *STMicroelectronics*. Dokumentację układu STLM75 można pobrać ze strony producenta www.st.com. Schemat podłączenia układu STLM75 na płycie STM3210E-EVAL pokazany jest na rysunku 18.17. Układ ma zwarte do masy trzy linie adresowe, czyli siedem bitów adresu to `1001 000`. Najmłodszy, ósmy bit $R/\bar{W}$ określa kierunek transmisji. Jeżeli *Master* będzie pisał do układu STLM75, to wyśle osiem bitów adresu `0x90`, gdy będzie czytał to wyśle `0x91`.



Rys. 18.18: Transmisja I²C: zapis bajtu wskaźnika do STLM75



Rys. 18.19: Transmisja I²C: odczyt 16 bitów temperatury z STLM75

Zgodnie z dokumentacją producenta, układ STLM75 mierzy temperaturę od -55°C do +125°C, z dokładnością ±0.5°C. Zawiera cztery rejestry, z których nas interesuje tylko pierwszy: *temperature register*. Jest on 16-bitowy, tylko do odczytu, z binarną wartością zmierzonej temperatury. Adres tego rejestru, to `D1D0 = 00`. Istnieje ponadto *pointer register*, który wskazuje na jeden z rejestrów wewnętrznych. Transmisję z układem STLM75 należy więc zacząć od wpisania do niego zera, dzięki czemu *pointer register* będzie wskazywał na rejestr ze zmierzoną temperaturą. Transmisja I^{2}C, zapisu bajtu wskaźnika do STLM75 pokazana jest na rysunku 18.18. Wysyłamy dwa bajty. Pierwszy `0x90` to adres z najmłodszym bitem zerowym, co oznacza, że kolejny bajt ma być zapisany do układu *slave* STLM75. Drugi bajt `0x00`, oznacza, że *pointer register* będzie wskazywał na *temperature register*.

Teraz możemy odczytać dwa bajty, które składają się na 16-bitowy *temperature register*. Transmisja I^{2}C odczytu 16 bitów temperatury z STLM75 pokazana jest na rysunku 18.19. Wysyłamy adres `0x91`. Ostatni bit oznacza, że będziemy czytać z STLM75. Oczekujemy na ACK, który musi wystawić STLM75 na znak, że rozpoznał swój adres i jest gotowy do przekazania danych do *Master*a. Odczytujemy pierwszy bajt i wystawiamy ACK, na znak, że STLM75 może przekazać drugi bajt. Odczytujemy drugi bajt i nie wystawiamy ACK, co oznacza NACK. Znakiem STOP kończymy transmisję.

Wartość binarna temperatury, to dziewięć najstarszych bitów rejestru *temperature register*. Pierwszy bajt odebrany zawiera część całkowitą temperatury wyrażoną w stopniach Celsjusza (w kodzie uzupełnień do dwóch). Najstarszy bit drugiego bajtu ma wagę pół stopnia. Jeżeli drugi bajt ma wartość `0x80`, do wyniku należy dodać 0.5°C.

18.6.2. Projekt26_I2C

Zgodnie z tabelą 13.1, procesor STM32F103ZE ma "na pokładzie" dwa układy I^{2}C. Oba: `I2C1` oraz `I2C2` są dołączone do magistrali APB1 (rysunek 15.4), taktowanej 36 MHz. Każdy kontroler I^{2}C może pracować w trybie *slave* (domyślny), albo *Master*.

W rozdziale 13.1 utworzyliśmy katalog *Libraries*, a w nim trzy podkatalogi. W jednym z nich: *STM32_EVAL*, zawarte są programy obsługi układów zawartych w płycie ewaluacyjnej. Jest także zestaw funkcji obsługujących termometr ST-LM35.

Utworzymy nowy projekt *Projekt26_I2C* przez skopiowanie np. *Projekt14_TFTLCD* i zmianę nazwy. W grupie *StdPeriphDrv* powinien być plik *stm32f10x_i2c.c*, a w grupie *STM3210E_EVAL*, plik *stm3210e_eval_i2c_tsensor.c*. Jeśli tych plików nie ma, to należy je dołączyć.

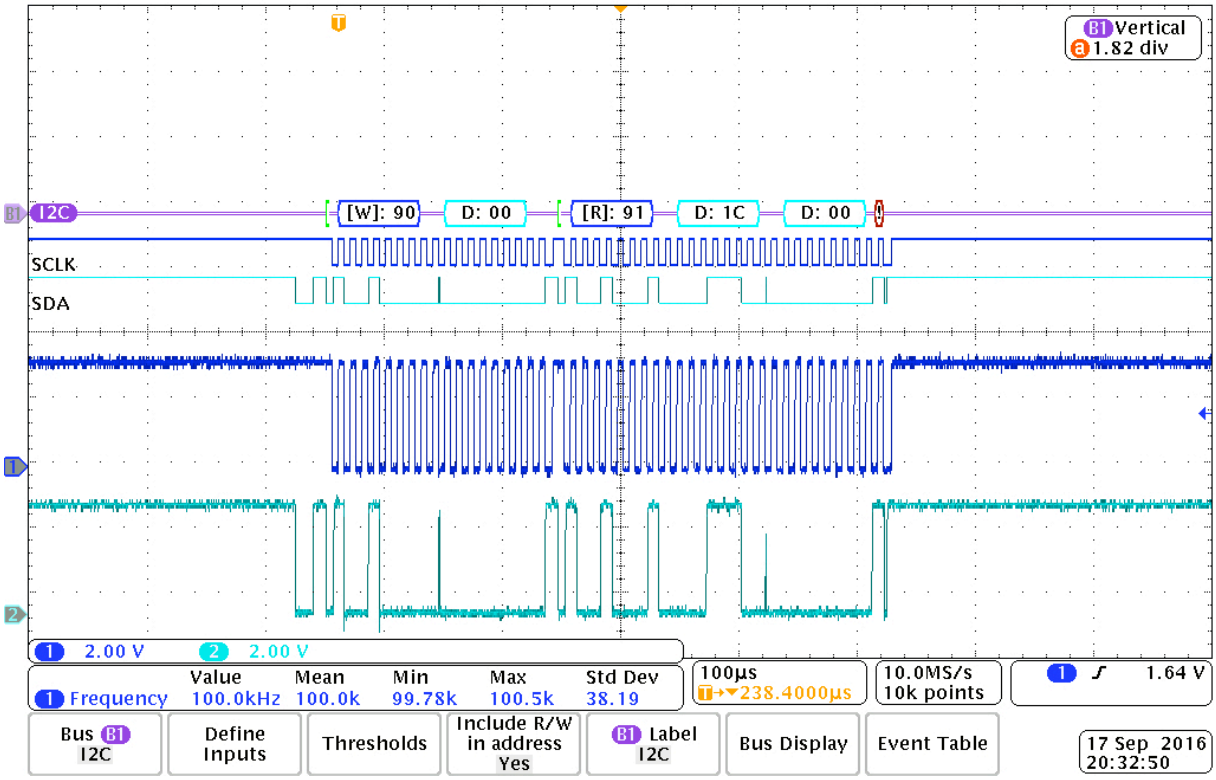
Na początku pliku *main.c* dodamy dwie dyrektywy:

```
#include "stm3210e_eval_i2c_tsensor.h"
#define  LM75_addr 0x90                // 1001 000x
```

W pętli głównej wykorzystamy tylko dwie funkcje:

```
LM75_Init();      // Initialize the Temperature Sensor
TempCelsius_Value = LM75_ReadTemp();     // Get Temperature value
```

Odczytaną wartość należy odpowiednio sformatować i wyświetlić. Możemy także przeliczyć zmierzoną temperaturę na Fahrenheity i wyświetlić w linii poniżej. Oscylogram transmisji I^{2}C odczytu temperatury z układu ST-LM75 pokazany jest na rysunku 18.20.



Rys. 18.20: Oscylogram transmisji I^{2}C z układem STLM75

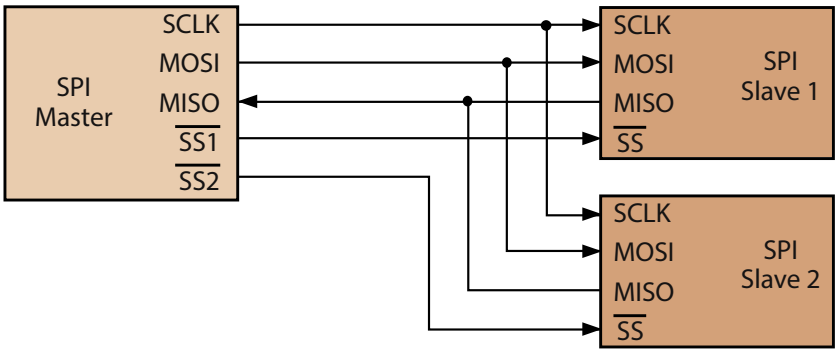
18.7. SPI Serial Peripheral Interface

Serial Peripheral Interface SPI jest standardem synchronicznej transmisji szeregowej *full duplex*, powszechnie stosowanym w systemach mikrokomputerowych. SPI jest prosty, z małym poborem energii, ale o znacznej przepustowości. Pewną niedogodnością SPI jest możliwość włączenia tylko jednego *Master*a (brak arbitrażu) i fakt, że układy *slave* nie mogą inicjować transmisji, ani wpływać na częstotliwość jej taktowania. Układ *Master* kontroluje całą wymianę danych – tylko on generuje sygnał taktujący SCLK.

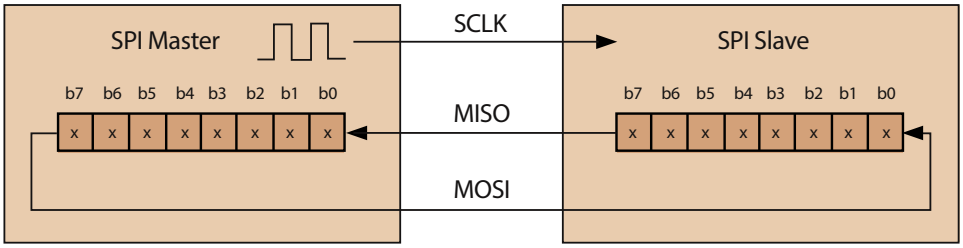
Zdefiniowane są cztery linie:

- SCLK (*Serial Clock Line*) – sygnał zegarowy
- MISO (*Master In Slave Out*) – dane *slave* → *Master*
- MOSI (*Master Out Slave In*) – dane *Master* → *slave*
- SS (*Slave Select*) – sygnał wyboru (aktywny niski)

Jak pokazano na rysunku 18.21, gdy *Master* chce wymienić dane z określonym układem *slave*, to wymusza stan niski sygnału SS wyboru właśnie tego układu. Następnie *Master* taktuje linię zegarową SCLK. Synchronicznie z zegarem SCLK, po liniach MOSI i MISO przesyłane są kolejne bity danych. Wymiana informacji następuje w obu kierunkach jednocześnie, jest więc to kanał *full duplex* – rysunek 18.22. Każdy takt zegara powoduje przesunięcie kolejki bitów o jedną pozycję.



Rys. 18.21: Układ *Master SPI* dołączony do dwóch *slave SPI*

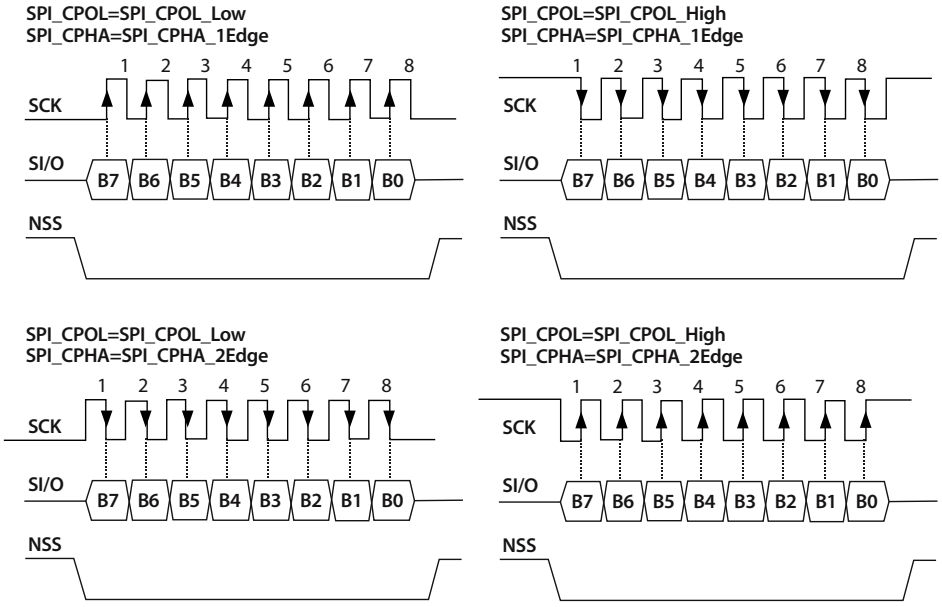


Rys. 18.22: Dwa bajty są przesyłane jednocześnie liniami MOSI i MISO

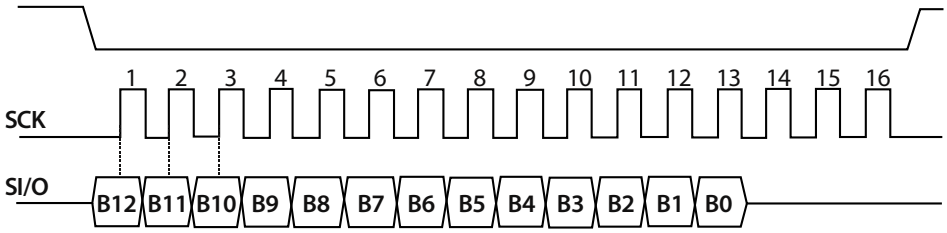
18.7.1. Projekt27_SPI

W projekcie wykorzystamy płytę ZL27ARM, która ma zamontowany termometr cyfrowy z wyjściem SPI, typu TC77 firmy *Microchip*. Dokumentację układu TC77 można pobrać ze strony producenta www.microchip.com. Zgodnie z tabelą 13.1, procesor STM32F103VB ma "na pokładzie" dwa układy SPI. Pierwszy, SPI1 jest dołączony do magistrali APB2 (rysunek 15.3), taktowanej 72 MHz. Maksymalna częstotliwość sygnału SCK to 18 MHz, co wynika z faktu, że wysłanie jednego bitu danych wymaga czterech taktów zegara. Drugi układ SPI2 jest dołączony do APB1: 36 MHz (rysunek 15.3). Dla tego układu, maksymalna częstotliwość SCK to 9 MHz. Każdy kontroler SPI może pracować w trybie *Master*, albo *slave* i obsługiwać transfer słów 8- lub 16-bitowych. Możliwa jest też komunikacja w jedną stronę z wykorzystaniem tylko jednej linii danych. Jest to przydatne w przypadku prostych układów, których nie trzeba konfigurować i wystarczy tylko odczyt danych. Kontrolery SPI pozwalają na określenie, czy aktywne jest zbocze narastające sygnału zegarowego, czy opadające i czy pierwsze, czy drugie. Łącznie daje to cztery sposoby generowania SCLK, jak pokazano na rysunku 18.23. Możliwy jest także wybór, czy dane są przesyłane od najstarszego, czy od najmłodszego bitu. Oczywiście kontroler SPI należy skonfigurować odpowiednio do układu z którym będzie współpracował. Każdy kontroler SPI ma wbudowane dwa układy do obliczania sum kontrolnych: jeden dla danych odbieranych i drugi dla nadawanych. Stosuje się je głównie w czasie komunikacji z zewnętrznymi układami pamięci. W mikrokontrolerach STM32 *high-density*, czyli np. w STM32F103ZE (płyta STM3210E-EVAL), kontrolery SPI mogą obsługiwać transmisję **I^{2}S** (*Integrated Interchip Sound*) używaną do komunikacji z urządzeniami audio.

Rysunek 18.24 został utworzony na podstawie dokumentacji układu TC77. Wynika z niego, że przy braku taransmisji, linia SCLK powinna być w stanie niskim (`SPI_CPOL_Low`), a odczyt stanu przesyłanego bitu następuje w czasie pierwszego zbocza narastającego



Rys. 18.23: Możliwe konfiguracje sygnału zegarowego SPI



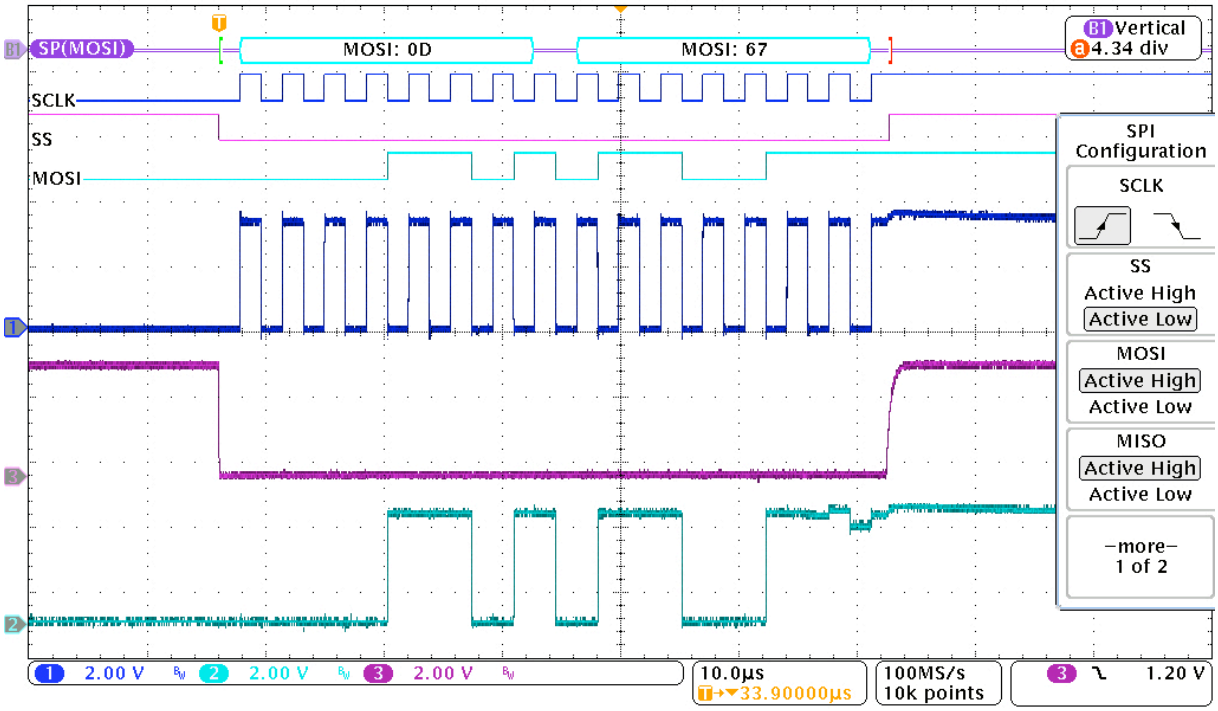
Rys. 18.24: Odczyt temperatury z układu TC77 *Microchip*

(`SPI_CPHA_1Edge`). Do książki dołączony jest *Projekt27_SPI* w którym wykonywany jest odczyt temperatury z TC77 i wyświetlany jest wynik, po przeliczeniu na °C. Oscylogram transmisji SPI pokazany jest na rysunku 18.25.

Przed uruchomieniem programu należy wykonać połączenia na płycie, zgodnie z tabelą 18.4. Sygnały z układu TC77 są wyprowadzone na złącze JP7, jak pokazano na schemacie w dodatku A i należy je dołączyć do układu SPI1.

Tabela 18.4: Połączenia na płycie ZL27ARM pomiędzy TC77 i SPI1

układ TC77	**złącze JP7**	**port GPIOA**	**SPI1**	**złącze JP6**
CS	2	PA4	SPI1_NSS	5
SI/O	4	PA7	SPI1_MOSI	8
SCK	6	PA5	SPI1_SCK	6



Rys. 18.25: Oscylogram odczytu z TC77 *Microchip*

Rozdział 19

Obiekt regulacji

Utworzymy dwa obiekty: oscylacyjny i wieloinercyjny z opóźnieniem. Zakładamy okres próbkowania (*sampling time*) $T_s = 50\ ms$. To oznacza 20 interwencji w sekundzie.

19.1. Obiekt oscylacyjny

Transmitancja operatorowa obiektu oscylacyjnego dana jest wzorem

$$G(s) = \frac{k\omega^2}{s^2 + 2\zeta\omega s + \omega^2}$$

gdzie k to wzmocnienie, ω to pulsacja drgań nietłumionych, a ζ to współczynnik tłumienia. Oscylacje wystąpią, gdy dwumian w mianowniku będzie miał bieguny zespolone sprzężone, czyli wyróżnik dwumianu Δ musi być mniejszy od zera

$$\Delta = 4\zeta^2\omega^2 - 4\omega^2 = 4\omega^2(\zeta^2 - 1)$$

Warunkiem oscylacji jest więc $0 < \zeta < 1$. Dolne ograniczenie wynika z warunku stabilności. W naszym obiekcie przyjmiemy:

$$k = 1, \qquad \omega = 1, \qquad \zeta = 0.2$$

Pozostaje wykonanie emulacji: czyli zamienimy transmitancję Laplace'a na transmitancję dyskretną $\mathcal{Z}$. Wykorzystamy do tego środowisko MatLab.

```
Ts=0.05;     % 50 ms
wn=1; zeta=0.2; s=tf('s');
Gplant1=wn*wn/(s^2+2*zeta*wn*s+wn^2);
% Obiekt: czas dyskretny
Gplant1D=c2d(Gplant1,Ts,'matched');     % matched pole-zero method
[num,den,Ts]=tfdata(Gplant1D); b=num{1,1}; a=den{1,1};  % parametry modelu
```

Uzyskane wyniki:

$$\begin{aligned} b_0 &= 0 & a_0 &= 1 \\ b_1 &= 0.00123732511175664 & a_1 &= -1.97772402308324 \\ b_2 &= 0.00123732511175664 & a_2 &= 0.980198673306755 \end{aligned}$$

Transmitancja dyskretna

$$\frac{Y(z)}{U(z)} = \frac{b_1 z^{-1} + b_2 z^{-2}}{1 + a_1 z^{-1} + a_2 z^{-2}}$$

Równanie różnicowe

$$y(k) = -a_1 y(k-1) - a_2 y(k-2) + b_1 u(k-1) + b_2 u(k-2)$$

19.2. Obiekt wieloinercyjny z opóźnieniem

Transmitancja operatorowa obiektu bez opóźnienia

$$G(s) = \frac{12}{(s+1)^3 \cdot (s+3) \cdot (s+4)}$$

Zamienimy transmitancję Laplace'a na transmitancję dyskretną $\mathcal{Z}$.

```
Ts=0.05;     % 50 ms
s=tf('s');
Gplant2=12/((s+1)^3*(s+3)*(s+4));
% Obiekt: czas dyskretny
Gplant2D=c2d(Gplant2,Ts,'matched');     % matched pole-zero method
[num,den,Ts]=tfdata(Gplant2D); b=num{1,1}; a=den{1,1};  % parametry modelu
```

Uzyskane wyniki:

$$\begin{array}{ll}
b_0 = 0 & a_0 = 1 \\
b_1 = 1.83064459030329e-07 & a_1 = -4.53312700300518 \\
b_2 = 7.32257836121317e-07 & a_2 = 8.21179495227475 \\
b_3 = 1.09838675418198e-06 & a_3 = -7.43052512579135 \\
b_4 = 7.32257836121317e-07 & a_4 = 3.35839076526575 \\
b_5 = 1.83064459030329e-07 & a_5 = -0.606530659712634
\end{array}$$

Transmitancja dyskretna

$$\frac{Y(z)}{U(z)} = \frac{b_1 z^{-1} + b_2 z^{-2} + b_3 z^{-3} + b_4 z^{-4} + b_5 z^{-5}}{1 + a_1 z^{-1} + a_2 z^{-2} + a_3 z^{-3} + a_4 z^{-4} + a_5 z^{-5}}$$

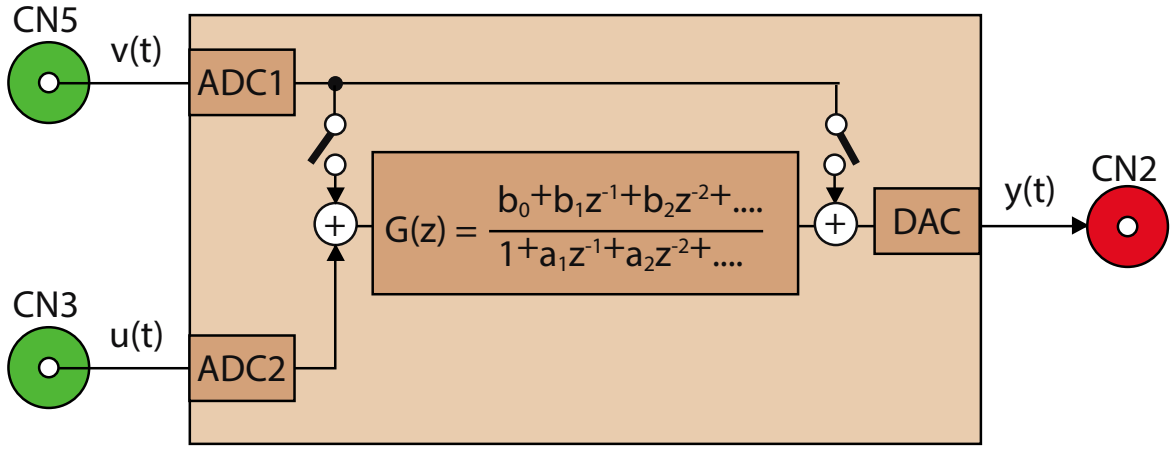
Do równania różnicowego możemy dodać opóźnienie $\tau = T_s \cdot delay$

$$y(k) = -\sum_{i=1}^{5} a_i y(k-i) + \sum_{i=1}^{5} b_i u(k-i-delay)$$

19.3. Projekt28_obiekt

Utworzymy teraz projekt, w którym wykonamy implementację obu opisanych obiektów. Obie realizacje są identyczne, różnią się tylko ilością i wartościami parametrów a_i, b_i. Schemat implementacji pokazany jest na rysunku 19.1. Gniazdo CN3 jest wejściem, a CN2 wyjściem. Do gniazda CN5 będziemy dołączali zakłócenie, które będzie można dodać do wejścia, albo do wyjścia obiektu. W pliku *main.c* zdefiniujemy zmienne i stałe globalne

```
// zmienne globalne -------------------------------------------
unsigned char txt2LCD[20] = {0};     // bufor funkcji sprintf
unsigned int k = 0;                  // licznik do wyświetlania
float U[3]={0.0}, Y[3]={0.0};        // input / output
const float A[2]={ -1.97772402308324, // Ts = 50 milisekund
                   0.980198673306755 };
const float B[2]={  0.00123732511175664,
                    0.00123732511175664 } ;
```



Rys. 19.1: Schemat implementacji obiektu SISO

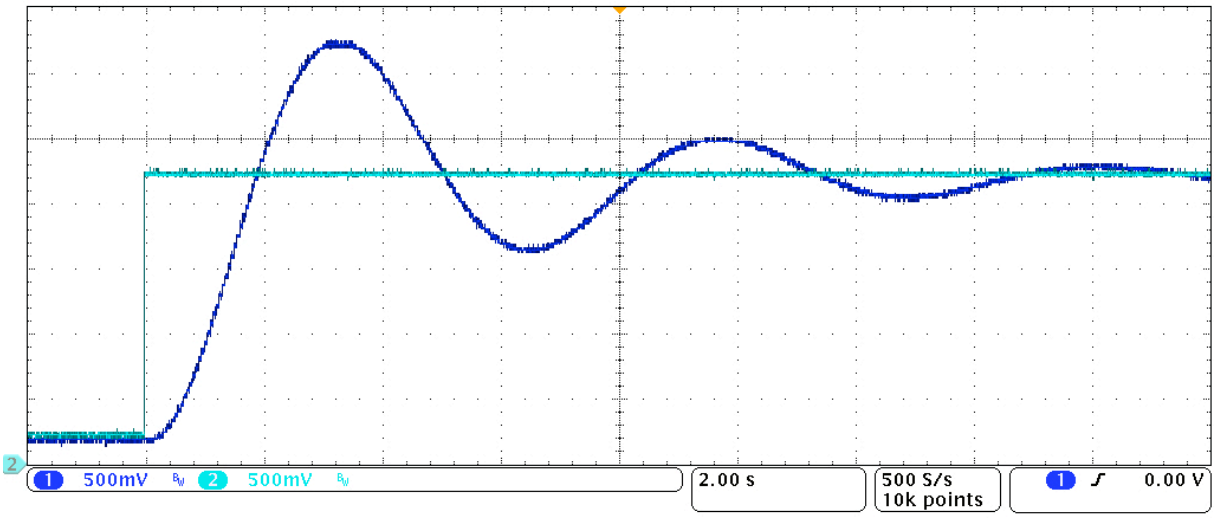
Obliczenia wykonamy w obsłudze przerwania systemowego SysTick.

```
extern unsigned char txt2LCD[]; // bufor funkcji sprintf
extern float U[3], Y[3];        // input / output
extern const float A[5], B[5];
extern unsigned int k;
void SysTick_Handler(void){
   unsigned int i;
   int u_bin, y_bin;
   float y, volt;
   GPIO_Set_Bits(GPIOF, GPIO_Pin_9);    // zapal LED niebieski
   // wystaw wyjście z poprzedniego cyklu, z ograniczeniem do zakresu
   y_bin = (int) Y[0] + 2048;
   if(y_bin > 3846)
     y_bin = 3846;
   else if(y_bin < 250)
     y_bin = 250;
   DAC_SetChannel1Data(DAC_Align_12b_R, (uint16_t) y_bin);
   // przetwornik A/D - pomiar z poprzedniego cyklu
   u_bin = ADC_GetConversionValue(ADC2);// wartość binarna odczytana z ADC2
   ADC_SoftwareStartConvCmd(ADC2, ENABLE);// rozpocznij przetwarzanie ADC2
   for(i=2; i>0; i--)
     U[i] = U[i-1];        // przesuwamy kolejkę
   U[0] = (float) u_bin - 2048;
   y = 0.0;
   for(i=0; i<2; i++){
     y -= A[i]*Y[i];
     y += B[i]*U[i];
   }
   for(i=2; i>0; i--)
     Y[i] = Y[i-1];        // przesuwamy kolejkę
   Y[0] = y;
   y_bin = (int) y + 2048; // y_bin z aktualnego cyklu
```

```
  // wyświetlamy bez ograniczenia
  if(k >= 320)
    k = 0;
  volt = u_bin * 0.0008059;      // przeliczone na napięcie
  LCD_SetTextColor(Blue);        // kolor textu
  sprintf((char *)txt2LCD,"CN3 U = %f [V]",volt);
  LCD_DisplayStringLine(Line8, (uint8_t *)txt2LCD);
  volt = y_bin * 0.0008059;      // przeliczone na napięcie
  LCD_SetTextColor(Green);       // kolor textu
  sprintf((char *)txt2LCD,"CN2 Y = %f [V]",volt);
  LCD_DisplayStringLine(Line9, (uint8_t *)txt2LCD);
  LCD_SetTextColor(Black);     // kasujemy poprzedni punkt
  LCD_DrawLine(0, 319-k, 192, LCD_DIR_VERTICAL);
  LCD_SetTextColor(Blue);      // kolor linii
  LCD_DrawLine(186-u_bin/22, 319-k, 1, LCD_DIR_HORIZONTAL);
  LCD_SetTextColor(Green);     // kolor linii
  LCD_DrawLine(186-y_bin/22, 319-k, 1, LCD_DIR_HORIZONTAL);
  k++;
  GPIO_Reset_Bits(GPIOF, GPIO_Pin_9);     // zgaś LED niebieski
}
```

Na początku obsługi przerwania zapalamy, a na końcu gasimy niebieski LED. Mierząc ten sygnał możemy określić czas wykonywania programu obsługi, co pozwala oszacować obciążenie procesora.

Pomiar jest z poprzedniego cyklu, wynik wystawiany jest w następnym. Aplikacja wprowadza więc opóźnienie dwóch taktów. Rysunek 19.2 przedstawia odpowiedź na skok obiektu oscylacyjnego.



Rys. 19.2: Odpowiedź na skok obiektu SISO oscylacyjnego

Rozdział 20

Cyfrowy regulator PID

20.1. Podstawowy schemat układu regulacji

Obiekt regulacji jest sterowany sygnałem $u(t)$, a wyjście obiektu $y(t)$ jest wielkością regulowaną. Regulator na podstawie uchybu $e(t)$ generuje sygnał sterujący $u(t)$, rysunek 20.1. Wartość zadana wielkości regulowanej to $y^{zad}(t)$. Celem jest utrzymywanie wielkości regulowanej $y(t)$ równej wielkości zadanej $y^{zad}(t)$, czyli dążymy do równości $y(t) = y^{zad}(t)$. Najprostszym rozwiązaniem jest regulator proporcjonalny, który jest zwykłym wzmacniaczem o wzmocnieniu K_r. Załóżmy że obiekt też jest prostym wzmacniaczem, ale o wzmocnieniu K_o. Przyjmijmy, że $K_o = 9$, $K_r = 1$ oraz $y^{zad} = 20$.

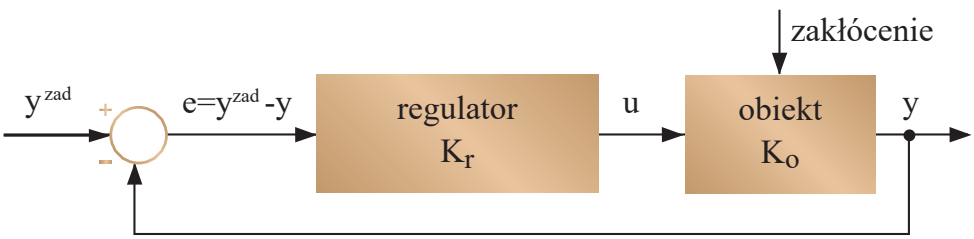
$$y = K_o u = K_o(K_r e) = K_o K_r (y^{zad} - y) = K_o K_r y^{zad} - K_o K_r y$$

$$y + K_o K_r y = y(1 + K_o K_r) = K_o K_r y^{zad} \implies y = \frac{K_o K_r}{1 + K_o K_r} y^{zad} = 0.9 y^{zad} = 18$$

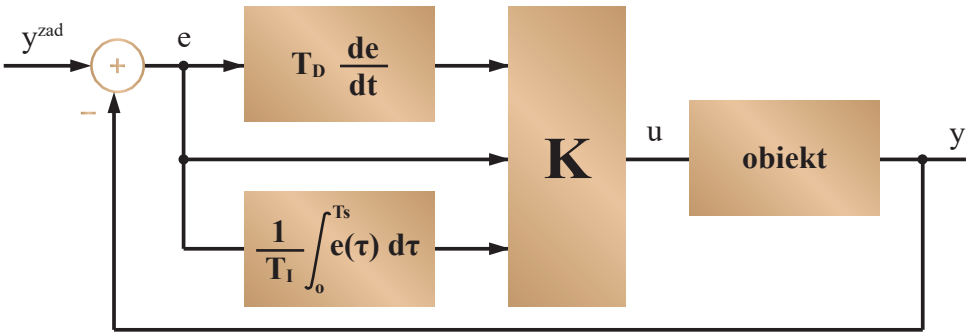
Różnica pomiędzy wartością zadaną y^{zad} a wyjściem obiektu, czyli **uchyb statyczny**, wynika z zasady działania układu ze sprzężeniem zwrotnym: regulator proporcjonalny „karmi się uchybem” – wyjście regulatora jest różne od zera tylko wtedy, gdy na wejściu jest niezerowy uchyb.

Sposobem na liwidację uchybu statycznego jest zastosowanie regulatora z członem całkującym. Całkowanie umożliwia likwidację uchybu statycznego, ale pogarsza stabilność. Dalsza poprawa działania może być osiągnięta przez dodanie członu różniczkującego. Różniczkowanie rozszerza pasmo przenoszonych sygnałów, czyli poprawia dynamikę układu regulacji oraz może poprawić stabilność członu całkującego. Uzyskaliśmy regulator PID (*Proportional Integral Derivative controller*) o równaniu w dziedzinie czasu ciągłego:

$$u(t) = K\left[e(t) + \frac{1}{T_I}\int_0^T e(\tau)\mathrm{d}\tau + T_D \frac{\mathrm{d}e(t)}{\mathrm{d}t}\right] \tag{20.1}$$



Rys. 20.1: Podstawowy schemat układu regulacji



Rys. 20.2: Struktura równoległa regulatora PID

20.1.1. Transmitancja operatorowa regulatora PID

Równanie (20.1) możemy zrealizować jako strukturę równoległą, jak na rysunku 20.2. Możemy także przedstawić je w postaci transmitancji operatorowej. Jest to stosunek transformaty Laplace'a sygnału wyjściowego do transformaty Laplace'a sygnału wejściowego, przy zerowych warunkach początkowych. Pamiętając, że operatorem całkowania jest $1/s$, a operatorem różniczkowania s, mamy:

$$\frac{U(s)}{E(s)} = G_{PID}(s) = K_P\left[1 + \frac{1}{T_I s} + T_D s\right] \tag{20.2}$$

W obu wzorach: (20.1) i (20.2), T_I to stała czasowa całkowania (czas zdwojenia), T_D to stała czasowa różniczkowania (czas wyprzedzenia). Transmitancja członu różniczkującego w takiej postaci jak we wzorze (20.2) nie jest realizowalna, gdyż stopień licznika jest wyższy od stopnia mianownika. W dalszej części rozdziału rozwiążemy ten problem.

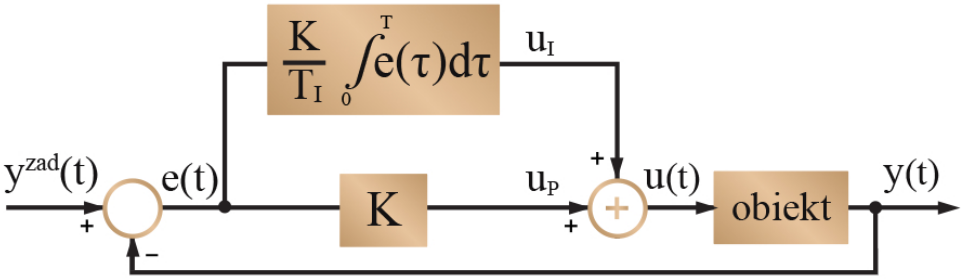
20.2. Cyfrowa realizacja algorytmu PI

Zaczniemy od uproszczonej wersji, czyli regulatora PI. Wzór (20.1) zostaje uproszczony do postaci

$$u(t) = K\left[e(t) + \frac{1}{T_I}\int_0^T e(\tau)\mathrm{d}\tau\right]$$

Zakładamy, że nasz układ regulacji jest systemem przyczynowym. Zarówno wyjścia obiektu, jak i regulatora zależą od wejść bieżących i przeszłych, czyli nie wykazują reakcji, zanim nie nastąpi pobudzenie. Zakładamy ponadto, że wszystkie sygnały przed chwilą $k = 0$ są zerowe. W przypadku obiektu, jeżeli pobudzenie $u(k < 0) = 0$, to także wyjście $y(k < 0) = 0$. W przypadku regulatora $u(k < 0) = 0$. Zakładamy ponadto, że wartość zadana może stać się niezerowa dopiero w chwili zerowej, czyli $y^{zad}(k < 0) = 0$. To oznacza, że także uchyb spełnia warunek przczynowości $e(k < 0) = 0$. W praktyce obiekt i regulator wystartujemy "od zera", czyli wszystkie zmienne pamiętane (przeszłe wartości sygnałów) zostaną zainicjowane zerami. Algorytm regulacji bezpośredniej (*Direct Digital Control*) DDC można zrealizować jako

- algorytm pozycyjny (położeniowy) – oblicza wartość absolutną sygnału sterującego element wykonawczy, na daną chwilę czasową,
- algorytm przyrostowy – określający jedynie przyrosty w stosunku do położenia poprzedniego, stosowany gdy element wykonawczy zawiera całkowanie.



Rys. 20.3: Układ regulacji PI

Jedna z możliwych realizacji to cyfrowy algorytm pozycyjny w strukturze równoległej, w którym całkowanie zastąpiono sumowaniem. Wzmocnienie proporcjonalne K włączymy części całkującej. Wyjście części proporcjonalnej oznaczymy jako $u_P(k)$, a całkującej jako $u_I(k)$. Wyjście regulatora jest sumą tych dwóch sygnałów, jak pokazano na rysunku 20.3.

20.2.1. Składowa proporcjonalna algorytmu

Ta część algorytmu polega na mnożeniu sygnału uchybu przez wzmocnienie proporcjonalne

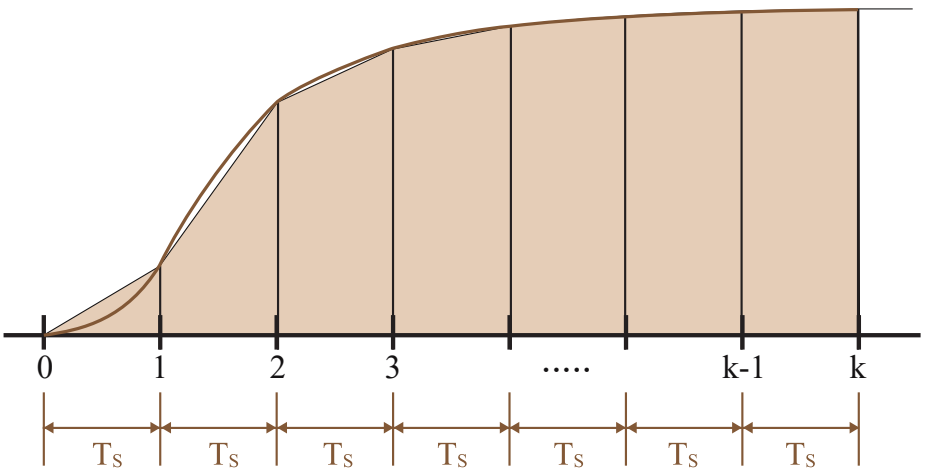
$$u_P(k) = K \cdot e(k)$$

20.2.2. Składowa całkowa algorytmu

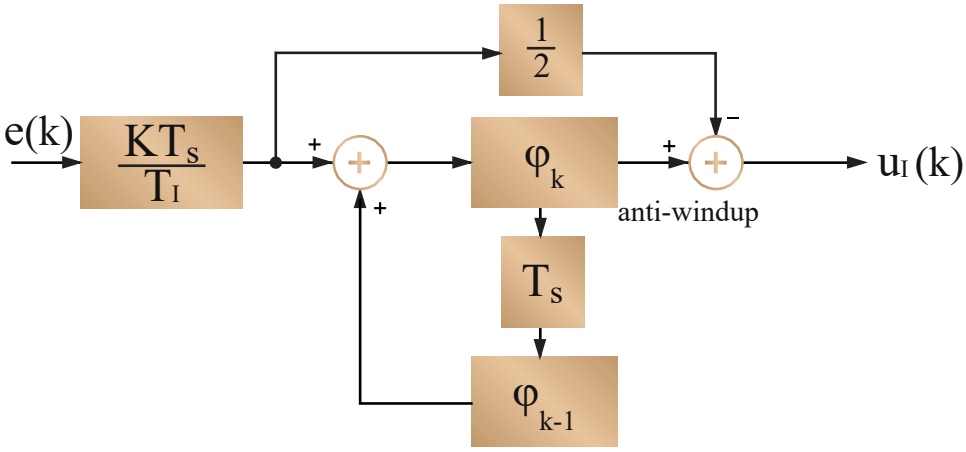
Całkowanie od chwili zerowej do bieżącej zastąpimy sumowaniem z wykorzystaniem metody trapezów (rys 20.4). Powierzchnia trapezu to iloczyn szerokości podstawy T_s (okres interwencji) i średniej arytmetycznej wysokości boków $(e(k) + e(k-1))/2$.

$$\begin{aligned} u_I(k) &= \frac{K}{T_I}\sum_{i=1}^{k} T_s \frac{e(i)+e(i-1)}{2} \\ &= \frac{K}{T_I}\left[T_s\frac{e(1)+0}{2} + T_s\frac{e(2)+e(1)}{2} + \ldots + T_s\frac{e(k-1)+e(k-2)}{2} + T_s\frac{e(k)+e(k-1)}{2}\right] \\ &= \frac{KT_s}{T_I}\left[\frac{e(k)}{2} + \sum_{i=1}^{k-1} e(i)\right] = \frac{KT_s}{T_I}\left[\sum_{i=1}^{k} e(i) - \frac{e(k)}{2}\right] \end{aligned}$$

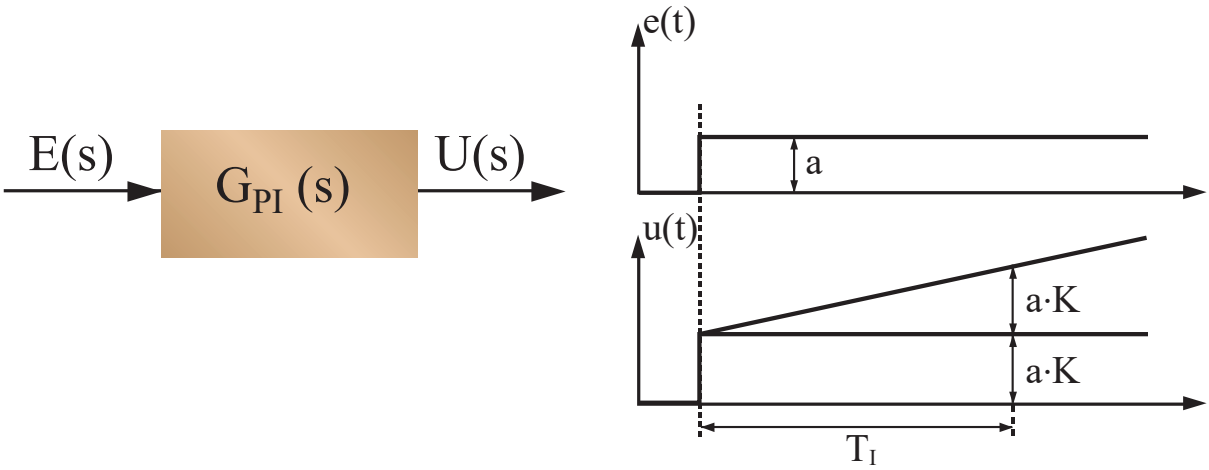
Schemat realizacji składowej całkowej algorytmu pokazany jest na rysunku 20.5. Jeżeli z powodu (na przykład) awarii, obiekt przez dłuższy czas nie odpowiada na sterowanie, to uchyb jest duży i często niezmienny. Układ całkujący wewnątrz regulatora osiągnie bardzo dużą wartość. Po usunięciu awarii obiekt znów stanie się sterowalny i uchyb zmaleje do mniejszej wartości. Ta mała wartość uchybu spowoduje, że "odcałkowywanie" wartości uzyskanej w trakcie awarii będzie bardzo długo trwało. To może spowodować, że układ regulacji długo będzie utrzymywał niezerowy uchyb. Takie niekorzystne zjawisko nazywane jest *windup* – "nawijanie". Najprostszym działaniem zapobiegawczym (*anti-windup*) jest całkowanie z nasyceniem. Dlatego ograniczymy wartość zmiennej φ_k (na rysunku 20.5) do zakresu wyjściowego przetwornika DAC.



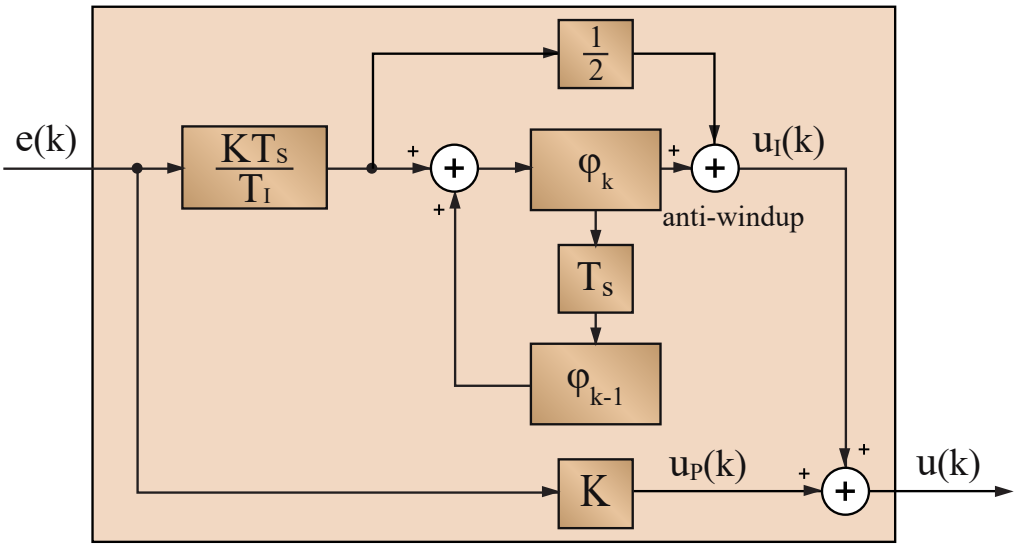
Rys. 20.4: Całkowanie metodą trapezów



Rys. 20.5: Realizacja składowej całkowej algorytmu PID



Rys. 20.6: Regulator PI – czas zdwojenia



Rys. 20.7: Schemat algorytmu pozycyjnego PI, równoległego

20.2.3. Czas zdwojenia T_I

Czas zdwojenia T_I to czas potrzebny na osiągnięcie przez sygnał wyjściowy (sterujący) regulatora PI, podwojonej wartości sygnału wyjściowego wynikającego z działania proporcjonalnego, przy założeniu skokowego sygnału wejściowego. Ta oficjalna definicja oznacza, że po czasie T_I człon całkujący doda taki sam sygnał jak człon proporcjonalny (czyli nastąpi podwojenie wyjścia), w odpowiedzi na skokową zmianę uchybu na wejściu regulatora. Ilustracją definicji czasu zdwojenia jest rysunek 20.6.

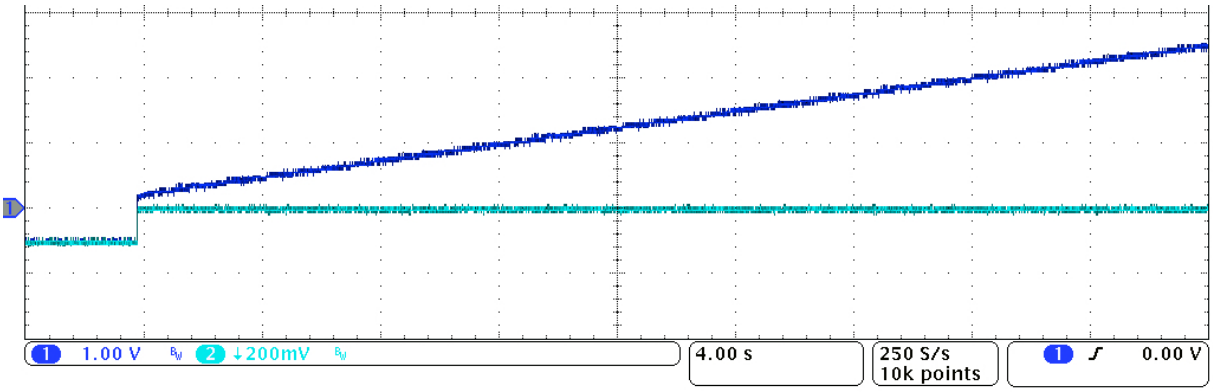
$$e(t) = a \cdot 1(t) \qquad \rightarrow \qquad E(s) = \frac{a}{s}, \qquad G_{PI}(s) = K\Big[1 + \frac{1}{T_I s}\Big]$$

$$U(s) = G_{PI} \cdot E(s) = K\Big[1 + \frac{1}{T_I s}\Big] \cdot \frac{a}{s} = \frac{K \cdot a}{s} + \frac{K \cdot a}{T_I s^2}$$

$$u(t) = K \cdot a + \frac{K \cdot a}{T_I} t = K \cdot a\Big(1 + \frac{1}{T_I} t\Big) \qquad \Rightarrow \qquad u(T_I) = K \cdot a \cdot 2$$

20.2.4. Projekt29_PI

Sposób realizacji regulatora PI pokazany jest na rysunku 20.14. Zakres wyjścia przetwornika DAC z włączonym wzmacniaczem wyjściowym, to binarnie $[250, 3846]$. W przypadku przesunięcia zakresu o 2048, zakres jest równy $[-1798, +1798]$. Takie ograniczenie nałożymy na wszystkie sygnały oraz zmienną φ_k, która jest bezpośrednio dołączona do sygnału $u_I(k)$. Zapewni to nam zerowy poziom nasycenia części całkującej regulatora. Rysunek 20.8 przedstawia odpowiedź regulatora PI na skokową zmianę uchybu $e(t)$. Silne całkowanie (z krótkim czasem zdwojenia) prowadzi do szybkiej likwidacji uchybu statycznego. Ale z drugiej strony może być przyczyną niestabilności. Można to wyjaśnić na przykładzie steru kierunku samolotu. Jeżeli wystąpi odchylenie od kierunku lotu, to zostanie ono scałkowane do dużej wartości, w wyniku czego ster kierunku zostanie wychylony silnie w przeciwną stronę. Samolot zareaguje gwałtownie i powstanie uchyb o przeciwnym znaku. Ten uchyb spowoduje gwałtowne odcałkowanie poprzedniej wartości i scałkowanie uchybu z nowym znakiem. Efektem będzie opóźnione, ale silne wychylenie steru kierunku

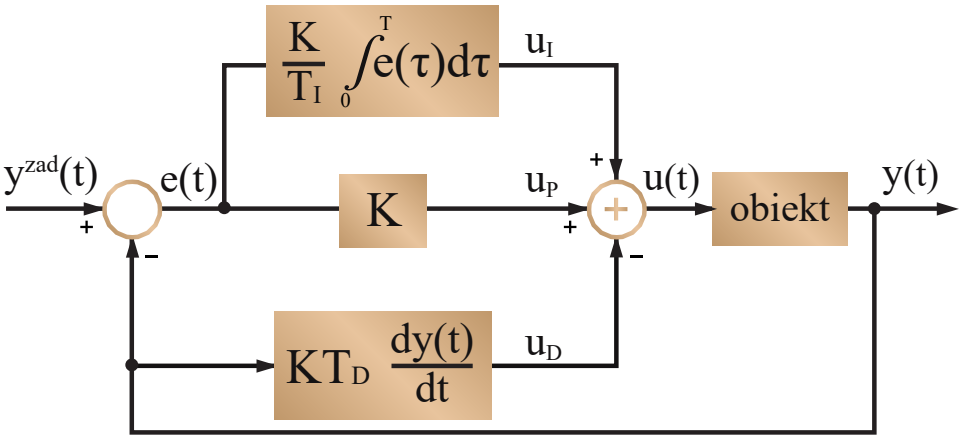


Rys. 20.8: Odpowiedź regulatora PI na skokową zmianę uchybu

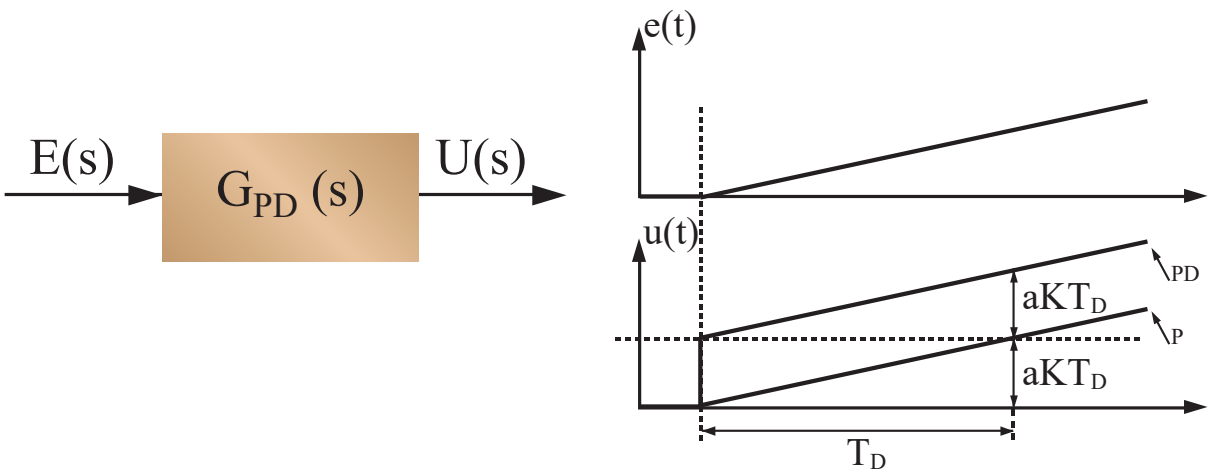
przeciwne do poprzedniego. Uzyskamy oscylacje kierunku lotu, czyli niestabilne zachowanie samolotu. Te oscylacjie powstały w wyniku całkowania źle dobranego w stosunku do dynamiki obiektu. Rozwiązaniem jest osłabienie całkowania przez zwiększenie czasu zdwojenia. Ale za to zapłacimy wydłużeniem czasu redukcji uchybu statycznego. Możemy jednak uzyskać dobry wynik wprowadzając człon różniczkujący, który będzie reagował na zmiany wartości regulowanej i przeciwstawi się oscylacjom kierunku lotu samolotu.

20.3. Cyfrowa realizacja algorytmu PID

Regulator PID (20.1) zrealizujemy także jako cyfrowy algorytm pozycyjny w strukturze równoległej, w którym całkowanie zastąpimy sumowaniem, a różniczkowanie to będzie różnica wsteczna. Wzmocnienie proporcjonalne K włączymy do poszczególnych części. Wyjście części proporcjonalnej oznaczymy jako $u_P(k)$, całkującej jako $u_I(k)$, różniczkującej jako $u_D(k)$. Wyjście regulatora jest sumą tych trzech sygnałów, jak pokazano na rysunku 20.9. Kolejna modyfikacja struktury to dołączenie na wejście części różniczkującej sygnału regulowanego y(t) zamiast uchybu. Dzięki temu regulator nie będzie reagował gwałtowną zmianą sygnału sterującego $u(t)$ w chwili zmiany wartości zadanej $y^{zad}(t)$.



Rys. 20.9: Układ regulacji PID



Rys. 20.10: Regulator PD – czas wyprzedzenia

20.3.1. Czas wyprzedzenia T_D

Człon całkujący reaguje na uchyb statyczny, natomiast człon różniczkujący na zmiany uchybu. Dlatego do zdefiniowania parametru różniczkowania T_D potrzebny jest sygnał zmienny w czasie.

Czas wyprzedzenia T_D to czas potrzebny na osiągnięcie przez sygnał wyjściowy (sterujący) regulatora PD, podwojonej wartości sygnału wyjściowego wynikającego z działania różniczkującego, przy założeniu narastającego liniowo sygnału wejściowego.

$$e(t) = a \cdot 1(t) \cdot t \qquad \rightarrow \qquad E(s) = \frac{a}{s^2}, \qquad G_{PD}(s) = K\Big[1 + T_D s\Big]$$

$$U(s) = G_{PD} \cdot E(s) = K\Big[1 + T_D s\Big] \cdot \frac{a}{s^2} = \frac{K \cdot a}{s^2} + \frac{K \cdot a \cdot T_D}{s}$$

$$u(t) = K \cdot a \cdot t + K \cdot a \cdot T_D = K \cdot a\Big(t + T_D\Big) \quad \Rightarrow \quad u(T_D) = 2(K \cdot a \cdot T_D)$$

20.3.2. Składowa różniczkowa algorytmu

Wzmocnienie amplitudy sygnału układu różniczkującego idealnego (o transmitancji $T_D s$) rośnie do nieskończoności wraz ze wzrostem częstotliwości. Wiemy, że wszystkie układy rzeczywiste mają ograniczone pasmo przenoszenia. To oznacza, że układ o transmitancji $T_D s$ nie jest realizowalny fizycznie (stopień licznika transmitancji jest wyższy od stopnia mianownika). Dlatego składową różniczkową zrealizujemy z wykorzystaniem rzeczywistego układu różniczkującego, który ma charakter różniczkująco–inercyjny. Opisany jest on równaniem

$$\alpha \frac{u_D(t)}{dt} + u_D(t) = T_D \frac{de(t)}{dt} \tag{20.3}$$

gdzie α to stała czasowa inercji, a T_D to czas wyprzedzenia. W celu wyznaczenia transmitancji układu, różniczkowanie we wzorze (20.3) zastępujemy operatorem s.

$$\alpha U_D(s)s + U_D(s) = T_D E(s)s \quad \rightarrow \quad G_D(s) = \frac{U_D(s)}{E(s)} = \frac{T_D s}{\alpha s + 1} = \frac{T_D}{\alpha + 1/s} \tag{20.4}$$

Jak widać wzmocnienie układu rzeczywistego jest ograniczone i dąży do wartości T_D/α.

Aproksymację układu różniczkowania wykonamy metodą Eulera, która wyprowadzona została z następującej definicji różniczki

$$\dot{x} = \lim_{\partial t \to 0} \frac{\partial x}{\partial t}$$

Istnieją dwie odmiany metody Eulera

- aproksymacja prostokątna w przód (*forward rectangular rule*)

$$\dot{x} \approx \frac{x(k+1) - x(k)}{T_s}$$

- aproksymacja prostokątna w tył (*backward rectangular rule*)

$$\dot{x} \approx \frac{x(k) - x(k-1)}{T_s}$$

W rzeczywistym układzie nie będziemy znali wartości próbki z przyszłości $x(k+1)$, więc musimy zastosować regułę wstecz. Przybliżenie dyskretne równania (20.3) jest więc następujące

$$\alpha \frac{u_D(k) - u_D(k-1)}{T_s} + u_D(k) = T_D \frac{e(k) - e(k-1)}{T_s} \tag{20.5}$$

Równanie (20.5) pomnożymy obustronnie przez T_s i podzielimy przez α

$$u_D(k) - u_D(k-1) + \frac{T_s}{\alpha} u_D(k) = \frac{T_D}{\alpha} e(k) - \frac{T_D}{\alpha} e(k-1) \tag{20.6}$$

Wzór (20.6) napiszemy dla kolejnych chwil czasowych od chwili obecnej k do początku

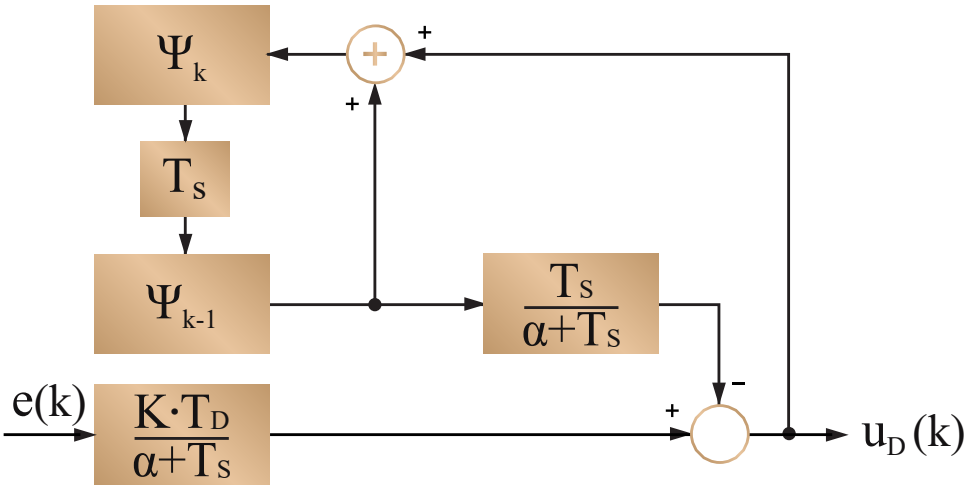
$$\begin{array}{lclclcl}
u_D(k) & - & u_D(k-1) & + & \frac{T_s}{\alpha} u_D(k) & = & \frac{T_D}{\alpha} e(k) & - & \frac{T_D}{\alpha} e(k-1) \\
u_D(k-1) & - & u_D(k-2) & + & \frac{T_s}{\alpha} u_D(k-1) & = & \frac{T_D}{\alpha} e(k-1) & - & \frac{T_D}{\alpha} e(k-2) \\
u_D(k-2) & - & u_D(k-3) & + & \frac{T_s}{\alpha} u_D(k-2) & = & \frac{T_D}{\alpha} e(k-2) & - & \frac{T_D}{\alpha} e(k-3) \\
\dots & & \dots & & \dots & & \dots & & \dots \\
u_D(2) & - & u_D(1) & + & \frac{T_s}{\alpha} u_D(2) & = & \frac{T_D}{\alpha} e(2) & - & \frac{T_D}{\alpha} e(1) \\
u_D(1) & - & 0 & + & \frac{T_s}{\alpha} u_D(1) & = & \frac{T_D}{\alpha} e(1) & - & 0
\end{array}$$

Ostatnie równania dodamy stronami

$$u_D(k) + \frac{T_s}{\alpha} \sum_{i=1}^{k} u_D(i) = \frac{T_D}{\alpha} e(k)$$

Aktualna wartość wyjścia $u_D(k)$ występuje w dwóch składnikach, dlatego musimy wykonać kilka przekształceń

$$u_D(k) + \frac{T_s}{\alpha} \sum_{i=1}^{k-1} u_D(i) + \frac{T_s}{\alpha} u_D(k) = \frac{T_D}{\alpha} e(k) \quad \rightarrow \quad u_D(k) \left[1 + \frac{T_s}{\alpha} \right] = \frac{T_D}{\alpha} e(k) - \frac{T_s}{\alpha} \sum_{i=1}^{k-1} u_D(i)$$



Rys. 20.11: Realizacja składowej różniczkowej algorytmu PID

Musimy jeszcze dodać wzmocnienie proporcjonalne. W tym celu wystarczy sygnał wejściowy pomnożyć przez K. Aproksymacja cyfrowa rzeczywistego układu różniczkującego opisana jest więc wzorem

$$u_D(k) = \frac{K \cdot T_D}{\alpha + T_s} e(k) - \frac{T_s}{\alpha + T_s} \sum_{i=1}^{k-1} u_D(i) \tag{20.7}$$

Realizacja części różniczkującej algorytmu PID, zgodnie ze wzorem (20.7) pokazana jest na rysunku 20.11 W naszym projekcie uchyb $e(k)$ zostanie zamieniony na zmierzone wyjście obiektu $y(k)$. Zgodnie z rysunkiem 20.9, wyjście $u(k)$ regulatora PID jest równe

$$u(k) = K \cdot e(k) + u_I(k) - u_D(k)$$

Łącząc wszystkie składowe możemy narysować strukturę równoległą regulatora PID – rysunek 20.12

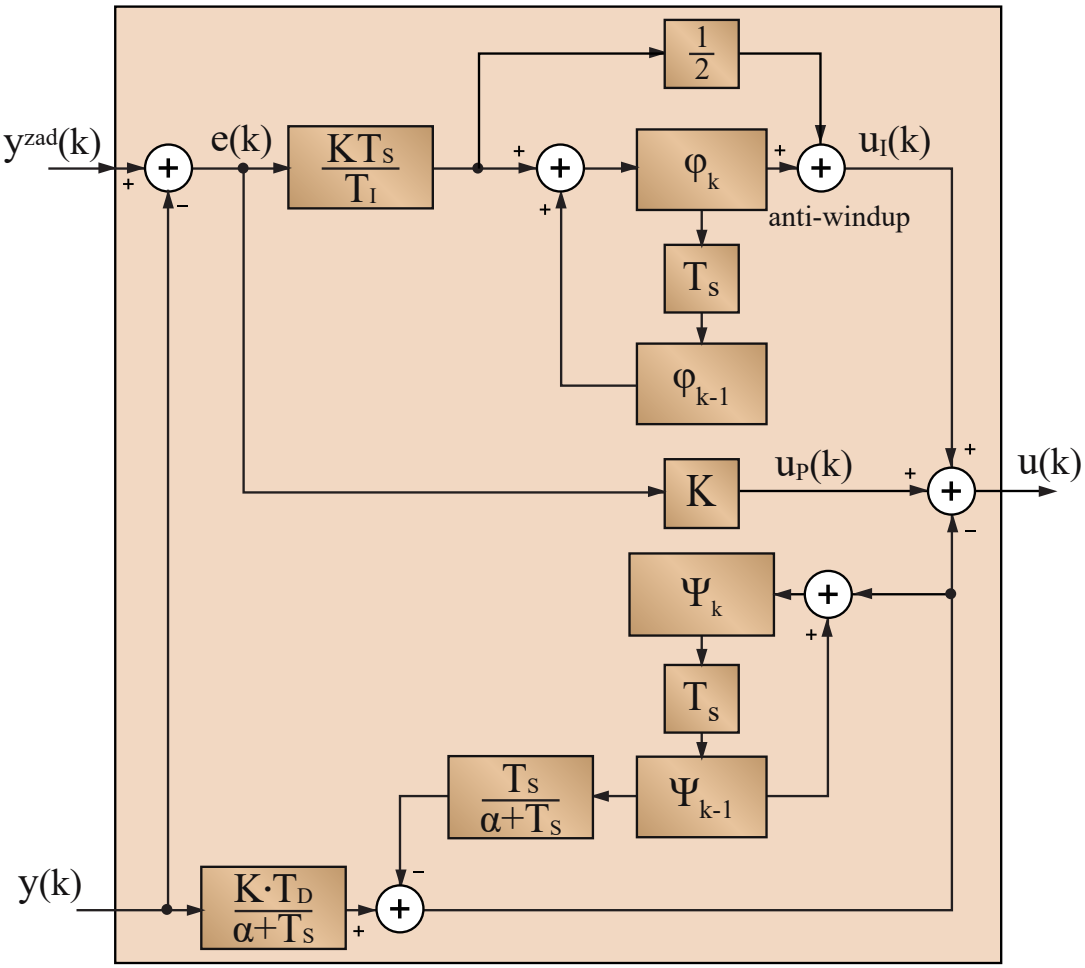
20.3.3. Projekt30_PID

Zakres wyjścia przetwornika DAC z włączonym wzmacniaczem wyjściowym, to binarnie $[250, 3846]$. W przypadku przesunięcia zakresu o 2048, zakres jest równy $[-1798, +1798]$. Takie ograniczenie nałożymy na wszystkie sygnały oraz zmienną φ_k, która jest bezpośrednio dołączona do sygnału $u_I(k)$. Zapewni to nam zerowy poziom nasycenia części całkującej regulatora.

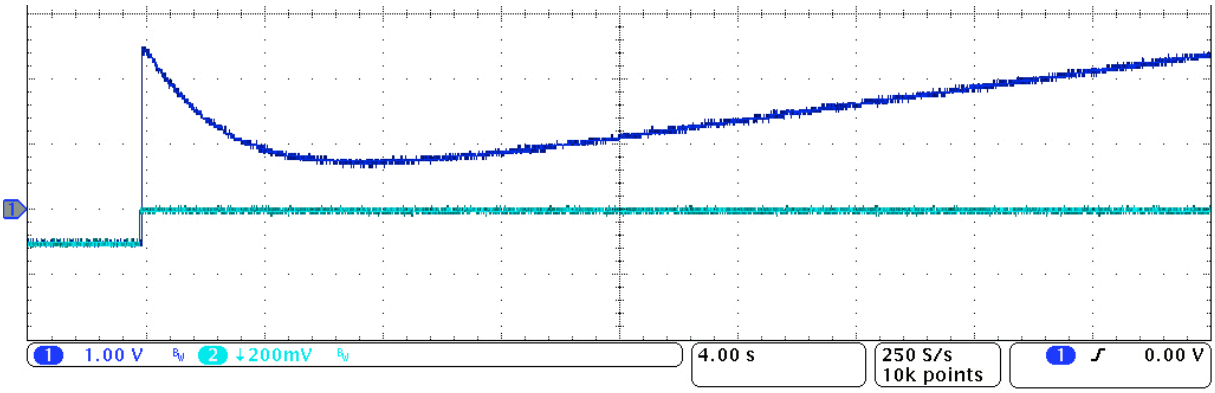
Na rysunkach 20.15 i 20.16, kanał 1 to odpowiedź obiektu, kanał 2 to sterowanie. Rysunek 20.15 przedstawia efekt regulacji PID obiektu oscylacyjnego. Nastawy:

$$K = 0.75, \quad T_I = 5.0, \quad T_D = 0.5, \quad \alpha = 3.0$$

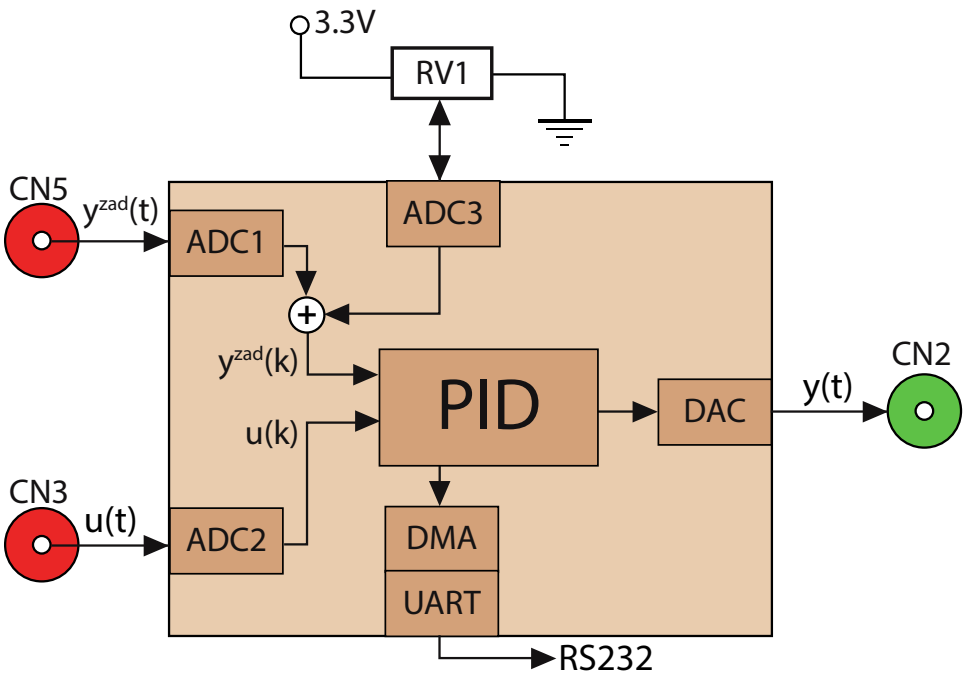
Rysunek 20.16 przedstawia efekt regulacji tego samego obiektu oscylacyjnego, przez regulator zaprojektowany metodą lokowania biegunów (*pole placement*). Metoda lokowania biegunów będzie opisana w rozdziale 21.



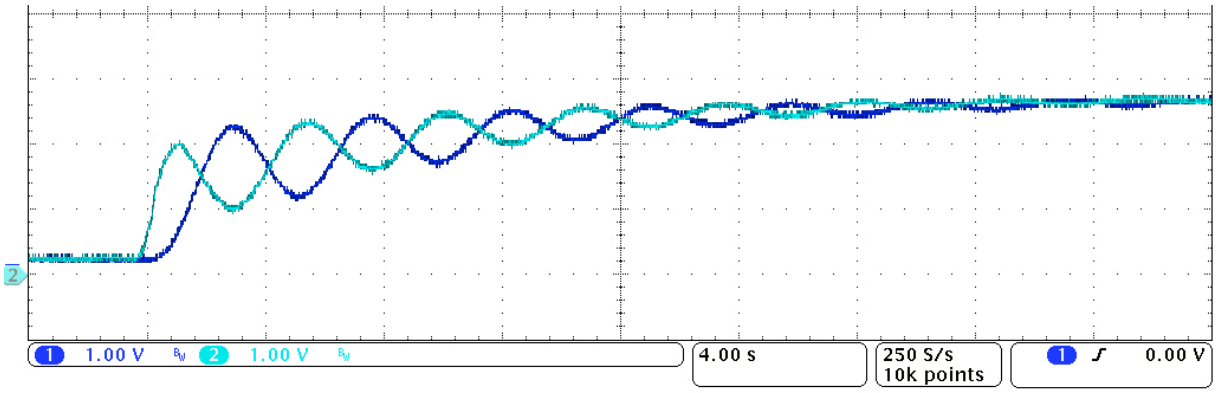
Rys. 20.12: Schemat algorytmu pozycyjnego PID, równoległego



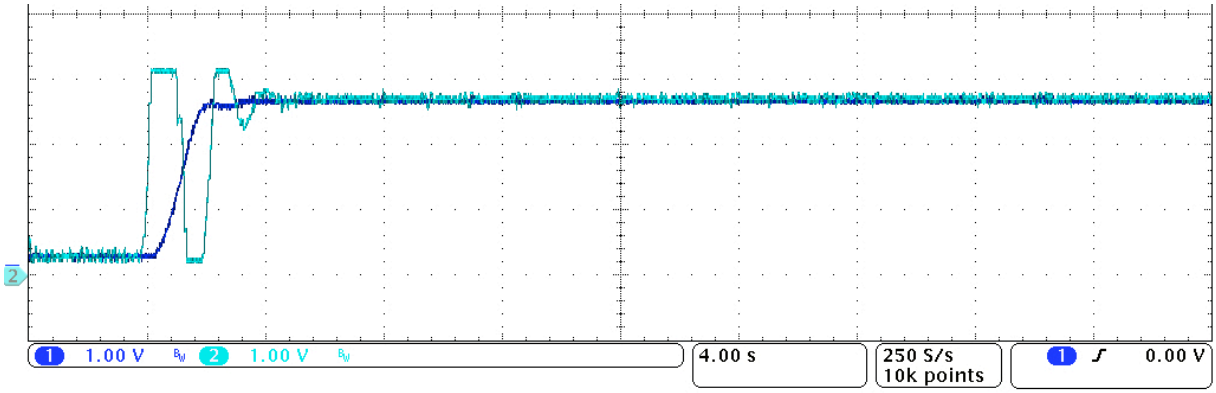
Rys. 20.13: Odpowiedź regulatora PID na skokową zmianę $y(t)$



Rys. 20.14: Realizacja regulatora PI / PID na płycie STM3210E-EVAL



Rys. 20.15: Wyjście obiektu po skokowej zmianie wartości zadanej



Rys. 20.16: Regulacja *pole placement*, bieguny $p_1 = 0.7,\ p_2 = 0.7,\ p_3 = 0.9$

20.4. Dobór nastaw regulatora PID

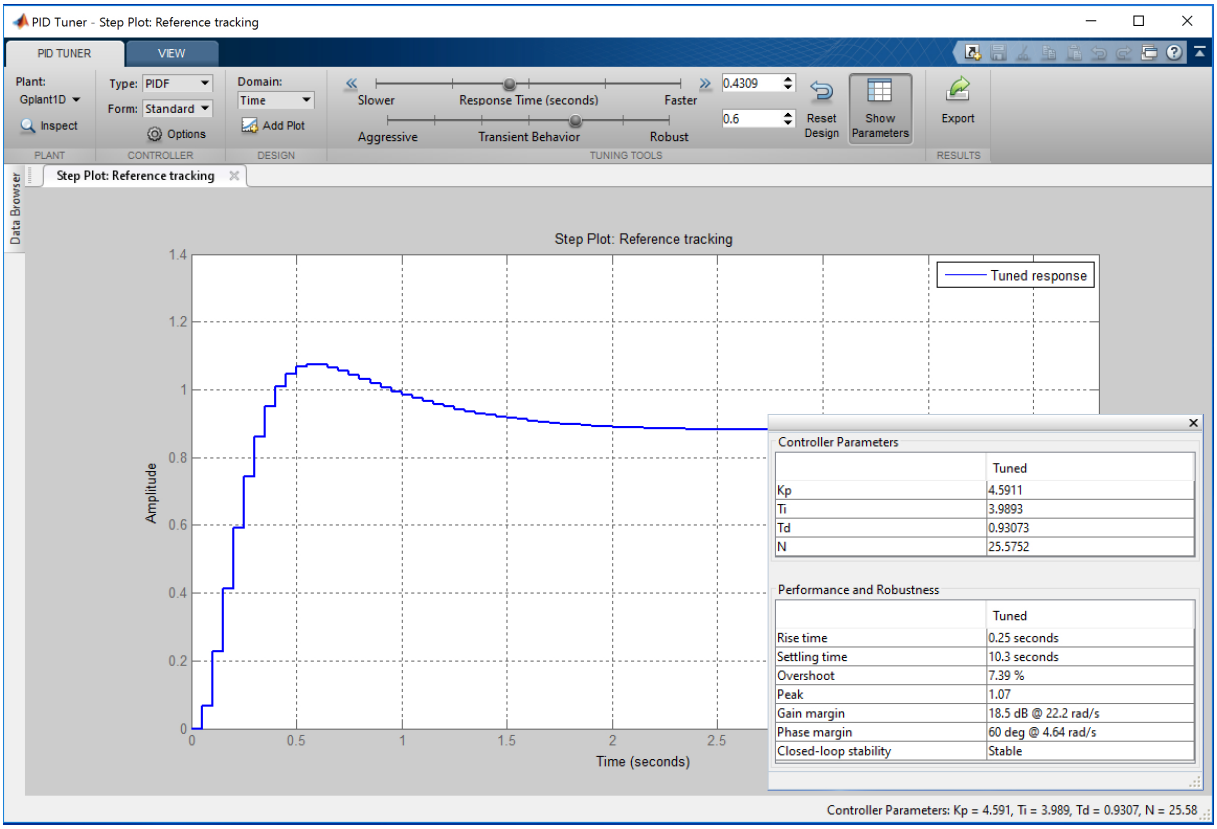
Istnieje wiele metod doboru nastaw regulatora PID. Najbardziej znana to metoda Zieglera-Nicholsa, a najczęściej stosowana to metoda prób i błędów. Znamy transmitancję obiektu, dlatego możemy wykorzystać przybornik (*tool-box*) MatLaba:

```
wn=1; zeta=0.2;      % obiekt oscylacyjny
Gplant1=wn*wn/(s^2+2*zeta*wn*s+wn^2);
Gplant1D=c2d(Gplant1,Ts,'matched');  % matched pole-zero method
pidtool(Gplant1D);  % dobór nastaw PID
```

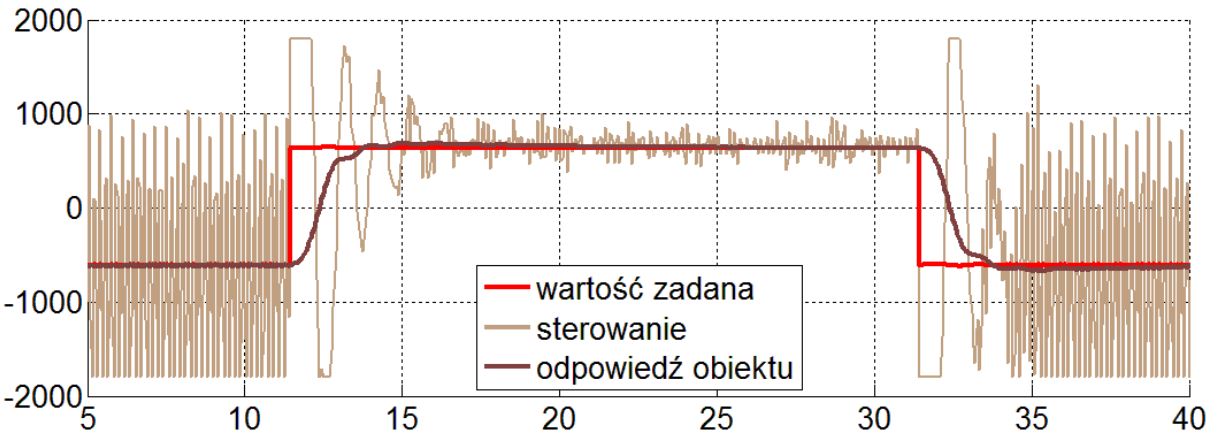
W polu CONTROLLER wybieramy konfigurację `Type:PIDF, Form:Standard`, wtedy transmitancja regulatora ma postać:

$$C = K_p\left[1 + \frac{1}{T_i}IF(z) + \frac{T_d}{T_d/N + DF(z)}\right] \tag{20.8}$$

Porównanie wzoru (20.8) z (20.2) oraz (20.4) prowadzi do wniosku, że $\alpha = T_d/N$. Zmieniając położenie suwaków w polu TUNING TOOLS, możemy wybrać odpowiednie parametry zamkniętego układu regulacji. W przykładzie pokazanym na rysunku 20.17, dobrano następujące nastawy regulatora PID:



Rys. 20.17: Dobór nastaw w środowisku MatLab



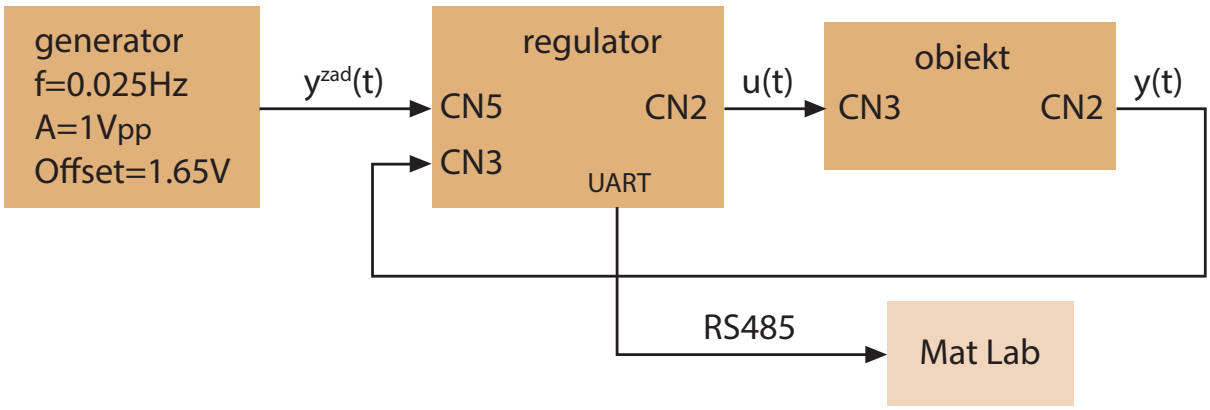
Rys. 20.18: Regulacja PID obiektu oscylacyjnego

K_p = 4.5911
T_i = 3.9893
T_d = 0.93073
$\alpha = T_d/N$ = 0.93073 / 25.5752 = 0.0363919

Efekt regulacji pokazano na rysunku 20.18. Obiekt jest regulowany bardziej agresywnie, niż na rysunku 20.15. Uzyskaliśmy krótki czas ustalania (*settling time*), ale kosztem silnego wzmocnienia szumów pomiarowych.

Programy projektu *projekt30_PID* zostały uzupełnione o nadawanie wartości sterowania $u(k)$, zmierzonej odpowiedzi $y(k)$ oraz wartości zadanej $r(k)$. Wykorzystano funkcje z projektu *Projekt24_1_Modbus_Master* do obsługi układu UART1 + DMA. Komunikat jest wysyłany po każdym wyliczeniu nowego sterowania w przerwaniu systemowym:

```
// nadamy komunikat RS232
n=snprintf((char *)Tx_message,40,"u=%8.2f;y=%8.2f;r=%8.2f;\13", u, y, r);
DMA1 -> IFCR = 0x00007000;        // kasujemy zgłoszenia przerwań
DMA_Cmd(DMA1_Channel4, DISABLE); // wyłącz kanał 4/DMA1
DMA1_Channel4 -> CNDTR = n;      // ilość bajtów do przesłania
DMA_Cmd(DMA1_Channel4, ENABLE);  // włącz kanal 4/DMA1 = transmisja
Tx_num++;
```



Rys. 20.19: Układ regulacji z rejestracją sygnałów

Parametry transmisji: 115'200, 8, E, 1. Skrypt MatLaba do odbioru i archiwizacji danych pokazany jest na rysunku 20.20.

rejestracja sygnałów regulatora

```
% wektory danych Rx
N = 1600;
U = zeros(N,1); % wektor wejść (sterowań) obiektu
Y = zeros(N,1); % wektor wyjść obiektu
R = zeros(N,1); % wektor wartości zadanej
Ts = 0.05;      % okres próbkowania
t = (0:Ts:(N-1)*Ts)';   % podstawa czasu
%
% konfiguracja portu COM8 i odbiór danych
delete(instrfindall); % zamknięcie wszystkich połączeń szeregowych
s = serial('COM8');   % COM8 to jest port utworzony przez mikrokontroler
set(s,'BaudRate',115200);
set(s,'StopBits',1);
set(s,'Parity','even');
set(s,'DataBits',8);
set(s,'Timeout',0.05);
set(s,'InputBufferSize',100);
set(s,'Terminator',13);
fopen(s);   % otwarcie kanału komunikacyjnego
Tp = 0.05;  % okres próbkowania
%
% pętla odbiorcza
for i=1:N
   txt = fread(s,34); % odczytanie z portu szeregowego
    % txt powinien zawierać y=%8.2f;u=%8.2f;r=%8.2f\13;
    eval(char(txt')); % rozpakujemy ramkę
    U(i) = u; % zapis do wektorów danych odebranych
    Y(i) = y;
    R(i) = r;
end
```

Rys. 20.20: Skrypt MatLaba do odbioru danych z regulatora

Rozdział 21

Lokowanie biegunów

Lokowanie biegunów (*pole placement*) to prosta metoda projektowania regulatorów, która daje bardzo dobre efekty. Polega na takim doborze transmitancji regulatora, aby układ regulacji działający w zamkniętej pętli miał z góry założone bieguny. Istnieje w dwóch odmianach: w przestrzeni stanów (sprzężenie od stanu, *state-space approach*) oraz z wykorzystaniem transmitancji wielomianowych (*polynomial approach*). My zastosujemy podejście wielomianowe jako prostsze.

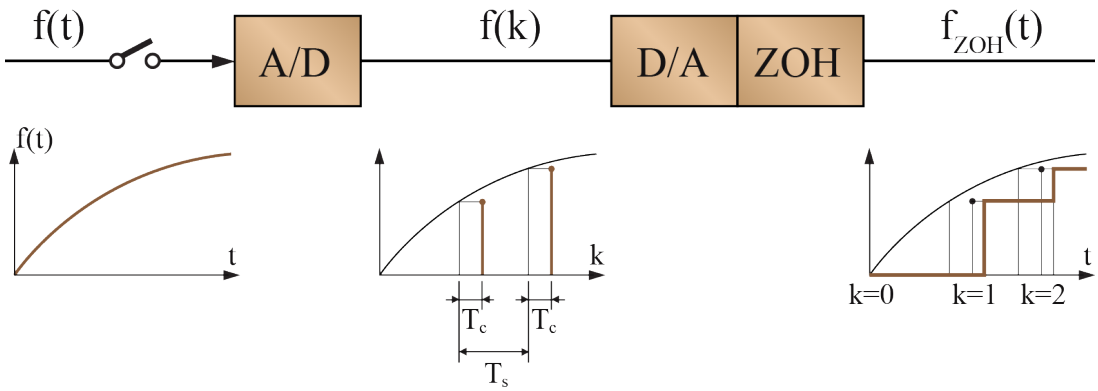
21.1. Transmitancja dyskretna i transformata $\mathcal{Z}$ (Laurenta)

W analizie układów LTI ciągłych, wykorzystywane jest przekształcenie operatorowe Laplace'a, które zdefiniowane jest wzorem

$$\mathcal{L}\Big\{f(t)\Big\} = F(s) = \int_0^\infty f(\tau)e^{-s\tau}\mathrm{d}\tau \tag{21.1}$$

gdzie $s = \sigma + j\omega$ jest zmienną zespoloną, w której σ to współczynnik tłumienia (*damping factor*), a ω to pulsacja.

W układach cyfrowych, sygnał ciągły $f(t)$ zamieniany jest na ciąg liczb w procesie próbkowania (rysunek 21.1). W określonych chwilach czasowych (numerowanych $k = 0, 1, 2, \ldots$) inicjowane jest przetwarzanie A/D, które trwa przez czas T_c (*conversion time*). Po czasie T_c, na wyjściu przetwornika A/D uzyskujemy liczbę, która najlepiej odpowiada chwilowej wartości sygnału $f(t)$. Czas konwersji jest na tyle krótki (np. 1 μs), że pominiemy go w dalszych rozważaniach. Chwile, w których następuje próbkowanie ($0,\ T_s,\ 2T_s,\ 3T_s,\ \ldots\ kT_s$) są odległe od siebie o stały przedział czasu, zwany okresem próbkowania (*sampling time*) T_s. Odwrotnością T_s jest częstotliwość próbkowania $f_s = 1/T_s$.



Rys. 21.1: Próbkowanie sygnałów ciągłych

W analizie matematycznej, do zapisu sygnału dyskretnego $f(k)$, wygodnie jest posłużyć się funkcją **delta Diraca**.

$$\delta(t) = \begin{cases} \infty & dla \quad t = 0 \\ 0 & dla \quad t \neq 0 \end{cases} \qquad \int_{\infty}^{\infty} \delta(\tau)\mathrm{d}\tau = 1$$

Jest to nieskończenie wysoki impuls w zerze, o jednostkowym polu powierzchni. Sygnał ciągły $f(t)$ pomnożymy przez ciąg równoodległych impulsów, z których każdy jest deltą Diraca. Uzyskamy ciąg impulsów o nieskończenie krótkim czasie trwania i polu powierzchni równym chwilowej wartości sygnału próbkowanego $f(kT_s) \cdot \delta(t - kT_s)$. Sygnał ciągły **zastępujemy sumą impulsów**

$$f(t) \qquad \longrightarrow \qquad \sum_{k=0}^{\infty} f(kT_s) \cdot \delta(t - kT_s) \tag{21.2}$$

obliczmy transformatę Laplace'a z sumy impulsów (21.2).

$$F_\delta(s) = \int_0^{\infty} \sum_{k=0}^{\infty} f(kT_s)\delta(\tau - kT_s)e^{-s\tau}\mathrm{d}\tau = \sum_{k=0}^{\infty} f(kT_s) \int_0^{\infty} \delta(\tau - kT_s)e^{-s\tau}\mathrm{d}\tau = \sum_{k=0}^{\infty} f(kT_s)e^{-skT_s}$$

W powyższym przekształceniu wykorzystaliśmy liniowość transformaty Laplace'a, twierdzenie o przesunięciu w czasie $x(t - t_0) \rightarrow X(s)e^{-st_0}$ oraz $\mathcal{L}\{\delta(t)\} = 1$. Zastosujemy podstawienie

$$z = e^{sT_s} \qquad \rightarrow \qquad z = e^{(\sigma + j\omega)T_s} = e^{\sigma T_s}e^{j\omega T_s} = e^{\sigma T_s}\cos(\omega T_s) + je^{\sigma T_s}\sin(\omega T_s)$$

gdzie z jest zmienną zespoloną. Uzyskaliśmy definicję transformaty $\mathcal{Z}$

$$\mathcal{Z}\Big\{f(k)\Big\} = F(z) = \sum_{k=0}^{\infty} f(k)z^{-k} \tag{21.3}$$

Wyrażenie $e^{-sT_s} = z^{-1}$ jest transformatą Laplace'a opóźnienia o czas T_s. Odwrotność zmiennej zespolonej z^{-1} jest więc odpowiednikiem opóźnienia o okres próbkowania. Łatwo to zrozumieć, jeżeli zapiszemy transformatę $\mathcal{Z}$ funkcji opóźnionej o jeden takt. Zgodnie z zasadą przyczynowości $f(k < 0) = 0$

$$\mathcal{Z}\Big\{f(k-1)\Big\} = \sum_{k=0}^{\infty} f(k-1)z^{-k} = \sum_{k=1}^{\infty} f(k-1)z^{-(k-1)}z^{-1} = \sum_{k=0}^{\infty} f(k)z^{-k}z^{-1} = F(z)z^{-1}$$

Możemy zastosować prosty schemat przechodzenia od opisu układu LTI za pomocą równania różnicowego, do opisu w dziedzinie transformat $\mathcal{Z}$, a stąd do transmitancji dyskretnej. Jeżeli mamy równanie różnicowe

$$y(k) = -a_1y(k-1) - a_2y(k-2) + b_0u(k) + b_1u(k-1) \tag{21.4}$$

to korzystając z interpretacji z^{-1} możemy łatwo przejść do dziedziny zmiennej zespolonej z

$$Y(z) = -a_1Y(z) \cdot z^{-1} - a_2Y(z) \cdot z^{-2} + b_0U(z) + b_1U(z) \cdot z^{-1}$$

$$Y(z)\Big[1 + a_1z^{-1} + a_2z^{-2}\Big] = U(z)\Big[b_0 + b_1z^{-1}\Big]$$

$$H(z) = \frac{Y(z)}{U(z)} = \frac{b_0 + b_1 z^{-1}}{1 + a_1 z^{-1} + a_2 z^{-2}} \tag{21.5}$$

Wprowadzimy teraz operator opóźnienia q^{-1}, który będziemy stosować w zapisie równań różnicowych i transmitancjach dyskretnych. Transmitancję (21.5) zapiszemy jako

$$H(q) = \frac{Y(q)}{U(q)} = \frac{b_0 + b_1 q^{-1}}{1 + a_1 q^{-1} + a_2 q^{-2}} \tag{21.6}$$

Pamiętajmy, że z to **zmienna zespolona**, a q^{-1} to **operator**. Wzory (21.5) i (21.6) wyglądają podobnie, ale nie możemy ich uważać za jednakowe. Ilustracją jest przykład równania rożnicowego

$$y(k+1) + ay(k) = u(k+1) + au(k) \tag{21.7}$$

Transmitancja liniowego układu dynamicznego LTI, opisanego równaniem różnicowym (21.7), to

$$H(z) = \frac{z+a}{z+a} = 1$$

Mogliśmy dokonać skrócenia, ponieważ z to zmienna zespolona. Ale nie powinniśmy uwierzyć, że układ LTI (21.7) jest identyczny z układem LTI (21.8)

$$y(k) = u(k) \tag{21.8}$$

W celu wykazania różnicy, rozwiążmy równanie 21.7. Założymy niezerowy stan początkowy o wartości $y(0)$ oraz $u(0) = 0$

$$\begin{aligned}
y(1) &= -ay(0) + u(1) + au(0) = -ay(0) + u(1) \\
y(2) &= -ay(1) + u(2) + au(1) = (-a)^2 y(0) - au(1) + u(2) + au(1) = (-a)^2 y(0) + u(2) \\
y(3) &= -ay(2) + u(3) + au(2) = (-a)^3 y(0) - au(2) + u(3) + au(2) = (-a)^3 y(0) + u(3) \\
&\dots \qquad \dots
\end{aligned}$$

rozwiązaniem równania różnicowego (21.7) jest więc:

$$y(k) = (-a)^k y(0) + u(k), \qquad k > 0$$

które jest identyczne z rozwiązaniem równania (21.8) **tylko wtedy, gdy warunek początkowy** $y(0)$ **jest zerowy**. Dla dużych k, warunek początkowy $y(0)$ możnaby pominąć, jeżeli $|a| < 1$. Uprawnione jest więc stwierdzenie, że transmitancja

$$H(z) = \frac{z+a}{z+a}$$

może być przybliżona jedynką dla $|a| < 1$, ale nie dla $|a| \geqslant 1$. Korzystając z operatora opóźnienia, nie mamy tych kłopotów. Wyrażenia

$$H(q) = \frac{q+a}{q+a}$$

nie można skrócić, gdyż q nie jest liczbą, tylko operatorem.

Z powyższego przykładu wynika, że liczba zespolona z i operator q nie są tym samym. W transmitancji dyskretnej, takiej jak (21.5) może zdarzyć się skrócenie zera i bieguna, gdy oba wielomiany $A(z)$ i $B(z)$ mają jednakowe czynniki (*factors*). Jednak w praktyce, takie skrócenie jest ryzykowne, a już zupełnie niedopuszczalne, gdy próbujemy skrócić biegun niestabilny.

21.1.1. Operator różnicy wstecznej Δ

$$\Delta = 1 - z^{-1} \qquad \rightarrow \qquad \Delta u(k) = u(k) - u(k-1)$$

Odwrotność $1/\Delta$ to operator całkowania

$$\frac{1}{\Delta} = \frac{1}{1 - z^{-1}} \qquad \rightarrow \qquad \Delta y(k) = \frac{u(k)}{\Delta} \qquad \rightarrow \qquad y(k) = u(k) + y(k-1) \tag{21.9}$$

Obliczymy odpowiedź impulsową $u(k) = \delta(k)$

$$\delta(k) = \begin{cases} 1 & gdy \quad k = 0 \\ 0 & gdy \quad k \neq 0 \end{cases}$$

obiektu (21.9) o transmitancji $1/\Delta$

$$\begin{aligned} y(0) &= u(0) + y(-1) = 1 + 0 = 1 \\ y(1) &= u(1) + y(0) \quad = 0 + 1 = 1 \\ y(2) &= u(2) + y(1) \quad = 0 + 1 = 1 \\ y(3) &= u(3) + y(2) \quad = 0 + 1 = 1 \\ &\dots \qquad \dots \end{aligned}$$

Otrzymaliśmy skok jednostkowy $\mathbf{1}(k) = 1, \ k \geqslant 0$. W dziedzinie czasu ciągłego

$$\int_{-\infty}^{\infty} \delta(t)dt = \mathbf{1}(t)$$

21.1.2. Bieguny transmitancji (*poles*)

Rozważmy obiekt o transmitancji

$$\frac{Y(z)}{U(z)} = \frac{z}{z-p} = \frac{1}{1 - pz^{-1}} \qquad \longrightarrow \qquad y(k) = p \cdot y(k-1) + u(k)$$

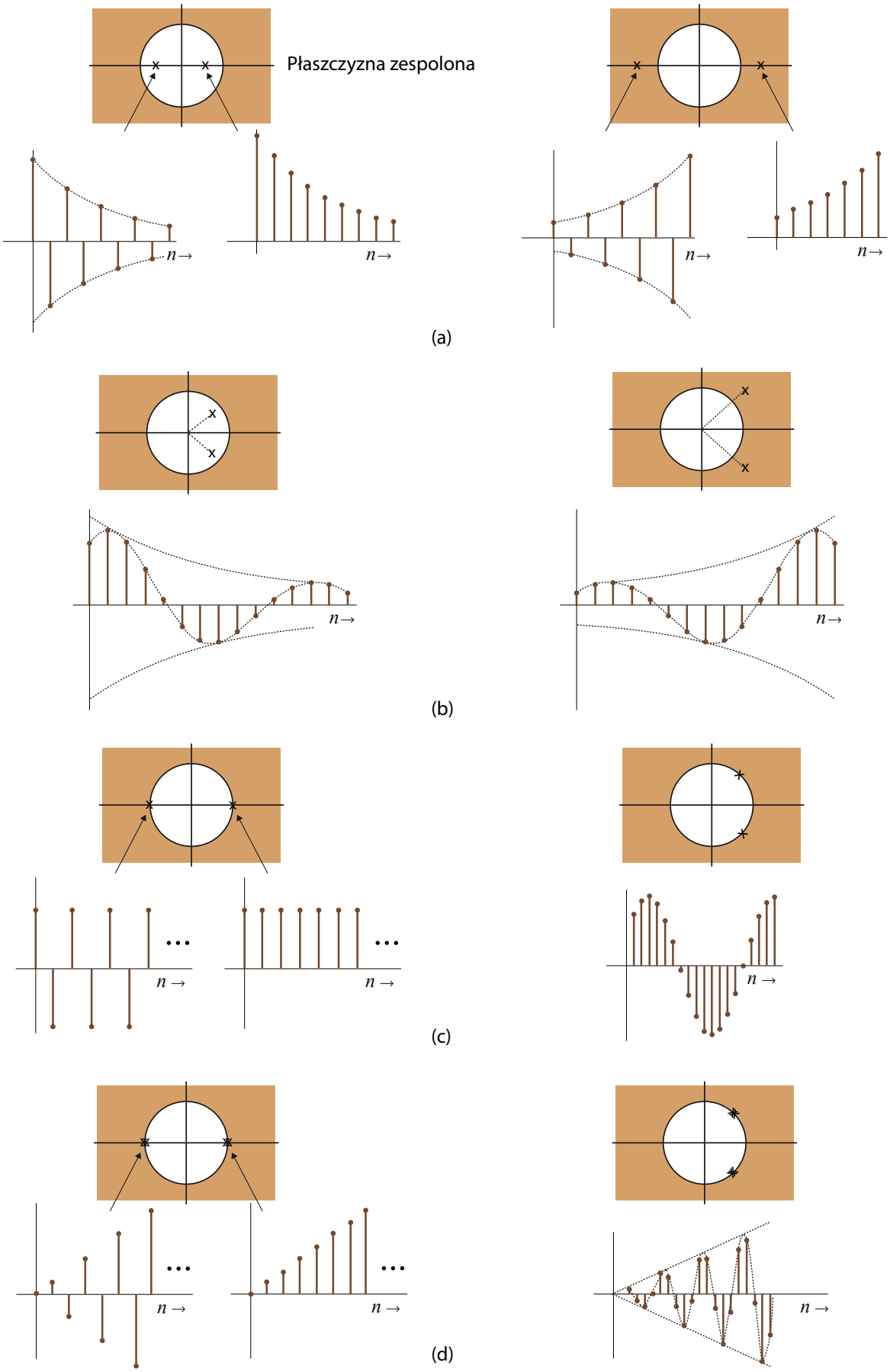
Każdy pierwiastek wielomianu licznika to **zero transmitancji**, a wielomianu mianownika, to **biegun transmitancji**. Nasz obiekt ma jedno zero w zerze i jeden biegun równy p. Wyznaczymy odpowiedź impulsową $u(k) = \delta(k)$

$$\begin{aligned} y(k) &= p \cdot y(k-1) + u(k) \\ y(0) &= p \cdot y(-1) + u(0) = 0 + 1 = 1 \\ y(1) &= p \cdot y(0) + u(1) = p \cdot 1 + 0 = p \\ y(2) &= p \cdot y(1) + u(2) = p \cdot p = p^2 \\ y(3) &= p \cdot y(2) + u(3) = p \cdot p^2 = p^3 \\ &\dots \qquad \dots \end{aligned}$$

Uzyskaliśmy wzór na odpowiedź inpulsową: $y(k) = p^k$

Odpowiedź impulsowa zależy od wartości bieguna p, jak pokazano na rysunku 21.2. Dla wartości rzeczywistych p:

- $0 < p < 1$ (rysunek 21.2a) odpowiedź impulsowa jest sygnałem monotonicznie malejącym – biegun stabilny



Rys. 21.2: Położenia biegunów i odpowiadające im odpowiedzi układu dyskretnego

- $-1 < p < 0$ (rysunek 21.2a) odpowiedź impulsowa maleje, ale przyjmuje naprzemiennnie wartości dodatnie i ujemne $(-p)^k$ – biegun stabilny dzwoniący
- $|p| > 1$ odpowiedź impulsowa narasta monotonicznie $p > 1$ lub naprzemiennie $p < -1$
- $|p| = 1$ odpowiedź o stałej amplitudzie (rysunek 21.2c), gdy $p = 1$ to mamy obiekt całkujący o transmitancji (21.9)

Podwójny biegun w jedynce (rysunek 21.2d) oznacza obiekt z podwójnym całkowaniem. Zrealizujemy go w przykładzie 21.1. Przykładowa transmitancja:

$$\frac{Y(z)}{U(z)} = \frac{1}{(1-z^{-1})(1-z^{-1})} = \frac{1}{1-2z^{-1}+z^{-2}}$$

Równanie różnicowe i odpowiedź impulsowa

$$\begin{aligned}
y(k) &= 2y(k-1) - y(k-2) + u(k) \\
y(0) &= 2y(-1) - y(-2) + u(0) = 1 \\
y(1) &= 2y(0) - y(-1) + u(1) = 2 \\
y(2) &= 2y(1) - y(0) + u(2) = 4 - 1 = 3 \\
y(3) &= 2y(2) - y(1) + u(3) = 6 - 2 = 4 \\
y(4) &= 2y(3) - y(2) + u(4) = 8 - 3 = 5 \\
&\dots \qquad \dots
\end{aligned}$$

Obiekt z podwójnym całkowaniem jest niestabilny: jego odpowiedź impulsowa rozbiega się do nieskończoności, jak pokazano na rysunku 21.2d.

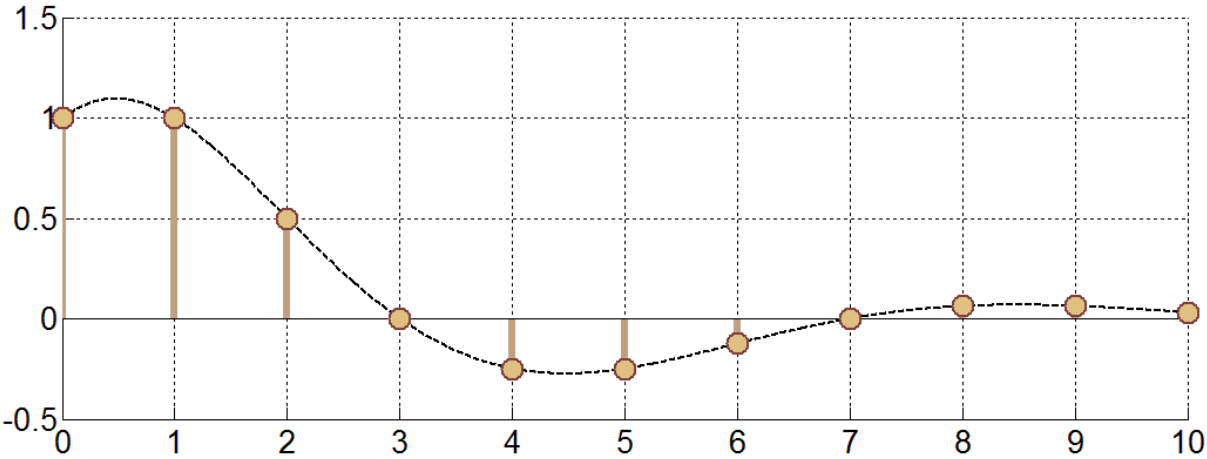
21.1.3. Bieguny zespolone sprzężone

Wyznaczymy bieguny (pierwiastki mianownika) transmitancji

$$\frac{Y(z)}{U(z)} = \frac{z^2}{z^2 - z + 0.5} \quad \rightarrow \quad \Delta = 1 - 2 = -1 \quad \rightarrow \quad z_{1/2} = 0.5 \pm 0.5j \tag{21.10}$$

mamy więc dwa pierwiastki zespolone sprzężone. Obliczymy odpowiedź impulsową

$$\begin{aligned}
y(k) &= y(k-1) - 0.5y(k-2) + u(k) \\
y(0) &= y(-1) - 0.5y(-2) + u(0) = 1 \\
y(1) &= y(0) - 0.5y(-1) + u(1) = 1 \\
y(2) &= y(1) - 0.5y(0) + u(2) = 1 - 1/2 = 1/2 \\
y(3) &= y(2) - 0.5y(1) + u(3) = 1/2 - 1/2 = 0 \\
y(4) &= y(3) - 0.5y(2) + u(4) = 0 - 1/4 = -1/4 \\
y(5) &= y(4) - 0.5y(3) + u(5) = -1/4 - 0 = -1/4 \\
y(6) &= y(5) - 0.5y(4) + u(6) = -1/4 + 1/8 = -1/8 \\
y(7) &= y(6) - 0.5y(5) + u(7) = -1/8 + 1/8 = 0 \\
y(8) &= y(7) - 0.5y(6) + u(8) = 0 + 1/16 = 1/16 \\
y(9) &= y(8) - 0.5y(7) + u(9) = 1/16 - 0 = 1/16 \\
&\dots \qquad \dots
\end{aligned}$$



Rys. 21.3: Odpowiedź impulsowa obiektu o transmitancji (21.10)

Para pierwiastków zespolonych oznacza dwa sygnały zespolone

$$y_1(k) = |z|^k e^{j\phi} = |z|^k \Big[cos(\phi \cdot k) + jsin(\phi \cdot k)\Big]$$
$$y_2(k) = |z|^k e^{-j\phi} = |z|^k \Big[cos(\phi \cdot k) - jsin(\phi \cdot k)\Big]$$

Po ich zsumowaniu uzyskamy sygnał rzeczywisty

$$y_2(k) = |z|^k e^{-j\phi} = 2|z|^k cos(\phi \cdot k)$$

gdzie moduł pierwiastka $|z|$ to współczynnik tłumienia, a ϕ to pulsacja. Jeżeli oba pierwiastki leżą wewnątrz okregu jednostkowego, to $|z| < 1$ i odpowiedź jest zanikająca, gdy na zewnątrz, to $|z| > 1$ i odpowiedź rozbiega się do nieskończoności, rysunek 21.2b. Oba bieguny na okręgu jednostkowym to sinus o stałej amplitudzie, rysunek 21.2c.

Rysunek 21.3 przedstawia odpowiedź impulsową obiektu o transmitancji (21.10) wraz z obwiednią sygnału.

21.1.4. Twierdzenie o wartości końcowej

Bieguny transmitancji (*poles*)

$$\frac{Y(z)}{U(z)} = \frac{bz}{z-p} = \frac{b}{1-pz^{-1}} \qquad \longrightarrow \qquad y(k) = p \cdot y(k-1) + b \cdot u(k)$$

Odpowiedź skokowa $u(k) = 1(k) = 1 \; gdy \; k \geqslant 0$

$$\begin{aligned}
y(k) &= p \cdot y(k-1) + b \cdot u(k) \\
y(0) &= p \cdot y(-1) + b \cdot u(0) = b \\
y(1) &= p \cdot y(0) + b \cdot u(1) = pb \\
y(2) &= p \cdot y(1) + b \cdot u(2) = p^2 b + pb + b \\
y(3) &= p \cdot y(2) + b \cdot u(3) = p^3 b + p^2 b + pb + b \\
y(4) &= p \cdot y(3) + b \cdot u(4) = p^4 b + p^3 b + p^2 b + pb + b \\
y(5) &= p \cdot y(4) + b \cdot u(5) = p^5 b + p^4 b + p^3 b + p^2 b + pb + b \\
&\ldots \\
y(k) &= b\Big(1 + p + p^2 + \ldots + p^k\Big) = b\frac{1-p^{k+1}}{1-p}
\end{aligned}$$

gdyż suma szeregu geometrycznego:

$$S_n = a{+}ap{+}\ldots{+}ap^{n-1} = \sum_{k=1}^{n} = a\frac{1-p^n}{1-p} \qquad \rightarrow \qquad S_{n+1} = a{+}\ldots{+}ap^n = \sum_{k=1}^{n+1} = a\frac{1-p^{n+1}}{1-p}$$

Wartość odpowiedzi na skok jednostkowy, dla $|p| \leqslant 1$, po nieskończonym czasie jest więc równa

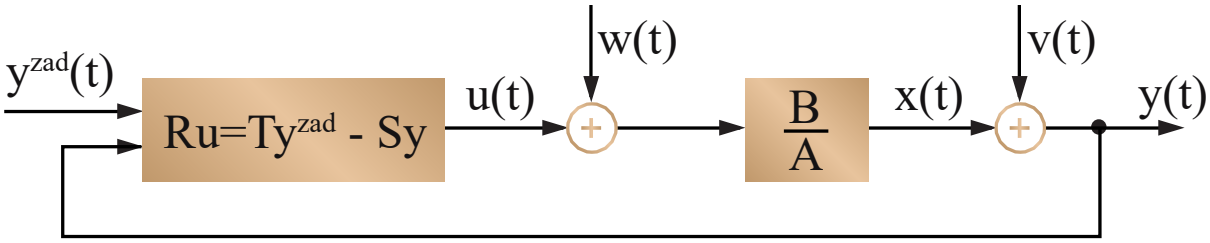
$$y(\infty) = \lim_{k\to\infty} b\frac{1-p^{k+1}}{1-p} = \frac{b}{1-p}$$

Jest to zgodne z twierdzeniem o wartości końcowej

$$y(\infty) = \lim_{z\to 1}\frac{Y(z)}{U(z)} = \lim_{z\to 1}\frac{bz}{z-p} = \frac{b}{1-p}$$

21.2. Prosty projekt regulatora

Zaczniemy od zaprojektowania regulatora do obiektu z podwójnym całkowaniem. Układ regulacji pokazany jest na rysunku 21.4. Obiekt opisany jest za pomocą transmitancji $B(q)/A(q)$. Zakładamy, że rząd wielomianu $B(q)$ jest mniejszy lub równy rzędowi $A(q)$ i że te wielomiany nie mają wspólnych czynników. Ponadto zakładamy, że wielomian $A(q)$ jest znormalizowany, czyli współczynnik przy najwyższej potędze q jest jedynką. Taki wielomian jest nazywany *monic*.



Rys. 21.4: Układ regulacji z lokowaniem biegunów

Wyjście obiektu to $x(t)$. Pomiar wyjścia obciążony jest szumem $v(t)$, do regulatora trafia więc sygnał $y(t) = x(t) + v(t)$. Na obiekt oddziaływuje zakłócenie sprowadzone do wejścia $\omega(t)$. Symbole R, T, S, B oraz A oznaczają wielomiany. Korzystając z operatora przesunięcia w przód q (*forward-shift operator*), układ regulacji możemy opisać równaniami różnicowymi

$$u = \frac{T(q)}{R(q)}y^{zad} - \frac{S(q)}{R(q)}y, \qquad y = v + x = v + \frac{B(q)}{A(q)}(u+\omega)$$

Z powyższych równań wyeliminujemy wymuszenie obiektu u

$$y = v + \frac{B}{A}\left[\frac{T}{R}y^{zad} - \frac{S}{R}y + \omega\right] = v + \frac{BT}{AR}y^{zad} - \frac{BS}{AR}y + \frac{B}{A}\omega$$

Obie strony mnożymy przez iloczyn wielomianów $A(q)\cdot R(q)$

$$ARy = ARv + BTy^{zad} - BSy + BR\omega \quad \rightarrow \quad (AR+BS)y = ARv + BTy^{zad} + BR\omega$$

$$y = \frac{BT}{AR+BS}y^{zad} + \frac{AR}{AR+BS}v + \frac{BR}{AR+BS}\omega \tag{21.11}$$

Równanie (21.11) opisuje w jaki sposób zamknięty układ regulacji odpowiada na wartość zadaną y^{zad} i zakłócenia v, ω. W naszym prostym projekcie zajmiemy się tylko transmitancją pomiędzy wyjściem obiektu, a wartością zadaną

$$y = \frac{B(q)T(q)}{A(q)R(q)+B(q)S(q)}y^{zad} \tag{21.12}$$

Wielomiany $A(q)$ i $B(q)$ są dane, możemy zmieniać $R(q)$, $S(q)$ oraz $T(q)$. Zgodnie z transmitancją (21.12), zamknięta pętla regulacji przenosi zera obiektu (wielomian $B(q)$), możemy jedynie dodać nowe zera, za pomocą wielomianu $T(q)$. Natomiast odpowiednio dobierając wielomiany $R(q)$ i $S(q)$, możemy całkowicie zmienić bieguny układu regulacji. Dlatego ta metoda to: **lokowanie biegunów** (*pole placement*). Naszym celem jest takie dobranie wielomianów $R(q)$ i $S(q)$, aby otrzymać wielomian $P(q)$ z zerami, które są założonymi biegunami układu regulacji, czyli chcemy aby spełniona była równość

$$A(q)R(q) + B(q)S(q) = P(q) \tag{21.13}$$

Równanie (21.13) jest nazywane równaniem diofantycznym i wygląda bardzo groźnie, ale jest łatwe do rozwiązania, gdyż możemy zamienić je na układ równań liniowych. Oczywistym warunkiem jest, aby rzędy wszystkich wielomianów były odpowiednio dobrane. Mniej oczywisty warunek istnienia rozwiązania jest taki, że wielomiany z transmitancji obiektu $A(q)$ i $B(q)$ nie mogą mieć wspólnych czynników, czyli jednakowych zer.

Budując regulator, sygnał sterujący będziemy wyliczać z równania regulatora, jak pokazano na rysunku 21.4, czyli

$$u = \frac{T(q)}{R(q)}y^{zad} - \frac{S(q)}{R(q)}y$$

Wymóg przyczynowości nakłada warunek, aby w obu przypadkach rząd wielomianu w mianowniku był nie mniejszy od rzędu wielomianu w liczniku transmitancji:

$$degree[R(q)] \geqslant degree[T(q)], \qquad degree[R(q)] \geqslant degree[S(q)]$$

Gdy, zgodnie ze wzorem (21.13), dobierzemy wielomiany $R(q)$ i $S(q)$, to transmitancja układu regulacji (21.12) będzie miała postać

$$y = \frac{B(q)T(q)}{P(q)}y^{zad}$$

Wielomian charakterystyczny $P(q)$ możemy rozłożyć na iloczyn dwóch wielomianów $P(q) = P_c(q)P_o(q)$. Ten rozkład odpowiada podziałowi na sprzężenie od stanu i na obserwator stanu. Dlatego przyjmiemy oznaczenia $P_c(q)$ to *controller polynomial* i $P_o(q)$ to *observer polynomial*. Wiemy z teorii sterowania, że możemy dobrać niezależnie bieguny regulatora $P_c(q)$, gdy obiekt jest sterowalny i bieguny obserwatora $P_o(q)$, gdy obiekt jest obserwowalny i że te bieguny nie wpływają na siebie.

$$y = \frac{B(q)T(q)}{P(q)}y^{zad} = \frac{B(q)T(q)}{P_c(q)\cdot P_o(q)}y^{zad}$$

Pozostaje jeszcze wybór wielomianu $T(q)$. Jeżeli przyjmiemy $T(q) = K \cdot P_o(q)$, to zostaną usunięte bieguny obserwatora. Bieguny układu zamkniętego musimy więc wybrać tak, aby

bieguny obserwatora $P_o(q)$ były silnie stabilne, czyli blisko zera w dodatniej części okręgu jednostkowego. Bieguny regulatora $P_c(q)$ wybieramy z jednym dominującym, który będzie decydował o zachowaniu układu regulacji. Uzyskaliśmy tansmitancję układu zamkniętego

$$y = \frac{K \cdot B(q)}{P_c(q)} y^{zad} \tag{21.14}$$

Wybór wartości parametru K wymaga ustalenia wzmocnienia w stanie ustalonym. W tym celu pobudzamy obiekt skokiem jednostkowym $\mathbf{1}(t)$ i obserwujemy wartość wyjścia dla $t \to \infty$. Wykorzystamy **twierdzenie o wartości końcowej** (*final value theorem*). Dla układów ciągłych (analogowych):

$$K_a = \lim_{t \to \infty} f(t) = \lim_{s \to 0} sX(s)$$

Ale ponieważ transformata skoku jednostkowego $\mathcal{L}\{\mathbf{1}(t)\} = 1/s$, więc wzmocnienie statyczne układu ciągłego jest równe

$$K_a = \lim_{s \to 0} X(s)$$

Ponieważ, gdy $s \to 0$, to $z \to 1$, więc wzmocnienie statyczne układu dyskretnego (gdy nie ma biegunów poza okręgiem jednostkowym), to

$$K_d = \lim_{z \to 1} X(z) = X(1)$$

Chcemy, aby w stanie ustalonym, wzmocnienie zamkniętej pętli regulacji było jednostkowe, dlatego wartość współczynnika K w transmitancji (21.14) wybierzemy jako równą

$$K = \frac{P_c(1)}{B(1)}$$

21.2.1. Wzory Viéte'a

Wielomian stopnia n o współczynnikach $P_{n-1} \dots P_0$ ma n pierwiastków $p_n \dots p_1$:

$$z^n + P_{n-1} z^{n-1} + \dots + P_1 + P_0 = (z - p_n)(z - p_{n-1}) \dots (z - p_2)(z - p_1)$$

Wzory Viéte'a określają wzajemne relacje pomiędzy pierwiastkami p_i, a współczynnikami P_j wielomianu

$$\begin{aligned}
p_1 + p_2 + \dots + p_n &= \sum_{i=1}^{n} p_i = -P_{n-1} \\
p_1 p_2 + p_1 p_3 + \dots + p_{n-1} p_n &= \sum_{i,j=1, i<j}^{n} p_i p_j = P_{n-2} \\
p_1 p_2 p_3 + p_1 p_2 p_4 + \dots + p_{n-2} p_{n-1} p_n &= \sum_{i,j,k=1, i<j<k}^{n} p_i p_j p_k = -P_{n-3} \\
\dots \qquad & \dots \\
p_1 p_2 \dots p_n &= (-1)^n P_0
\end{aligned}$$

Przykłady

$$
\begin{aligned}
&z^3 + z^2P_2 + zP_1 + P_0 = (z-p_1)(z-p_2)(z-p_3)\\
&P_2 = -p_1 - p_2 - p_3\\
&P_1 = p_1p_2 + p_2p_3 + p_1p_3\\
&P_0 = -p_1p_2p_3
\end{aligned}
$$

$$
\begin{aligned}
&z^4 + z^3P_3 + z^2P_2 + zP_1 + P_0 = (z-p_1)(z-p_2)(z-p_3)(z-p_4)\\
&P_3 = -p_1 - p_2 - p_3 - p_4\\
&P_2 = p_1p_2 + p_1p_3 + p_1p_4 + p_2p_3 + p_2p_4 + p_3p_4\\
&P_1 = -p_1p_2p_3 - p_1p_2p_4 - p_1p_3p_4 - p_2p_3p_4\\
&P_0 = p_1p_2p_3p_4
\end{aligned}
$$

$$
\begin{aligned}
&z^5 + z^4P_4 + z^3P_3 + z^2P_2 + zP_1 + P_0 = (z-p_1)(z-p_2)(z-p_3)(z-p_4)(z-p_5)\\
&P_4 = -p_1 - p_2 - p_3 - p_4 - p_5\\
&P_3 = p_1p_2 + p_1p_3 + p_1p_4 + p_1p_5 + p_2p_3 + p_2p_4 + p_2p_5 + p_3p_4 + p_3p_5 + p_4p_5\\
&P_2 = -p_1p_2p_3 - p_1p_2p_4 - p_1p_2p_5 - p_1p_3p_4 - p_1p_3p_5 - p_1p_4p_5 - p_2p_3p_4 - p_2p_3p_5 -\\
&\qquad\qquad - p_2p_4p_5 - p_3p_4p_5\\
&P_1 = p_1p_2p_3p_4 + p_1p_2p_3p_5 + p_1p_2p_4p_5 + p_1p_3p_4p_5 + p_2p_3p_4p_5\\
&P_0 = -p_1p_2p_3p_4p_5
\end{aligned}
$$

i tak dalej ...

Przykład 21.1. *Regulacja obiektu z podwójnym całkowaniem.*

Obiekt regulacji ma transmitancję operatorową $H(s) = 1/s^2$. Zakładamy okres próbkowania modelu cyfrowego tego obiektu równy 50 milisekund, $T_s = 0.05$. Emulacja obiektu metodą ZOH (domyślna dla funkcji `c2d` MatLaba) daje transmitancję dyskretną

$$H(z) = \frac{0.00125z + 0.00125}{z^2 - 2z + 1} = \frac{T_s^2}{2} \cdot \frac{z+1}{z^2 - 2z + 1} = \frac{B(z)}{A(z)} \tag{21.15}$$

Korzystając z projektu *Projekt24_obiekt*, na jednej z płyt STM3210E-EVAL należy zaimplementować obiekt (21.15). Na drugiej zrealizujemy regulator.

Przyjmiemy najprostszą możliwą postać wielomianów regulatora

$$R(z) = r_0z + r1, \qquad S(z) = s_0z + s_1$$

Napiszemy teraz równanie diofantyczne (21.13), w którym oznaczymy $h = T_s$:

$$A(z)R(z) + B(z)S(z) = (z^2 - 2z + 1)(r_0z + r_1) + \frac{h^2}{2}(z+1)(s_0z + s_1) = P(z)$$

Po przekształceniu otrzymujemy

$$z^3r_0 + z^2\Big[r_1 - 2r_0 + \frac{h^2}{2}s_0\Big] + z\Big[r_0 - 2r_1 + \frac{h^2}{2}(s_0 + s_1)\Big] + \Big[r_1 + \frac{h^2}{2}s_1\Big] = P(z) \tag{21.16}$$

Jest to wielomian trzeciego stopnia, wybieramy więc trzy bieguny $p_1,\ p_2,\ p_3$

$$P(z) = (z - p_1)(z - p_2)(z - p_3) = z^3 + z^2 P_2 + zP_1 + P_0 \tag{21.17}$$

Z porównania współczynników przy najwyższej potędze we wzorach (21.16) i (21.17) wynika, że $r_0 = 1$. Po uwzględnieniu wartości $r_0 = 1$, równanie (21.16) otrzymuje postać:

$$z^3 + z^2\Big[r_1 - 2 + \frac{h^2}{2}s_0\Big] + z\Big[1 - 2r_1 + \frac{h^2}{2}(s_0 + s_1)\Big] + \Big[r_1 + \frac{h^2}{2}s_1\Big] = P(z) \tag{21.18}$$

Na podstawie wzorów Viéte'a wyliczamy współczynniki $P_2,\ P_1,\ P_0$. Porównując współczynniki przy jednakowych potęgach z we wzorach 21.17 i 21.18 otrzymujemy układ równań

$$\begin{aligned} r_1 + \frac{h^2}{2}s_0 &= P_2 + 2 \\ -2r_1 + \frac{h^2}{2}(s_0 + s_1) &= P_1 - 1 \\ r_1 + \frac{h^2}{2}s_1 &= P_0 \end{aligned}$$

którego rozwiązaniem jest

$$\begin{aligned} r_1 &= \frac{P_2 - P_1 + P_0 + 3}{4} \\ s_0 &= \frac{3P_2 + P_1 - P_0 + 5}{2h^2} \\ s_1 &= \frac{-P_2 + P_1 + 3P_0 - 3}{2h^2} \end{aligned}$$

Wielomian $P(z)$ rozdzielimy na $P_o(z) \cdot P_c(z) = (z - p_1) \cdot (z - p_2)(z - p_3)$. Przyjmiemy $P_o(z) = T(z) = z - p_1$. Tramsmitancja układu regulacji (21.14)

$$H_1 = \frac{K \cdot B(q)}{P_c(q)} = K\frac{h^2}{2}\frac{z + 1}{(z - p_2)(z - p_3)}$$

Możemy wyliczyć wartość współczynnika K

$$K = \frac{P_c(1)}{B(1)} = \frac{2(1 - p_2)(1 - p_3)}{2h^2} = \frac{(1 - p_2)(1 - p_3)}{h^2}$$

Uzyskaliśmy prawo regulacji (rysunek 21.4)

$$R(z)U(z) = T(z)Y^{zad}(z) - S(z)Y(z)$$

$$\begin{aligned} (z + r_1)U(z) &= K(z - p_1)Y^{zad}(z) - (s_0 z + s_1)Y(z) \quad | \cdot z^{-1} \\ (1 + r_1 z^{-1})U(z) &= K(1 - p_1 z^{-1})Y^{zad}(z) - (s_0 + s_1 z^{-1})Y(z) \end{aligned}$$

Równanie różnicowe regulatora

$$u(k) = -r_1 u(k - 1) + Ky^{zad}(k) - Kp_1 y^{zad}(k - 1) - s_0 y(k) - s_1 y(k - 1) \tag{21.19}$$

Jak widać, równanie różnicowe (21.19) jest bardzo proste i łatwiejsze w implementacji nawet od regulatora PID.

21.2.2. Projekt31_1_pole_placement

Zmieniając równanie różnicowe w Projekt30_PID na (21.19) utworzymy regulator z lokowaniem biegunów.

W rozdziale 21.1 obliczyliśmy transformatę Laplace'a z sumy impulsów (21.2), a następnie stosując podstawienie $z = e^{sT_s}$ uzyskaliśmy definicję transformaty $\mathcal{Z}$ (21.3).

$$z = e^{sT_s} \qquad \rightarrow \qquad z = e^{(\sigma+j\omega)T_s} = e^{\sigma T_s} e^{j\omega T_s} = e^{\sigma T_s}\cos(\omega T_s) + je^{\sigma T_s}\sin(\omega T_s)$$

Zauważmy, że jeżeli współczynnik tłumienia $\sigma = 0$, to

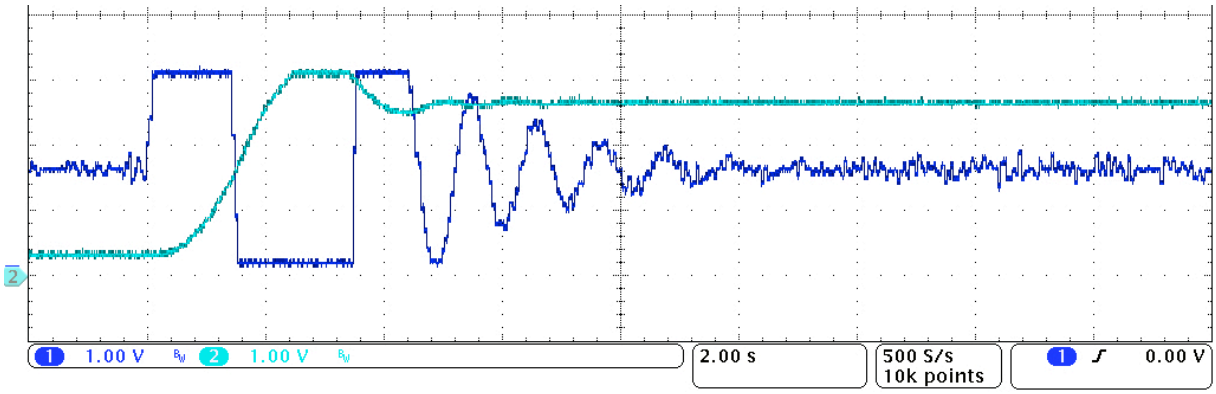
$$\sigma = 0 \qquad \rightarrow \qquad e^{(0+j\omega)T_s} = 1 \cdot e^{j\omega T_s} = \cos(\omega T_s) + j\sin(\omega T_s)$$

Jest to równanie okręgu jednostkowego na płaszczyźnie zespolonej z. Bieguny położone na okręgu jednostkowym będą generować sygnał o stałej amplitudzie (rysunek 21.2c). Bieguny wewnątrz okręgu jednostkowego sa stabilne i generują sygnał zanikający, bieguny na zewnątrz są niestabilne. Biegun rzeczywisty generuje sygnał stały lub stały naprzemienny. Para biegunów zespolonych sprzężonych to $e^{j\omega T_s} + e^{-j\omega T_s} = 2\cos(\omega T_s)$. Wielokrotny biegun położony na okręgu jednostkowym, także powoduje niestabilność (rysunek 21.2d). Wnętrze koła jednostkowego odpowiada więc lewej półpłaszczyźnie układów ciągłych opisanych transmitancją Laplace'a. Jeżeli biegun układu ciągłego przesuwamy do minus nieskończoności, to dla układów dyskretnych odpowiada to biegunowi w zerze, gdyż $e^{-\infty} = 0$. Bieguny z ujemną częścią rzeczywistą, generują sygnały naprzemienne, mówimy, że są to **bieguny dzwoniące**. Dlatego będziemy wybierali tylko bieguny z dodatnią częścią rzeczywistą (prawy półokrąg).

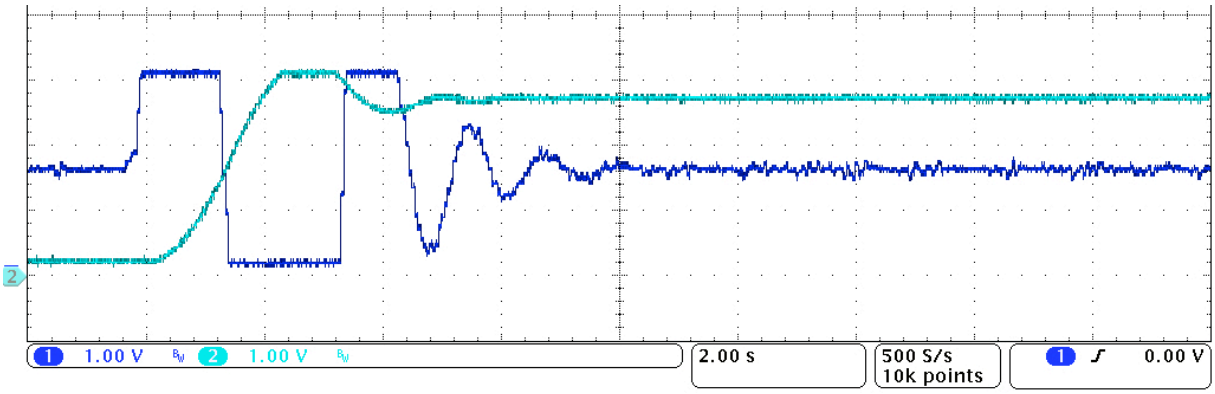
Korzystając ze środowiska MatLab wyznaczymy teraz parametry regulatora opisanego w przykładzie 21.1. Odpowiedzi na skok wartości zadanej pokazane są na rysunkach od 21.5 do 21.8. Widać, że regulator jest bardzo wrażliwy na dobór biegunów. Bieguny rzeczywiste bliższe zeru powodują agresywne sterowanie, natomiast zbliżone do jedynki – sterowanie bardzo spokojne, aperiodyczne. Dobór biegunów jest więc krytyczny.

Tabela 21.1: Parmetry regulatora *pole placement* z przykładu 21.1

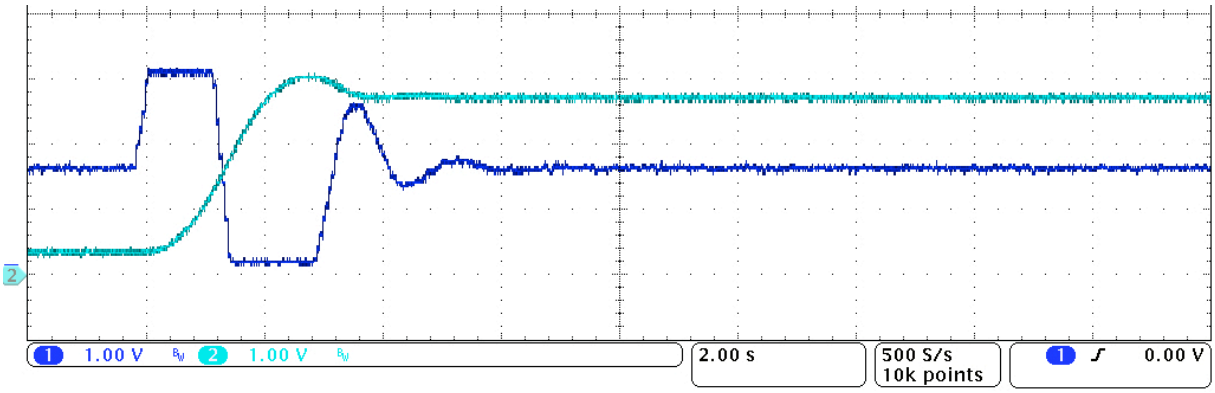
p_1	p_2	p_3	K	r_1	s_0	s_1	rysunek
0.6	0.6	0.9	16	-0.216	92.8	-86.4	21.5
0.7	0.7	0.9	12	-0.37275	58.2	-54.6	21.6
0.8	0.8	0.9	8	-0.539	31.2	-29.6	21.7
0.9	0.9	0.975	1	-0.7824375	5.95	-5.85	21.8



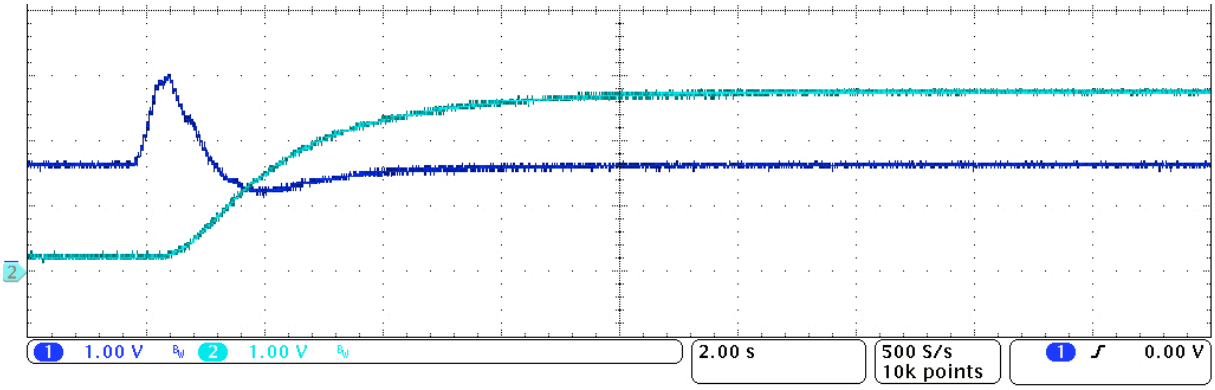
Rys. 21.5: Bieguny $p_1 = 0.6,\ p_2 = 0.6,\ p_3 = 0.9$



Rys. 21.6: Bieguny $p_1 = 0.7,\ p_2 = 0.7,\ p_3 = 0.9$



Rys. 21.7: Bieguny $p_1 = 0.8,\ p_2 = 0.8,\ p_3 = 0.9$



Rys. 21.8: Bieguny $p_1 = 0.9,\ p_2 = 0.9,\ p_3 = 0.975$

Na rysunkach powyżej kanał 1 to sterowanie, kanał 2 to odpowiedź obiektu.

21.3. Zwiększenie odporności na zakłócenia

Układ regulacji z lokowaniem biegunów pokazany jest na rysunku 21.4. Obliczmy oryginalne wyjście obiektu $x(t)$. To wyjście, z dodanym szumem pomiarowym $v(t)$ jest

zawracane do regulatora jako sygnał $y(t)$.

$$x = \frac{B}{A}\left[w + \frac{T}{R}y^{zad} - \frac{S}{R}(x+v)\right] = \frac{B}{A}w + \frac{BT}{AR}y^{zad} - \frac{BS}{AR}x - \frac{BS}{AR}v$$

$$ARx + BSx = BRw + BTy^{zad} - BSv$$

$$x = \frac{BT}{AR+BS}y^{zad} + \frac{BR}{AR+BS}w - \frac{BS}{AR+BS}v \tag{21.20}$$

21.3.1. Zakłócenie stałe odniesione do wejścia obiektu

Zacznijmy od sytuacji, gdy sygnał zakłócający $w(t)$ ma stałą wartość: $w(t) = constans$. Uchyb nie wystąpi, gdy wzmocnienie statyczne transmitancji $BR/(AR+BS)$ będzie zerowe. To prowadzi do wymagania $R(1) = 0$, czyli wielomian R musi mieć czynnik $(z-1)$. Transmitancja regulatora jest określona wzorem

$$u = \frac{T}{R}y^{zad} - \frac{S}{R}y$$

Wymaganie $R = (z-1)(\ldots)$ oznacza biegun w jedynce, czyli akcję całkującą regulatora.

Przykład 21.2. *Regulator z wewnętrznym integratorem.*

Zbudujemy teraz regulator sterujący tym samym obiektem, ale w celu poprawy odporności na zakłócenia o stałym poziomie, dodamy do niego wewnętrzny integrator. To oznacza, że wielomian $R(z)$ musi mieć czynnik $z-1$.

$$R(z) = (z-1)(z+r_1), \qquad S(z) = s_0z^2 + s_1z + s_2$$

Równanie diofantyczne (21.13) ma postać (znów oznaczymy $h = T_s$)

$$AR + BS = (z^2-2z+1)(z-1)(z+r_1) + \frac{h^2}{2}(z+1)(s_0z^2+s_1z+s_2) = P(z)$$

Po przekształceniu otrzymujemy

$$z^4 + \Big(r_1 - 3 + \frac{h^2}{2}s_0\Big)z^3 + \Big(3 - 3r_1 + \frac{h^2}{2}(s_0+s_1)\Big)z^2 + \Big(3r_1 - 1 + \frac{h^2}{2}(s_1+s_2)\Big)z + \Big(\frac{h^2}{2}s_2 - r_1\Big) = P(z)$$

Jest to wielomian czwartego stopnia, wybieramy więc cztery bieguny $p_1,\ p_2,\ p_3,\ p_4$

$$P(z) = (z-p_1)(z-p_2)(z-p_3)(z-p_4) = z^4 + z^3P_3 + z^2P_2 + zP_1 + P_0$$

Na podstawie wzorów Viéte'a wyliczamy współczynniki $P_3,\ P_2,\ P_1,\ P_0$. Porównując współczynniki przy jednakowych potęgach z, otrzymujemy układ równań

$$\begin{aligned}
r_1 + \frac{h^2}{2}s_0 &= P_3 + 3\\
-3r_1 + \frac{h^2}{2}(s_0+s_1) &= P_2 - 3\\
3r_1 + \frac{h^2}{2}(s_1+s_2) &= P_1 + 1\\
-r_1 + \frac{h^2}{2}s_2 &= P_0
\end{aligned}$$

który przedstawimy w postaci macierzowej

$$\begin{bmatrix} 1 & h^2/2 & 0 & 0 \\ -3 & h^2/2 & h^2/2 & 0 \\ 3 & 0 & h^2/2 & h^2/2 \\ -1 & 0 & 0 & h^2/2 \end{bmatrix} \cdot \begin{bmatrix} r_1 \\ s_0 \\ s_1 \\ s_2 \end{bmatrix} = \begin{bmatrix} P_3+3 \\ P_2-3 \\ P_1+1 \\ P_0 \end{bmatrix}$$

W środowisku MatLab, powyższy układ równań $Ax = b$, rozwiązujemy poleceniem *backslash*, czyli `x = A\b`. Wielomian $P(z)$ rozdzielimy na

$$P_o(z) \cdot P_c(z) = (z-p_1)(z-p_2) \cdot (z-p_3)(z-p_4)$$

Przyjmiemy $P_o(z) = T(z) = (z-p_1)(z-p_2)$. Tramsmitancja układu regulacji (21.14)

$$H_1 = \frac{K \cdot B(q)}{P_c(q)} = K\frac{h^2}{2}\frac{z+1}{(z-p_3)(z-p_4)}$$

Możemy wyliczyć wartość współczynnika K

$$K = \frac{P_c(1)}{B(1)} = \frac{2(1-p_3)(1-p_3)}{2h^2} = \frac{(1-p_3)(1-p_4)}{h^2}$$

Uzyskaliśmy prawo regulacji (zgodnie z rysunkiem 21.4)

$$R(z)U(z) = T(z)Y^{zad}(z) - S(z)Y(z)$$

$$\Big(z^2+z(r_1-1)-r_1\Big)U(z) = K\Big(z^2-z(p_1+p_2)+p_1p_2\Big)Y^{zad}(z) - \Big(s_0z^2+s_1z+s_2\Big)Y(z) \quad |\cdot z^{-2}$$

$$\Big(1+(r_1-1)z^{-1}-r_1z^{-2}\Big)U(z) = K\Big(1-(p_1+p_2)z^{-1}+p_1p_2z^{-2}\Big)Y^{zad} - \Big(s_0+s_1z^{-1}+s_2z^{-2}\Big)Y(z)$$

Równanie różnicowe regulatora

$$\begin{aligned} u(k) = {} & (1-r_1)u(k-1) + r_1u(k-2) \\ & +Ky^{zad}(k) - K(p_1+p_2)y^{zad}(k-1) + Kp_1p_2y^{zad}(k-1) \\ & -s_0y(k) - s_1y(k-1) - s_2y(k-2) \end{aligned}$$

21.3.2. Szum pomiarowy

Przypomnijmy wzór (21.20)

$$x = \frac{BT}{AR+BS}y^{zad} + \frac{BR}{AR+BS}w - \frac{BS}{AR+BS}v$$

Redukcję wpływu szumu pomiarowego $v(t)$ na wyjście obiektu $x(t)$ możemy osiągnąć przez odpowiedni dobór wielomianu $S(z)$. Jeżeli nie występuje błąd systematyczny pomiaru, to $v(t)$ jest szumem białym, czyli sygnałem szerokopasmowym z zerową wartością oczekiwaną. Jeden z możliwych sposobów redukcji wpływu tego szumu to dodanie czynnika $(z+1)$ do wielomianu $S(z)$. Przykład transmitancji z zerem w minus jeden:

$$\frac{Y(z)}{U(z)} = \frac{z+1}{2z} = \frac{1+z^{-1}}{2} \qquad \rightarrow \qquad y(k) = \frac{u(k)+u(k-1)}{2}$$

Jest to średnia arytmetyczna z dwóch kolejnych próbek. Zero transmitancji równe (-1) jest więc filtrem górno-zaporowym (tłumi wysokie częstotliwości). Czynnik $(z+1)$ dodany do wielomianu $S(z)$ przyczyni się do tłumienia szumu pomiarowego $v(t)$.

Przykład 21.3. *Regulator z wewnętrznym integratorem i redukcją szumu pomiarowego.*

Do obiektu z podwójnym całkowaniem zbudujemy regulator z czynnikiem $(z-1)$ w wielomianie $R(z)$ i czynnikiem $(z+1)$ w $S(z)$.

$$R(z) = (z-1)(z^2 + r_1 z + r_2), \qquad S(z) = (z+1)(s_0 z^2 + s_1 z + s_2)$$

Równanie diofantyczne (21.13) ma postać ($h = T_s$)

$$AR + BS = (z^2 - 2z + 1)(z-1)(z^2 + r_1 z + r_2) + \frac{h^2}{2}(z+1)(z+1)(s_0 z^2 + s_1 z + s_2) = P(z)$$

Po przekształceniu otrzymujemy

$$P(z) = z^5 + z^4\Big[r_1 - 3 + \frac{h^2}{2}s_0\Big] + z^3\Big[r_2 - 3r_1 + 3 + \frac{h^2}{2}(s_1 + 2s_0)\Big] +$$
$$+ z^2\Big[-3r_2 + 3r_1 - 1 + \frac{h^2}{2}(s_2 + 2s_1 + s_0)\Big] + z\Big[3r_2 - r_1 + \frac{h^2}{2}(2s_2 + s_1)\Big] - r_2 + \frac{h^2}{2}s_2$$

Jest to wielomian piątego stopnia, wybieramy więc pięć biegunów p_1, p_2, p_3, p_4, p_5. Na podstawie wzorów Viéte'a wyliczamy współczynniki P_4, P_3, P_2, P_1, P_0. Porównując współczynniki przy jednakowych potęgach z, otrzymujemy układ równań

$$\begin{aligned}
r_1 + \frac{h^2}{2}s_0 &= P_4 + 3 \\
r_2 - 3r_1 + \frac{h^2}{2}(s_1 + 2s_0) &= P_3 - 3 \\
-3r_2 + 3r_1 + \frac{h^2}{2}(s_2 + 2s_1 + s_0) &= P_2 + 1 \\
3r_2 - r_1 + \frac{h^2}{2}(2s_2 + s_1) &= P_1 \\
-r_2 + \frac{h^2}{2}s_2 &= P_0
\end{aligned}$$

który przedstawimy w postaci macierzowej

$$\begin{bmatrix} 0 & 1 & 0 & 0 & h^2/2 \\ 1 & -3 & 0 & h^2/2 & h^2 \\ -3 & 3 & h^2/2 & h^2 & h^2/2 \\ 3 & -1 & h^2 & h^2/2 & 0 \\ -1 & 0 & h^2/2 & 0 & 0 \end{bmatrix} \cdot \begin{bmatrix} r_2 \\ r_1 \\ s_2 \\ s_1 \\ s_0 \end{bmatrix} = \begin{bmatrix} P_4 + 3 \\ P_3 - 3 \\ P_2 + 1 \\ P_1 \\ P_0 \end{bmatrix}$$

W środowisku MatLab, układ równań $Ax = b$, rozwiązujemy `x = A\b`. Wielomian $P(z)$ rozdzielimy na

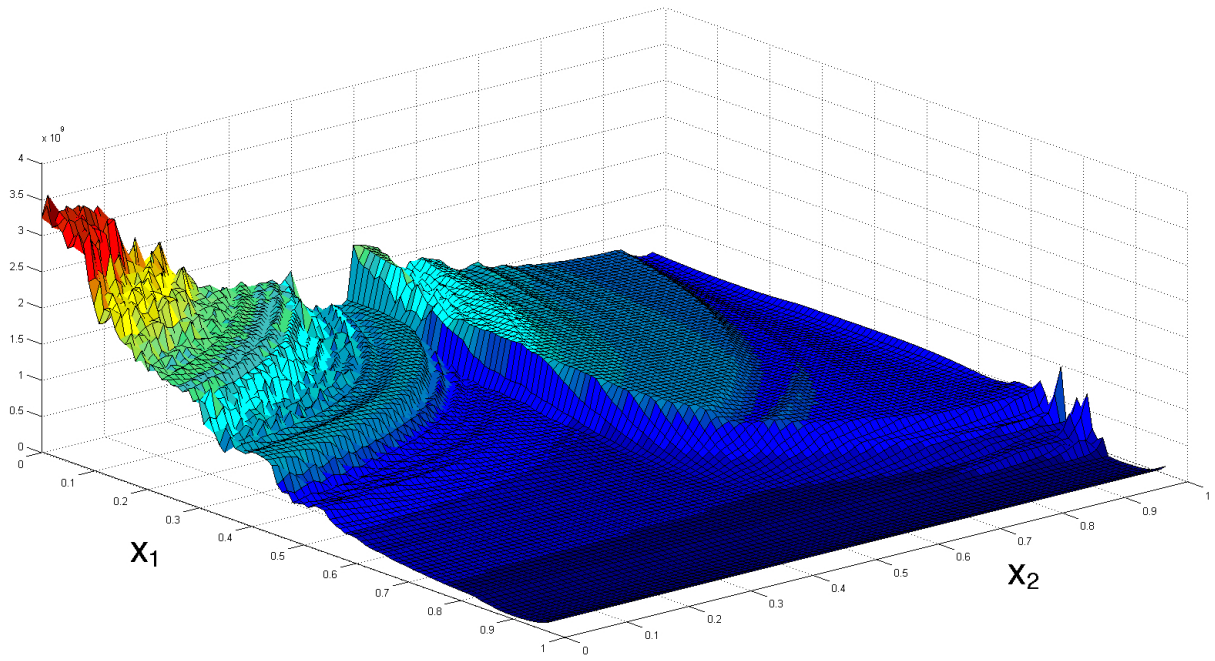
$$P_o(z) \cdot P_c(z) = (z - p_1)(z - p_2) \cdot (z - p_3)(z - p_4)(1 - p_5)$$

Przyjmiemy $P_o(z) = T(z) = (z - p_1)(z - p_2)$. Tramsmitancja układu regulacji (21.14)

$$H_1 = \frac{K \cdot B(q)}{P_c(q)} = K\frac{h^2}{2}\frac{z+1}{(z-p_3)(z-p_4)(z-p_5)}$$

Możemy wyliczyć wartość współczynnika K

$$K = \frac{P_c(1)}{B(1)} = \frac{2(1-p_3)(1-p_4)(1-p_5)}{2h^2} = \frac{(1-p_3)(1-p_4)(1-p_5)}{h^2}$$



Rys. 21.9: Funkcja celu (21.22), dwie zmienne decyzyjne: $x_1 = p_1 = p_2,\ x_2 = p_3 = p_4 = p_5$

Uzyskaliśmy prawo regulacji (zgodnie z rysunkiem 21.4)

$$\begin{aligned} R(z)U(z) &= T(z)Y^{zad}(z) - S(z)Y(z) \\ \Big[z^3 + z^2(r_1 - 1) + z(r_2 - r_1) - r_2\Big]U(z) &= K\Big[z^2 - z(p_1 + p_2) + p_1p_2\Big]Y^{zad}(z) - \\ &- \Big[s_0z^3 + z^2(s_1 + s_0) + z(s_1 + s_2) + s_2\Big]Y(z) \end{aligned}$$

Równanie różnicowe regulatora

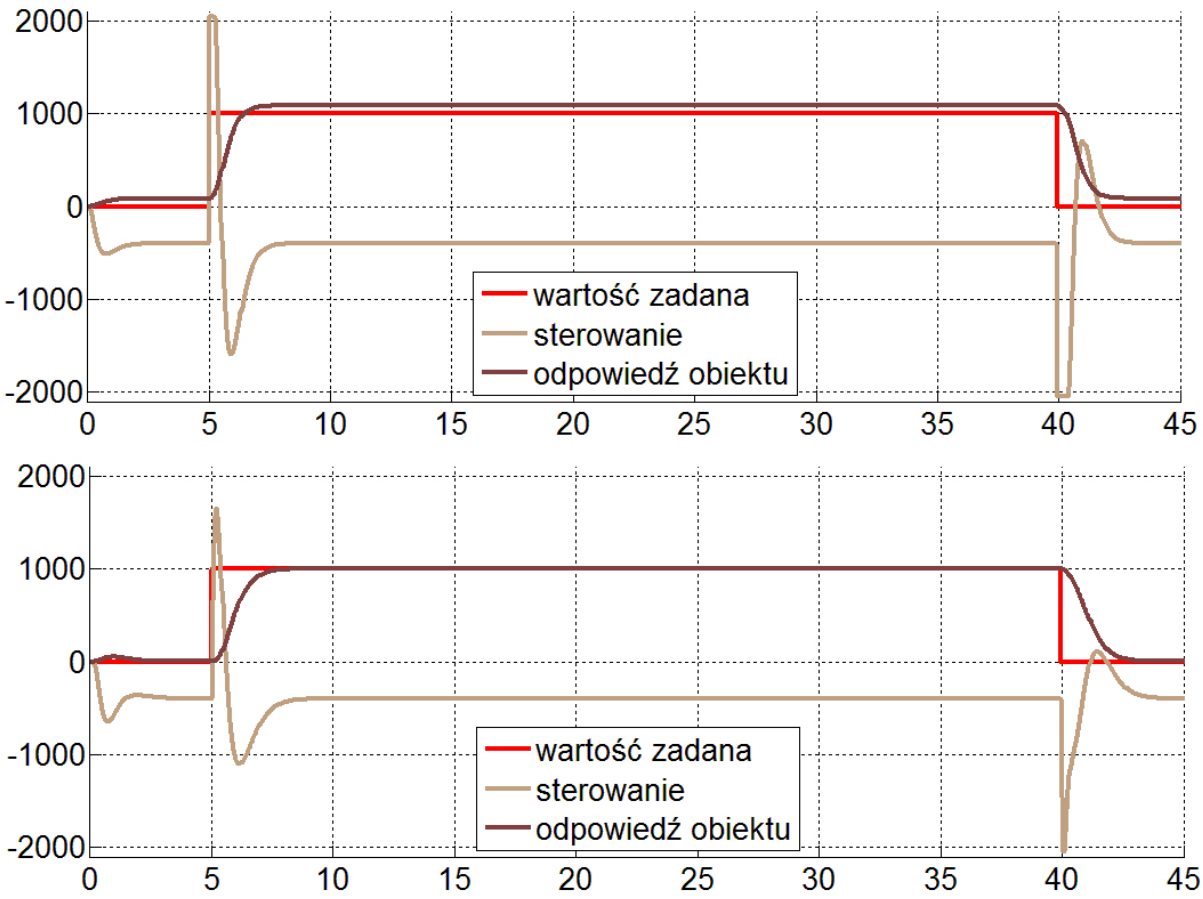
$$\begin{aligned} u(k) = {} & (1 - r_1)u(k-1) + (r_1 - r_2)u(k-2) + r_2u(k-3) + \\ & + Ky^{zad}(k-1) - K(p_1 + p_2)y^{zad}(k-2) + Kp_1p_2y^{zad}(k-3) + \\ & - s_0y(k) - (s_1 + s_0)y(k-1) - (s_1 + s_2)y(k-2) - s_2y(k-3) \end{aligned} \tag{21.21}$$

Wykorzystamy środowisko MatLab do wyznaczenia biegunów oraz do porównania regulatorów z przykładów 21.1 i 21.3. Wartości biegunów wyznaczymy rozwiązując zadanie optymalizacji nieliniowej (bez ograniczeń) z funkcją celu:

$$J = \sum_{k=1}^{N} \Big(y^{zad}(k) - y(k)\Big)^2 + \lambda \sum_{k=2}^{N} (u(k) - u(k-1))^2 \tag{21.22}$$

gdzie N to horyzont symulacji, a λ to współczynnik kary za zmienność sterowania .Musimy założyć pewną trajektorię wartości zadanej $y^{zad}(k)$ oraz punkt początkowy optymalizacji, na przykład $[0.7,\ 0.7,\ 0.8,\ 0.8,\ 0.9]$. Zakładając, równość biegunów obserwatora $p_1 = p_2$ oraz niezależnie biegunów regulatora $p_3 = p_4 = p_5$ możemy wykreślić płaszczyznę funkcji celu J (rysunek 21.9). Rysunek wykonano dla wartości parametru $\lambda = 1$.

Ilustracją działania wewnętrznego integratora są dwie symulacje MatLab pokazane na rysunku 21.10. Założono istnienie stałego zakłócenia jednostkowego $w(t) = 1$. Górna część rysunku 21.10 do odpowiedź obiektu z regulatorem z przykładu 21.1. Dolna część to wynik symulacji z regulatorem z przykładu 21.3. Widać korzystny wpływ wewnętrznego integratora, który całkowicie niweluje oddziaływanie zakłócenia o stałej wartości.



Rys. 21.10: Zakłócenie stałe $w(t) = 1$. Porównanie działania regulatorów: bez integratora (rysunek górny) i z integratorem (dolny)

Kolejne dwie symulacje wykonano dodając szum pomiarowy do wyjścia obiektu. Porównanie obu regulatorów pokazane jest na rysunku 21.11. Podobnie jak poprzednio, górna część rysunku 21.11 to wynik symulacji z regulatorem z przykładu 21.1, a dolna część to wynik symulacji z regulatorem z przykładu 21.3.

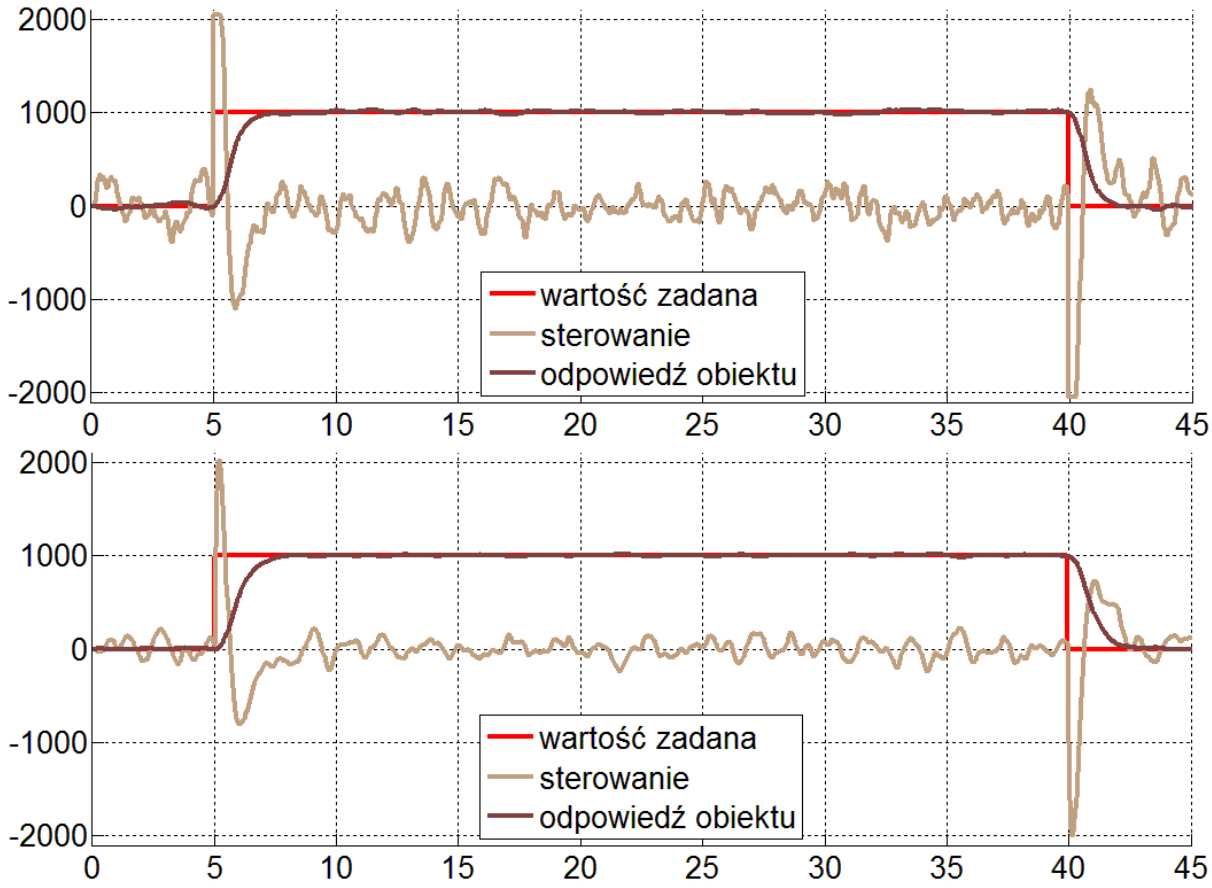
Niewielka poprawa jakości regulacji wynika z zastosowania czynnika $(z + 1)$ w wielomianie $S(z)$. Czynnik ten ma własności filtru dolnoprzepustowego. Jednak jest to filtr tylko pierwszego rzędu, dlatego jego wpływ na redukcję szumu pomiarowego jest niewielki. Układ wymaga dodania właściwie zaprojektowanego filtru dolnoprzepustowego. Taki filtr może całkowicie zniwelować negatywny wpływ szumu pomiarowego na jakość regulacji.

Równanie różnicowe regulatora z przykładu 21.1 opisane jest wzorem (21.19). W przypadku tego regulatora także możemy zdefiniować zadanie optymalizacji z funkcją celu (21.22). Przykładowe rozwiązanie:

```
// stałe globalne
const float K = 13.0759014334951;      // lambda = 1
const float r1 = -0.505149524504115, s0 = 38.0453330901474;
const float s1 = -35.6811972611663, p1 = 0.819199017291067;
```

Załóżmy, że szum pomiarowy ma postać $(\ldots + 1, -1, +1, -1 \ldots)$. Równanie różnicowe regulatora (21.19) przetwarza ten szum: $u(k) = \ldots - s_0 y(k) - s_1 y(k-1)$. Uzyskamy silne wzmocnienie szumu pomiarowego do poziomu $\pm(s_0 + s_1) = 73.7265$

$$\ldots +1, -1, +1, -1 \ldots \blacktriangleright \left[u(k) = \ldots - s_0 y(k) - s_1 y(k-1)\right] \blacktriangleright \ldots -73.7, +73.7, -73.7, +73.7$$



Rys. 21.11: Wpływ szumu pomiarowego na wynik regulacji: bez redukcji (rysunek górny) i z redukcją (dolny) – symulacja MatLab

To samo obliczenie wykonamy dla regulatora z wewnętrznym integratorem i redukcją szumu pomiarowego z przykładu 21.3, którego równanie różnicowe opisane jest wzorem (21.21). Przykładowe parametry regulatora uzyskane w wyniku rozwiązania zadania optymalizacji z funkcją celu (21.22):

```
// stałe globalne
const float K = 1.05158223181824;      // lambda = 0.1
const float r2 = 0.523693151129846, r1 = -1.39180500642087;
const float s2 = 3.188728114160, s1 = -6.578707794129, s0 = 3.39515340624;
const float p1 = 0.900908290939043, p2 = 0.90069916844619;
```

Fragment równania różnicowego (21.21) odpowiedzialny za wzmocnienie szumu pomiarowego ma postać:

$$u(k) = \ldots - s_0 y(k) - (s_1 + s_0)y(k-1) - (s_1 + s_2)y(k-2) - s_2 y(k-3)$$

Zakładając taką samą postać szumu pomiarowego $\ldots +1, -1, +1, -1 \ldots$

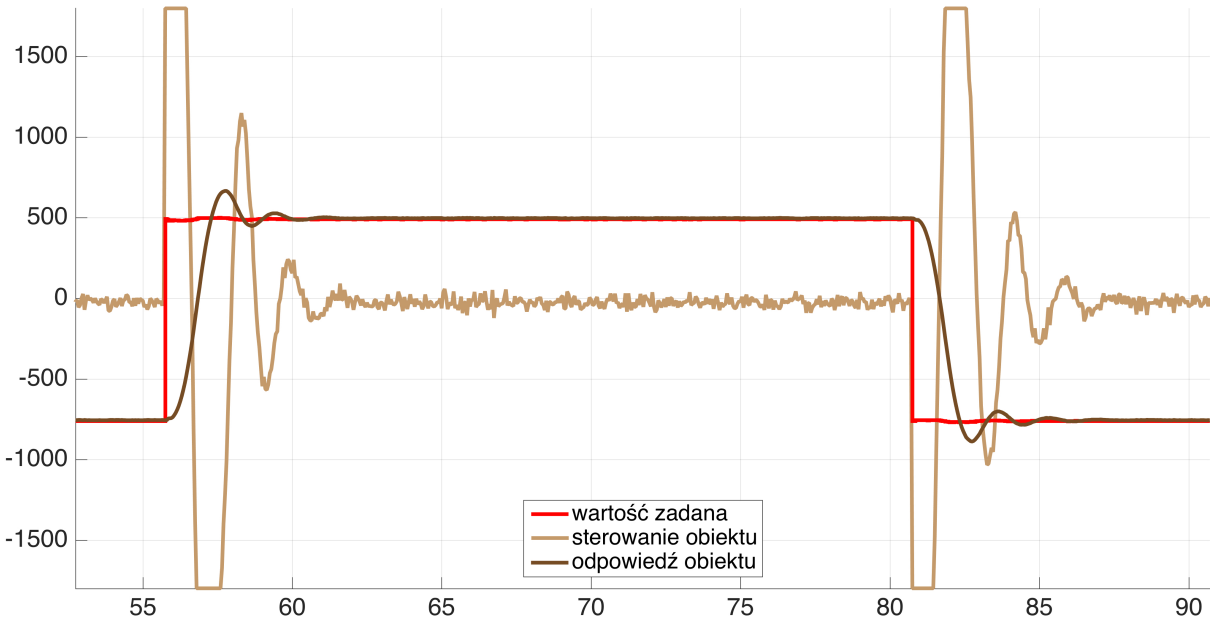
$$\ldots \blacktriangleright \Big[u(k) = \ldots -3.39y(k) + 3.19y(k-1) + 3.39y(k-2) - 3.19y(k-3)\Big] \blacktriangleright \ldots 0,\ 0,\ 0,\ 0 \ldots$$

uzyskujemy całkowitą jego redukcję.

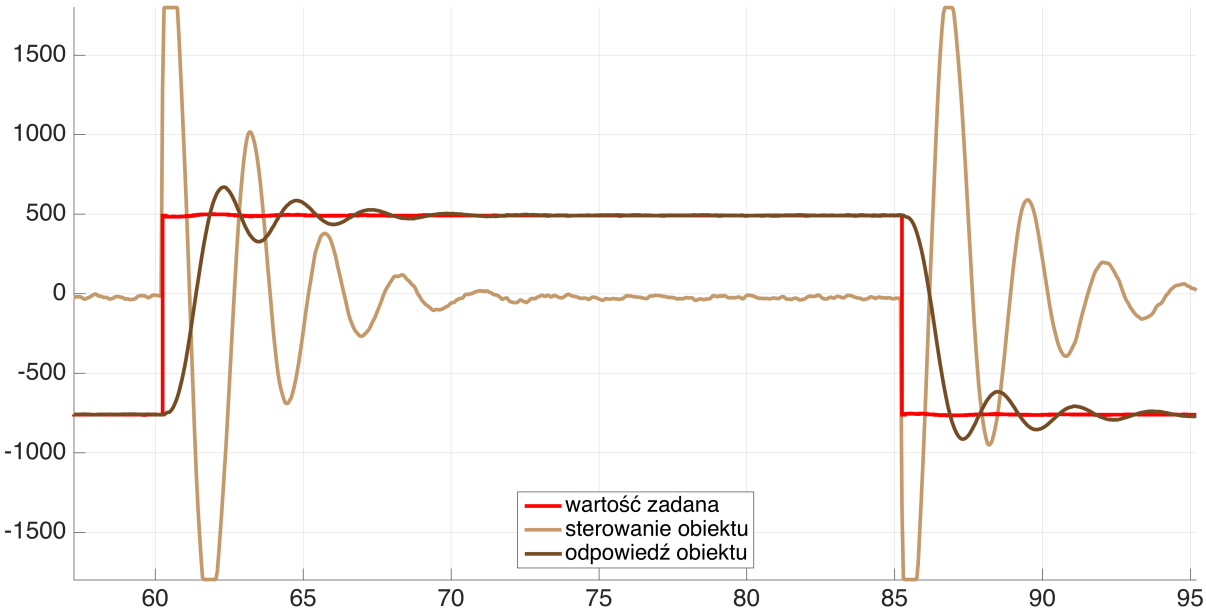
W układzie rzeczywistym szum nie jest ciągiem jedynek o przeciwnych znakach, więc redukcja nie jest tak idealna. Jednak przykład powyższy pokazuje mechanizm wzmacniania, albo osłabiania szumu pomiarowego, jaki może wystąpić w regulatorach z lokowaniem biegunów.

21.3.3. Projekt31_2_pole_placement

Modyfikując nieco *Projekt31_1_pole_placement* w zakresie prawa regulacji zrealizujemy regulator z przykładu 21.3. Redukcja szumu umożliwiła zmniejszenie wartości współczynnika kary za zmienność sterowania do poziomu $\lambda = 0.1$ i bardziej agresywną regulację, co widać na rysunku 21.13.



Rys. 21.12: Sygnały w układzie z regulatorem bez redukcji szumu pomiarowego, $\lambda = 1$



Rys. 21.13: Sygnały w układzie z redukcją szumu pomiarowego (przykład 21.3), $\lambda = 0.1$

21.3.4. Projekt31_3_pole_placement, obiekt oscylacyjny

Przykład 21.4. *Regulator z wewnętrznym integratorem i redukcją szumu pomiarowego.*

Zaprojektujemy teraz regulator do obiektu z *Projekt28_1_obiekt_oscylacyjny*. Obiekt regulacji ma transmitancję dyskretną

$$H(z) = \frac{b_1 z + b_2}{z^2 + a_1 z + a_2} = \frac{B(z)}{A(z)}$$

Wybieramy wersję regulatora z wewnętrznym integratorem i redukcją szumu pomiarowego:

$$R(z) = (z-1)(z^2 + r_1 z + r_2), \qquad S(z) = (z+1)(s_0 z^2 + s_1 z + s_2)$$

Równanie diofantyczne (21.13) ma postać ($h = T_s$)

$$AR + BS = (z^2 + a_1 z + a_2)(z-1)(z^2 + r_1 z + r_2) + (b_1 z + b_2)(z+1)(s_0 z^2 + s_1 z + s_2) = P(z)$$

Po przekształceniu otrzymujemy

$$\begin{aligned} P(z) = z^5 &+ z^4\Big[r_1 + a_1 - 1 + b_1 s_0\Big] + z^3\Big[r_2 + r_1(a_1 - 1) + a_2 - a_1 + s_1 b_1 + s_0(b_1 + b_2)\Big] + \\ &+ z^2\Big[r_2(a_1 - 1) + r_1(a_2 - a_1) - a_2 + s_2 b_1 + s_1(b_1 + b_2) + b_2 s_0\Big] + \\ &+ z\Big[r_2(a_2 - a_1) - a_2 r_1 + s_2(b_1 + b_2) + s_1 b_2\Big] - r_2 a_2 + b_2 s_2 \end{aligned}$$

Jest to wielomian piątego stopnia, wybieramy więc pięć biegunów p_1, p_2, p_3, p_4, p_5. Na podstawie wzorów Viéte'a wyliczamy współczynniki P_4, P_3, P_2, P_1, P_0. Porównując współczynniki przy jednakowych potęgach z, otrzymujemy układ równań, który przedstawimy w postaci macierzowej

$$\begin{bmatrix} 0 & 1 & 0 & 0 & b_1 \\ 1 & a_1 - 1 & 0 & b_1 & b_1 + b_2 \\ a_1 - 1 & a_2 - a_1 & b_1 & b_1 + b_2 & b_2 \\ a_2 - a_1 & -a_2 & b_1 + b_2 & b_2 & 0 \\ -a_2 & 0 & b_2 & 0 & 0 \end{bmatrix} \cdot \begin{bmatrix} r_2 \\ r_1 \\ s_2 \\ s_1 \\ s_0 \end{bmatrix} = \begin{bmatrix} P_4 - a_1 + 1 \\ P_3 + a_1 - a_2 \\ P_2 + a_2 \\ P_1 \\ P_0 \end{bmatrix}$$

W środowisku MatLab, układ równań $Ax = b$, rozwiązujemy `x = A\b`. Wielomian $P(z)$ rozdzielimy na

$$P_o(z) \cdot P_c(z) = (z - p_1)(z - p_2) \cdot (z - p_3)(z - p_4)(1 - p_5)$$

Przyjmiemy $P_o(z) = T(z) = (z - p_1)(z - p_2)$. Tramsmitancja układu regulacji (21.14)

$$H_1 = \frac{K \cdot B(q)}{P_c(q)} = K \frac{b_1 z + b_2}{(z - p_3)(z - p_4)(z - p_5)}$$

Możemy wyliczyć wartość współczynnika K

$$K = \frac{P_c(1)}{B(1)} = \frac{(1 - p_3)(1 - p_4)(1 - p_5)}{b_1 + b_2}$$

Uzyskaliśmy prawo regulacji identyczne, jak w przykładzie 21.3

$$\begin{aligned} R(z)U(z) &= T(z)Y^{zad}(z) - S(z)Y(z) \\ \Big[z^3 + z^2(r_1 - 1) + z(r_2 - r_1) - r_2\Big]U(z) &= K\Big[z^2 - z(p_1 + p_2) + p_1 p_2\Big]Y^{zad}(z) - \\ &- \Big[s_0 z^3 + z^2(s_0 + s_1) + z(s_1 + s_2) + s_2\Big]Y(z) \end{aligned}$$

oraz identyczne równanie różnicowe regulatora, które dane jest wzorem (21.21).

Podobnie, jak poprzednio, zdefiniujemy funkcję celu (21.22)

$$J = \sum_{k=1}^{N} \left(y^{zad}(k) - y(k)\right)^2 + \lambda \sum_{k=2}^{N} \left(u(k) - u(k-1)\right)^2$$

Płaszczyzna funkcji celu ma wiele minimów lokalnych, dlatego wynik optymalizacji zależy od punktu startowego. Obiekt oscylacyjny wymaga sterowania tłumiącego oscylacje, dlatego wybierzemy bieguny blisko jedynki: [0.9, 0.9, 0.95, 0.95, 0.95] oraz współczynnik kary za zmienność sterowania $\lambda = 1$. Uzyskane rozwiązanie:

```
// stałe globalne
const float K = 1.49186323310569;      // lambda = 1
const float r2 = 0.501444463529559, r1 = -1.35921697644422;
const float s2 = 3.265733441883, s1 = -6.739267752035, s0 = 3.481271267386;
const float p1 = 0.899083869201071, p2 = 0.897219532193124;
```

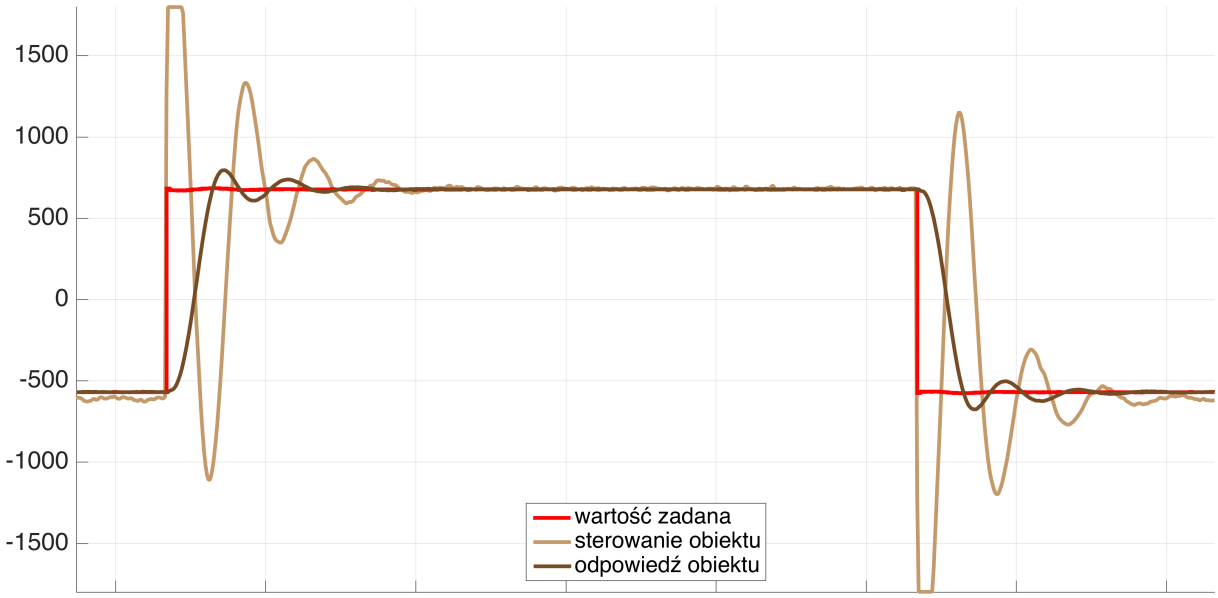
Załóżmy, że na wyjściu obiektu jest zero, a szum pomiarowy zastąpimy sygnałem plus/minus jeden. Fragment równania różnicowego (21.21) odpowiedzialny za wzmocnienie szumu pomiarowego ma postać:

$$u(k) = \ldots - s_0 y(k) - (s_0 + s_1) y(k-1) - (s_1 + s_2) y(k-2) - s_2 y(k-3)$$

Zakładając postać szumu pomiarowego $\ldots +1, -1, +1, -1 \ldots$ ▶

$$\ldots \blacktriangleright \left[u(k) = \ldots -3.48y(k) + 3.26y(k-1) + 3.48y(k-2) - 3.26y(k-3)\right] \blacktriangleright \ldots 0,\ 0,\ 0,\ 0 \ldots$$

uzyskujemy całkowitą jego redukcję. Porównanie rysunku 21.14 z rysunkiem 20.18 (regulacja PID tego samego obiektu) pokazuje wyraźną przewagę regulacji z lokowaniem biegunów oraz redukcją szumu pomiarowego i zakłóceń o stałym poziomie.



Rys. 21.14: Sygnały w układzie z redukcją szumu pomiarowego (przykład 21.4), $\lambda = 1$

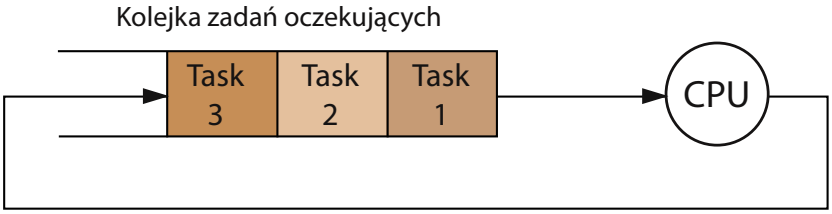
Rozdział 22

RTOS Real Time Operating Sysytem

System operacyjny czasu rzeczywistego (*real-time operating system*) RTOS tworzy się do zarządzania programami obsługującymi jednocześnie kilka zadań (*tasks*). Cały system musi spełnić (z pewnym marginesem bezpieczeństwa) wymagania czasowe narzucane przez te sterowane /regulowane zadania, czyli taski. Rolą RTOS jest szeregowanie (ustalanie kolejności obsługi) oraz podział czasu, pomiędzy zadania oczekujące w kolejce do obsłużenia (rysunek 22.1). W systemie operacyjnym czasu rzeczywistego trzeba określić, któremu z zadań należy przydzielić procesor oraz na jak długi czas, aby wszystkie wykonywane procesy spełniały zdefiniowane dla nich ograniczenia czasowe. Istnieje wiele algorytmów przydziału czasu procesora. Najprostszy to ***round-robin sheduling***, stosowany gdy wszystkie zadania mają jednakowy priorytet i nie ma wymagań krytycznych. Czas procesora jest podzielony na równe odcinki, które są przydzielane kolejno (karuzela) dla poszczególnych zadań.

Earliest Deadline First EDF to często stosowany, optymalny algorytm szeregowania zadań. Za każdym razem, kiedy w systemie pojawi się niezajęty proces, z kolejki zadań oczekujących pobierane jest to o najkrótszym, wymaganym czasie na obsłużenie (*deadline*). Istnieje twierdzenie, zgodnie z którym, jeśli EDF nie może uszeregować danego zbioru zadań to żaden algorytm z dynamicznym przydziałem priorytetów nie znajdzie wykonalnego szeregowania. Druga część tego twierdzenia stanowi, że jeżeli dany algorytm z dynamicznym przydziałem priorytetów potrafi uszeregować dany zbiór zadań to również EDF potrafi. Zbiór zadań jest szeregowalny za pomocą algorytmu EDF wtedy i tylko wtedy, gdy istnieje nadmiar wydajności procesora w stosunku do potrzeb.

W tym rozdziale napiszemy własny, bardzo prosty RTOS. Będzie on przydzielał równe odcinki czasu pracy procesora, kolejno do istniejących zadań (tasków). Pierwszy task (`Main`), to pętla główna, w której będą wyświetlane informacje o realizacji wszystkich tasków, oraz mogą być wykonywane inne procesy dotyczące całego systemu, jak na przykład transmisja szeregowa. Dwa pozostałe taski, które nazwiemy `Task1` i `Task2`, mają obsługiwać dwa niezależne procesy zewnętrzne. W trakcie inicjalizacji, do obu `Task1` i `Task2` będą przekazane wskaźniki na tablice z danymi inicjującymi.



Rys. 22.1: Kolejka zadań oczekujących na obsłużenie

Elementy architektury procesora wspierające RTOS:

- Podwójny wskaźnik stosu: dostępne są dwa wskaźniki stosu. MSP jest używany przez system operacyjny i programy obsługi przerwań, PSP przez taski. Tryby i stany pracy procesora były opisane w rozdziale 11.2.
- Licznik SysTick: prosty licznik do generacji przerwań systemowych. Program obsługi przerwania od SysTick zarządza taskami.
- Przerwania SVC (*Supervisor Call*) i PendSV (*Pendable Service Call*): dwa przerwania wywoływane programowo (instrukcją SVC i wpisem do rejestru ICSR). Umożliwiają nadzorowany dostęp do uprzywilejowanych zasobów taskom pracującym na poziomie braku uprzywilejowania.
- Poziom braku uprzywilejowania (*unprivileged level*): poprawia stabilność systemu poprzez ograniczenie dostępu tasków do krytycznych zasobów systemu. Mechanizm ten może być połaczony z *Memory Protection Unit* MPU w celu dalszego wzmocnienia odporności systemu.

22.1. Podwójny wskaźnik stosu: MSP i PSP

W rozdziale 3.3 (rysunek 3.9) wspomnieliśmy, że istnieją dwa wskaźniki stosu w miejscu rejestru `r13 (SP)`.

- *Main Stack Pointer* MSP jest wskaźnikiem domyślnym, inicjowanym po RESET. Jest używany w trybie *Thread mode*, gdy bit `b1` rejestru `CONTROL` (`SPSEL`) jest zerowy oraz zawsze w trybie obsługi przerwania (*Handler mode*).
- *Processor Stack Pointer* PSP jest używany w trybie *Thread mode*, gdy bit `b1` rejestru `CONTROL` (`SPSEL`) jest jedynką.

Instrukcje PUSH i POP oraz większość instrukcji wykorzystujących `r13 (SP)`, używają aktualnego wskaźnika stosu, czyli określonego stanem bitu `SPSEL`. Możliwy jest ponadto bezpośredni dostęp do obu wskaźników za pomocą instrukcji `MRS` i `MSR`. W prostych aplikacjach, bez systemu RTOS, możliwe jest korzystanie wyłącznie z MSP, ignorując PSP. Jednak jeżeli stosujemy RTOS, mechanizm dwóch stosów umożliwia podjęcie działań obronnych, gdy jakiś task doprowadzi do awarii systemu, gdyż wyjątki systemowe używają stosu MSP. W naszym projekcie wykorzystamy ten mechanizm do wyświetlenia niebieskiego ekranu z komunikatem o bierzącym numerze tasku i adresem instrukcji, w obsłudze wyjątku *Hard Fault*.

Mechanizm obsługi przerwania został opisany w rozdziale 11.3, przypomnijmy: bezpośrednio przed rozpoczęciem wykonywania programu obsługi przerwania procesor automatycznie chroni na stosie środowisko przerywanego programu. Jest to *Interrupt Stacking*. Bezpośrednio przed powrotem, to środowisko jest odtwarzane ze stosu: *Interrupt Unstacking*.

- **Interrupt Stacking**. Po przyjęciu przerwania, a przed wykonaniem pierwszej instrukcji programu obsługi, procesor wysyła na stos osiem rejestrów: cztery najmłodsze (`r0, r1, r2, r3`) oraz cztery bezpośrednio związane ze stanem procesora (`r12, LR, PSR, PC`), rysunek 11.4 = 22.2. Używa do tego stosu bieżącego, czyli MSP lub PSP w zależności od tego który jest używany przez przerwany program. Do `LR` zapisuje informację o bieżącym trybie pracy i przełącza się w *Handler Mode*, poziom uprzywilejowany i wskaźnik stosu MSP.

- **Interrupt Unstacking**. Po zakończeniu obsługi przerwania, automatycznie zdejmowane ze stosu są rejestry, które były tam umieszczone w fazie *stacking*.

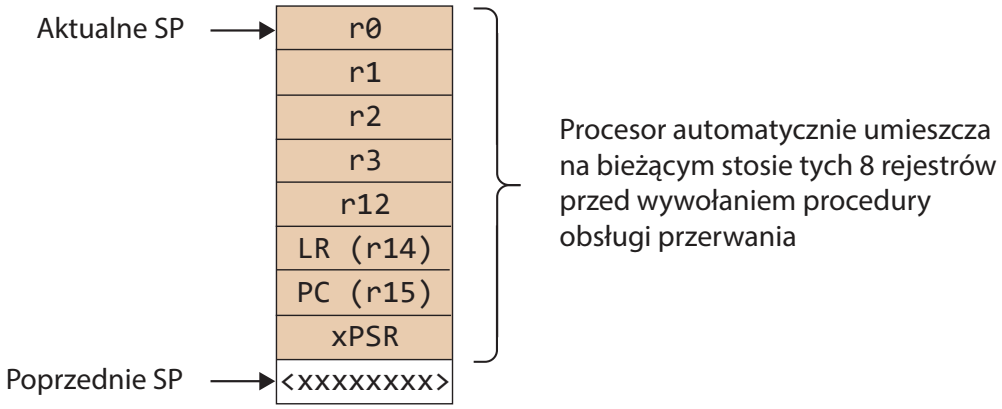
Program obsługi przerwania wraca przez wykonanie instrukcji `BL lr`, jednak znaczenie rejestru `LR` w tym przypadku jest inne niż przy powrocie z normalnego podprogramu.

- `LR` w przypadku normalnego podprogramu zawiera adres powrotu, czyli adres instrukcji bezpośrednio za instrukcją wywołania. Zawartość `LR` jest kopiowana do `PC` i sterowanie wraca do programu wywołującego.
- W przypadku obsługi przerwania adres powrotu jest umieszczony na stosie, natomiast `LR` zawiera dodatkowe informacje o trybie pracy procesora w momencie przyjęcia przerwania.
  `LR = 0xFFFFFFF1` powrót do obsługi przerwania (*Handler Mode*) o niższym priorytecie, środowisko jest odtwarzane z wykorzystaniem wskaźnika MSP
  `LR = 0xFFFFFFF9` powrót do normalnego programu (*Thread Mode*), środowisko jest odtwarzane z wykorzystaniem wskaźnika MSP (*Main Stack*)
  `LR = 0xFFFFFFFD` powrót do normalnego programu (*Thread Mode*), środowisko jest odtwarzane z wykorzystaniem wskaźnika PSP (*Process Stack*)

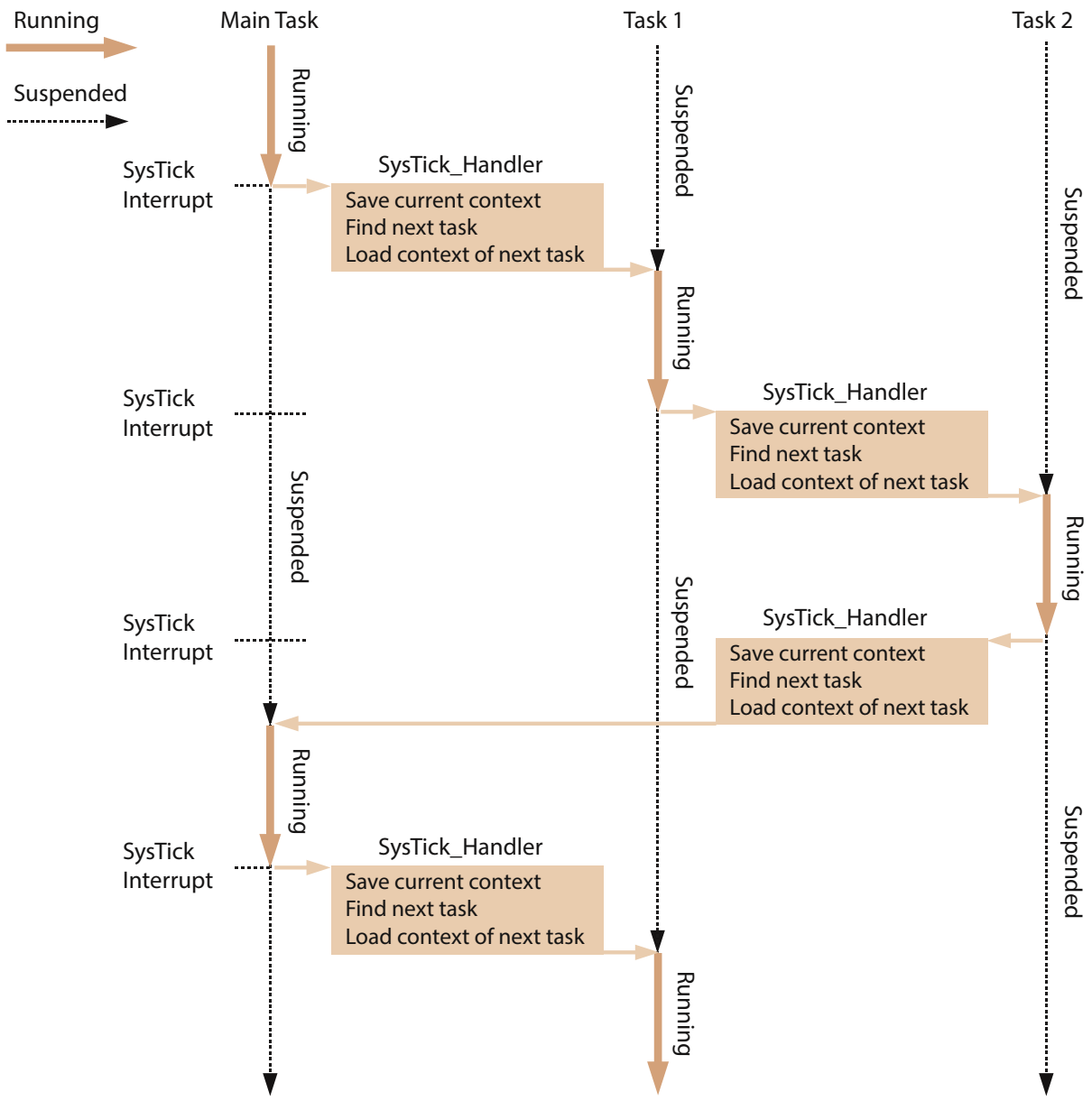
22.2. Projekt32_Multitasking

W projekcie wykorzystamy te zasoby procesora, które są przeznaczone do wspierania systemów RTOS. Cały system będzie synchronizowany przez przerwania systemowe SysTick. W obsłudze przerwania SysTick, system będzie zapamiętywał stan aktualnie wykonywanego zadania, wybierał kolejne zadanie i odtwarzał stan tego kolejnego z chwili, gdy było ono ostatnio zawieszone. Ten stan, czyli wartości wszystkich rejestrów i wszystkich flag, nazywamy **kontekst**em (*context*) danego zadania. Przełączanie kontekstów zadań wykonywane będzie w obsłudze przerwania systemowego SysTick, jak pokazano na rysunku 22.3. W każdej chwili tylko jeden task ma przydzielony czas procesora (*running*), pozostałe są zawieszone (*suspended*).

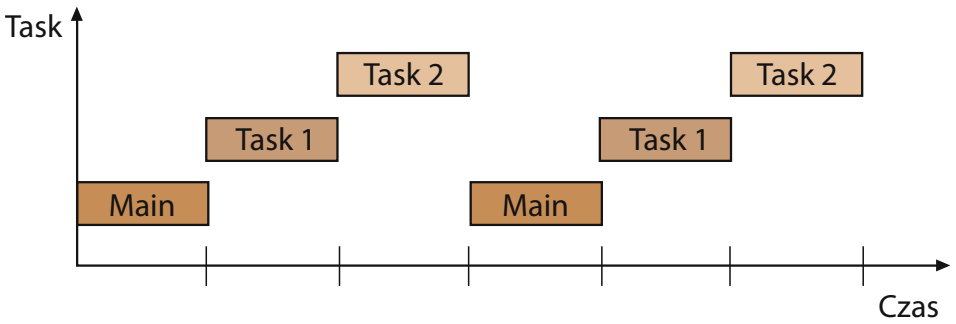
Z rysunku 22.3 wynika, że system RTOS potrzebuje do działania także pewnego czasu procesora - głównie na okresowe wykonywanie programu `SysTick_Handler`. To oznacza, że zbyt częste przełączanie zadań niekorzystnie zmieni proporcję czasu procesora zużywanego na potrzeby RTOS w stosunku do czasu, jaki może być przeznaczony na wykonywanie



Rys. 22.2: Rozmieszczenie rejestrów chronionych na stosie w operacji *Stacking*



Rys. 22.3: Obsługa przerwania systemowego SysTick wykonuje przełączanie kontekstów



Rys. 22.4: Szeregowanie zadań i przydział czasu w Projekt31_Multitasking

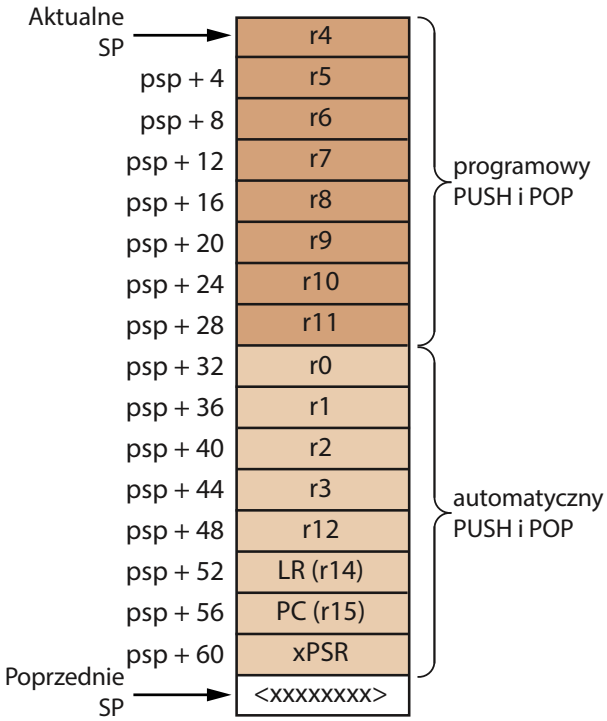
zadań użytecznych. Z drugiej strony nadmiernie wydłużenie okresu pomiędzy przerwaniami systemowymi spowoduje, że zadania w kolejce będą zbyt długo czekały na przydział procesora, obsługa zadań będzie "szarpana". Należy więc znaleźć kompromis. Typowe czasy to od 10 do 100 ms. My zaprogramujemy SysTick na 50 ms.

W naszym przykładzie nie będzie współdzielonych zasobów, czyli każdy task będzie miał swoje zadania i swoje zasoby:

- **Main** – inkrementuje licznik `counter_main`, miga zielonym LEDem i wyświetla wartości wszystkich zmiennych globalnych modyfikowanych w czasie pracy systemu: trzy liczniki i napięcie potencjometru zmierzone przez `Task1`. Zasoby: wyświetlacz, zmienna globalna `counter_main` i zielony LED
- **Task1** – zwiększa licznik `counter_task1` o zainicjowaną wartość, miga czerwonym LEDem i mierzy napięcie potencjometru, które umieszcza w zmiennej globalnej `voltage`. Zasoby: przetwornik ADC1, zmienne globalne `counter_task1` i `voltage` oraz czerwony LED
- **Task2** – zwiększa licznik `counter_task2` o zainicjowaną wartość i miga niebieskim LEDem. Zasoby: zmienna globalna `counter_task2` i niebieski LED

Ilustracją działania naszego RTOS jest rysunek 22.4. W tablicach inicjujących oba taski będzie tylko jeden parametr: liczba, o jaką należy zwiększać licznik. Dla `Task1` będzie to dwa, dla `Task2` – trzy.

W większości systemów RTOS, zadania wykonują pętlę nieskończoną, na przykład sterują niezależnymi procesami zewnętrznymi, czyli nigdy nie rezygnują z należnego im czasu procesora. To znacznie upraszcza organizację systemu. Przyjmując wszystkie powyższe założenia, uzyskaliśmy najprostszy z możliwych (ale w pełni funkcjonalny) system RTOS. Utworzymy *Projekt31_Multitasking* przez skopiowanie projektu *Projekt21_ADC1* i zmianę nazwy. Do katalogu *Src* dodamy trzy nowe pliki: *AsmCode.s*, *Task1.c* i *Task2.c*,



Rys. 22.5: Struktura kontekstu na stosie

a do katalogu *Inc* odpowiadające im pliki nagłówkowe. W chwili przejścia bieżącego tasku do obsługi przerwania systemowego, procesor automatycznie umieszcza na bieżącym stosie (MSP albo PSP) osiem rejestrów zgodnie z rysunkiem 22.2. Na początku obsługi przerwania SysTick, wypchniemy na ten sam stos pozostałe rejestry, czyli `r4 - r11`. Łącznie struktura danych zapamiętanych na stosie (kontekst tasku) będzie miała postać jak na rysunku 22.5. W celu wygodnego dostępu do tych danych, zdefiniujemy strukturę kontekstu. W pliku nagłówkowym *AsmCode.h* zdefiniujemy trzy stałe i dwie struktury

```
#define MAX_TASKS 3
#define TASK_FLAG_INIT 0x01 //jedynka oznacza zainicjowanie kontekstu
#define TASK_FLAG_EXEC 0x04 //jedynka oznacza brak wcześniejszej egzekucji
// struktury ----------------------------------------------------
typedef struct {    // rozmieszczenie rejestrów na stosie
   uint32_t r4;  // pushed by software
   uint32_t r5;
   uint32_t r6;
   uint32_t r7;
   uint32_t r8;
   uint32_t r9;
   uint32_t r10;
   uint32_t r11;
   uint32_t r0;    // pushed by hardware
   uint32_t r1;
   uint32_t r2;
   uint32_t r3;
   uint32_t r12;
   uint32_t lr;
   uint32_t pc;
   uint32_t psr;
}stack_frame_t;

typedef struct{    // structure of task_table
   uint32_t sp;    // Task stack pointer
   int flags;      // Task status flags
} task_table_t;
```

W pliku *main.c* zaczniemy od zdefiniowania zmiennych globalnych

```
// zmienne globalne ---------------------------------------------
unsigned char txt2LCD[20] = {0};    // bufor funkcji sprintf
unsigned int counter_main = 0;
unsigned int counter_task1 = 0;
unsigned int counter_task2 = 0;
unsigned int current_task = 0;      // numer bieżącego tasku
float voltage;                      // napięcie potencjometru (Task1)

task_table_t task_table[3];  // tablica tasków
                             // task_table[0]: stack of the main task
                             // task_table[1]: stack of task1
                             // task_table[2]: stack of task2
```

```
// tablice parametrów inicjujących taski ----------------------
const uint32_t var1 = 2, var2 = 3;
```

W pamięci mamy więc dwie tablice z parametrami inicjującymi taski. W naszym przykładzie obie są jednoelementowe: var1 i var2. Ponadto jest tablica trzech tasków. Każdy element tej tablicy pamięta wskaźnik stosu i dwie flagi swojego tasku. Ostatnia zmienna systemowa to current_task, czyli numer bieżącego tasku. Na końcu pliku *main.c* dopiszemy dwie funkcje systemowe

```
// Funkcje systemu RTOS ---------------------------------------
void new_task(void (*func)(void*), uint32_t args){
   volatile unsigned int i;
   stack_frame_t * frame;
   for(i=1; i < MAX_TASKS; i++){
     if(task_table[i].flags == 0){
       frame = (stack_frame_t *)(task_table[i].sp - sizeof(stack_frame_t));
       frame -> r4 = 0;
       frame -> r5 = 0;
       frame -> r6 = 0;
       frame -> r7 = 0;
       frame -> r8 = 0;
       frame -> r9 = 0;
       frame -> r10 = 0;
       frame -> r11 = 0;
       frame -> r0 = (uint32_t)args;
       frame -> r1 = 0;
       frame -> r2 = 0;
       frame -> r3 = 0;
       frame -> r12 = 0;
       frame -> pc = ((uint32_t)func);
       frame -> lr = 0;
       frame -> psr = 0x21000000;      // set default PSR value
       task_table[i].flags = TASK_FLAG_EXEC | TASK_FLAG_INIT;
       task_table[i].sp = (uint32_t) frame;
       set_PSP(task_table[i].sp);
       break;
     }
   }
}
// update the stack table: save the current task's stack pointer
 void update_sp(void){
   if(current_task == 0){
     task_table[current_task].sp = get_MSP();
   }
   else if ((task_table[current_task].flags & TASK_FLAG_EXEC) == 0){
     task_table[current_task].sp = get_PSP();
   }
}
```

```
// identify the next task to be executed
uint32_t get_next_task(void){
  current_task ++;
  if(current_task == MAX_TASKS){
    current_task = 0;
    set_MSP(task_table[current_task].sp);
    return 0xFFFFFFF9;      // exit interrupt by using the main stack
  }
  else if(task_table[current_task].flags & TASK_FLAG_INIT){
    set_PSP(task_table[current_task].sp);
    if(task_table[current_task].flags & TASK_FLAG_EXEC)
      task_table[current_task].flags &= ~TASK_FLAG_EXEC;
    return 0xFFFFFFFD;      // exit interrupt by using the process stack
  }
  current_task = 0;  // próba załączenia niezainicjowanego tasku
  set_MSP(task_table[current_task].sp);
  return 0xFFFFFFF9;
}
```

W pliku *AsmCode.s* umieścimy programy asemblerowe systemu RTOS. Podstawowym programem całego RTOS jest obsługa przerwania systemowego SysTick

```
                AREA AsmCode, CODE, READONLY
;-------- obsluga przerwania SysTick --------
                IMPORT    get_next_task
                IMPORT    update_sp
                EXPORT    SysTick_Handler
                ALIGN
SysTick_Handler PROC
                CPSID I               ; set PRIMASK to disable IRQ
                TST       lr,#0x04
                MRSEQ     r0,msp
                MRSNE     r0,psp
                STMDB     r0!,{r4-r11}
                MSREQ     msp,r0
                MSRNE     psp,r0

                BL        update_sp
                BL        get_next_task
                MOV       lr,r0

                TST       lr,#0x04
                MRSEQ     r0,msp
                MRSNE     r0,psp
                LDMFD     r0!,{r4-r11}
                MSREQ     msp,r0
                MSRNE     psp,r0
                CPSIE I               ; clear PRIMASK to enable IRQ
                BX        lr
                ENDP
```

Ponadto potrzebne są cztery proste funkcje związane z obsługą MSP i PSP

```
;--------- Read main stack pointer ----------
          EXPORT  get_MSP
          ALIGN
get_MSP   PROC
          MRS     r0,msp
          BX      lr
          ENDP
;--------- Write main stack pointer ----------
          EXPORT  set_MSP
          ALIGN
set_MSP   PROC
          MSR     msp,r0
          BX      lr
          ENDP
;-------- Read process stack pointer ---------
          EXPORT  get_PSP
          ALIGN
get_PSP   PROC
          MRS     r0,psp
          BX      lr
          ENDP
;-------- Write process stack pointer ---------
          EXPORT  set_PSP
          ALIGN
set_PSP   PROC
          MSR     psp,r0
          BX      lr
          ENDP

          END
```

Napiszemy programy obu tasków i umieścimy w plikach *Task1.c* i *Task2.c*

```
// STM3210E-EVAL:PROJEKT31_MultiTasking, Task2 -------------------
#include "stm32f10x.h"
extern unsigned int counter_task2;
void my_task_2(unsigned int *data){
   volatile unsigned int i;
   unsigned int varTask;
   varTask = *data;
   while(1){
     GPIO_SetBits(GPIOF, GPIO_Pin_9);  // zapal LED niebieski
     counter_task2 += varTask;
     for(i=0; i<1600000; i++);
     GPIO_ResetBits(GPIOF, GPIO_Pin_9);
   for(i=0; i<1600000; i++);
   }
}
```

Do tasku pierwszego możemy dodać odczyt napięcia suwaka potencjometru z przetwornika ADC1. W pliku *main.c*, po inicjalizacji RCC, GPIO, NVIC, ADC1 oraz wyświetlacza, zainicjujemy RTOS

```
// inicjalizacja Multitasking
task_table[1].sp = (unsigned int) 0x20001700;
task_table[2].sp = (unsigned int) 0x20002700;
new_task(my_task_1, (uint32_t) &var1); //inicjalizuje Task1 ze wskaźnikiem
new_task(my_task_2, (uint32_t) &var2); //inicjalizuje Task2 ze wskaźnikiem
```

Pierwsze dwie instrukcje wpisują do tablicy adresy, od których umieścimy stosy obu tasków. Ponieważ MSP jest inicjowane adresem `0x20000700`, to mamy trzy stosy rozmieszczone w równych odległościach `0x1000`. Stosy powinny być rozsunięte w pamięci na tyle mocno, by nie groziło, że jeden stos nasunie się na inny. Taki przypadek zawsze oznacza totalną awarię systemu.

Pierwsze wywołanie funkcji `new_task(...);` inicjuje miejsce na kontekst tasku `Task1`, na jego stosie. Licznik pętli `for` w funkcji `new_task(...);` zaczyna się od jedynki, a nie od zera, czyli zaczynamy od pierwszego tasku, a nie od `main();`. Funkcja sprawdza flagi w tablicy tasków i jeżeli task nie był zainicjowany, to inicjowane jest miejsce (od początku stosu) na kontekst. Funkcja kończy się instrukcją `break;` co oznacza że będzie zainicjowane tylko jedno miejsce na kontekst tasku. Drugie wywołanie tej samej funkcji inicjuje kontekst drugiego tasku. Mapa pamięci po inicjalizacji systemu RTOS pokazana jest poniżej

adres	rejestr	Little Endian	Big Endian	tablica `task_table`
0x20000054		00 00 00 00	0x00000000	
0x20000058		00 00 00 00	0x00000000	
0x2000005C	PSP_1	C0 16 00 20	0x200016C0	
0x20000060	$flagi_1$	05 00 00 00	0x00000005	0101
0x20000064	PSP_2	C0 26 00 20	0x200026C0	
0x20000068	$flagi_2$	05 00 00 00	0x00000005	0101

adres	rejestr	Little Endian	Big Endian	kontext tasku `Task1`
0x200016C0	r4	00 00 00 00	0x00000000	
0x200016C4	r5	00 00 00 00	0x00000000	
0x200016C8	r6	00 00 00 00	0x00000000	
0x200016CC	r7	00 00 00 00	0x00000000	
0x200016D0	r8	00 00 00 00	0x00000000	
0x200016D4	r9	00 00 00 00	0x00000000	
0x200016D8	r10	00 00 00 00	0x00000000	
0x200016DC	r11	00 00 00 00	0x00000000	
0x200016E0	r0	E8 2F 00 08	0x08002FE8	wskaźnik na `var1`
0x200016E4	r1	00 00 00 00	0x00000000	
0x200016E8	r2	00 00 00 00	0x00000000	
0x200016EC	r3	00 00 00 00	0x00000000	
0x200016F0	r12	00 00 00 00	0x00000000	
0x200016F4	LR(r14)	00 00 00 00	0x00000000	
0x200016F8	PC(r15)	61 2A 00 08	0x08002A61	adres funkcji `my_task_1`
0x200016FC	xPSR	00 00 00 21	0x21000000	wartość startowa rejestru `xPSR`
0x20001700		XX XX XX XX		

adres	rejestr	Little Endian	Big Endian	kontext tasku Task2
0x200026C0	r4	00 00 00 00	0x00000000	
0x200026C4	r5	00 00 00 00	0x00000000	
0x200026C8	r6	00 00 00 00	0x00000000	
0x200026CC	r7	00 00 00 00	0x00000000	
0x200026D0	r8	00 00 00 00	0x00000000	
0x200026D4	r9	00 00 00 00	0x00000000	
0x200026D8	r10	00 00 00 00	0x00000000	
0x200026DC	r11	00 00 00 00	0x00000000	
0x200026E0	r0	EC 2F 00 08	0x08002FEC	wskaźnik na var2
0x200026E4	r1	00 00 00 00	0x00000000	
0x200026E8	r2	00 00 00 00	0x00000000	
0x200026EC	r3	00 00 00 00	0x00000000	
0x200026F0	r12	00 00 00 00	0x00000000	
0x200026F4	LR(r14)	00 00 00 00	0x00000000	
0x200026F8	PC(r15)	26 2B 00 08	0x08002B25	adres funkcji my_task_2
0x200026FC	xPSR	00 00 00 21	0x21000000	wartość startowa rejestru xPSR
0x20002700		XX XX XX XX		

Dopiero po zainicjowaniu RTOS możemy uruchomić przerwania systemowe SysTick

```
SysTick_Config_Div8(450000ul);     // przerwanie co 0.05 sekundy
```

Następnie program przejdzie do wykonywania pętli głównej, w której będą wyświetlane liczniki inkrementowane przez wszystkie trzy zadania oraz naprzemiennie zapalana i gaszona dioda LED (zielona).

Po czasie 50 ms, zostanie zgłoszone pierwsze przerwanie systemowe. W obsłudze SysTick_Handler (strona 344) zaczniemy od zabronienia przerwań. W rejestrze LR(r14) mamy kod wskaźnika stosu używanego przez przerwane zadanie (poprzedni task). Jest to albo 0xFFFFFFF9 gdy było MSP, albo 0xFFFFFFFD gdy PSP. Oba te kody różnią się tylko jednym bitem: 0x9 binarnie to 1001, 0xD binarnie to 1101. Różnica jest na bicie b2, dlatego maska do testowania to 0x04 (0100). W zależności od wyniku testu, do rejestru r0 ładujemy bieżącą wartość wskaźnika stosu przerwanego zadania: albo MSP, albo PSP. Kluczowa jest tu instrukcja STMDB r0!,{r4-r11}

STMDB *STore Multiple registers, Decrement address Before each access*

Wykrzyknik za oznaczeniem rejestru bazowego r0 to *writeback suffix*. W efekcie mamy instukcję, która zapisuje na stos rejestry od r4, kolejno do r11, jednocześnie zwracając w r0 adres wskazujący na ostatni zapisany rejestr. Jest to więc równoważne operacji PUSH, tylko z adresem w r0, a nie w SP. Kolejne instrukcje wpisują nową, zmodyfikowaną wartość wskaźnika stosu, odpowiednio do MSP, albo PSP (flagi nie były zmieniane od czasu TST). W tym momencie mamy na właściwym stosie zachowany cały kontekst poprzedniego (już teraz zawieszonego) tasku.

Teraz wywoływana jest funkcja C update_sp (strona 343), która zapisuje bieżący wskaźnik stosu w odpowiednie miejsce tablicy tasków task_table. Przerwany task jest

rozpoznawany po numerze pamiętanym w zmiennej systemowej `current_task`. W przypadku tasku `main` ten zapis jest wykonywany zawsze, gdyż jego egzekucja była już rozpoczęta. W przypadku pozostałych dwóch, zapis jest wykonywany tylko wtedy, dany task był wcześniej wykonywany, gdyż dopiero pierwsze wykonanie inicjuje wartość stosu PSP.

Kolejna wywoływana funkcja C to `get_next_task` (strona 344). Ta funkcja inkrementuje numer wykonywanego tasku (modulo `MAX_TASKS`). Jeżeli nowy numer to zero (task `Main`), do MSP jest ładowany wskaźnik z tablicy tasków. Funkcja wraca (przez rejestr `LR(r14)`) z wartością `0xFFFFFFF9` w rejestrze `r0`. Jeżeli nowy numer wskazuje na `Task1` lub `Task2`, to sprawdzamy, czy dany task ma zainicjowany kontekst. Jeśli nie, to nie jest następnym taskiem do uruchomienia i wracamy z taskiem zerowym. Jeśli ma zainicjowany kontekst, zerujemy jego flagę `TASK_FLAG_EXEC`, na znak, że był już wykonywany i wracamy z nowym taskiem oraz kodem `0xFFFFFFFD` w `r0`.

Kolejna instrukcja programu obsługi przerwania zapisuje właściwy kod do `LR(r14)` i następuje odtworzenie kontekstu nowego tasku. W MSP, albo PSP mamy już nową wartość. Korzystając z rejestru bazowego `r0` odtwarzamy kontekst do rejestrów `r4 - r11` za pomocą instrukcji `LDMFD r0!,{r4-r11}`

`LDMFD` *LoaD Multiple registers, popping data from Full Descending stack*

Na końcu oczywiście zezwalamy na przerwania. Instrukcja `BX lr` jest powrotem z przerwania z wykorzystaniem stosu zgodnego z kodem w `LR(r14)` i odtwarza hardware'ową część kontekstu.

Możemy przetestować program. Na wyświetlaczu powinny być zwiększane wszystkie trzy liczniki.

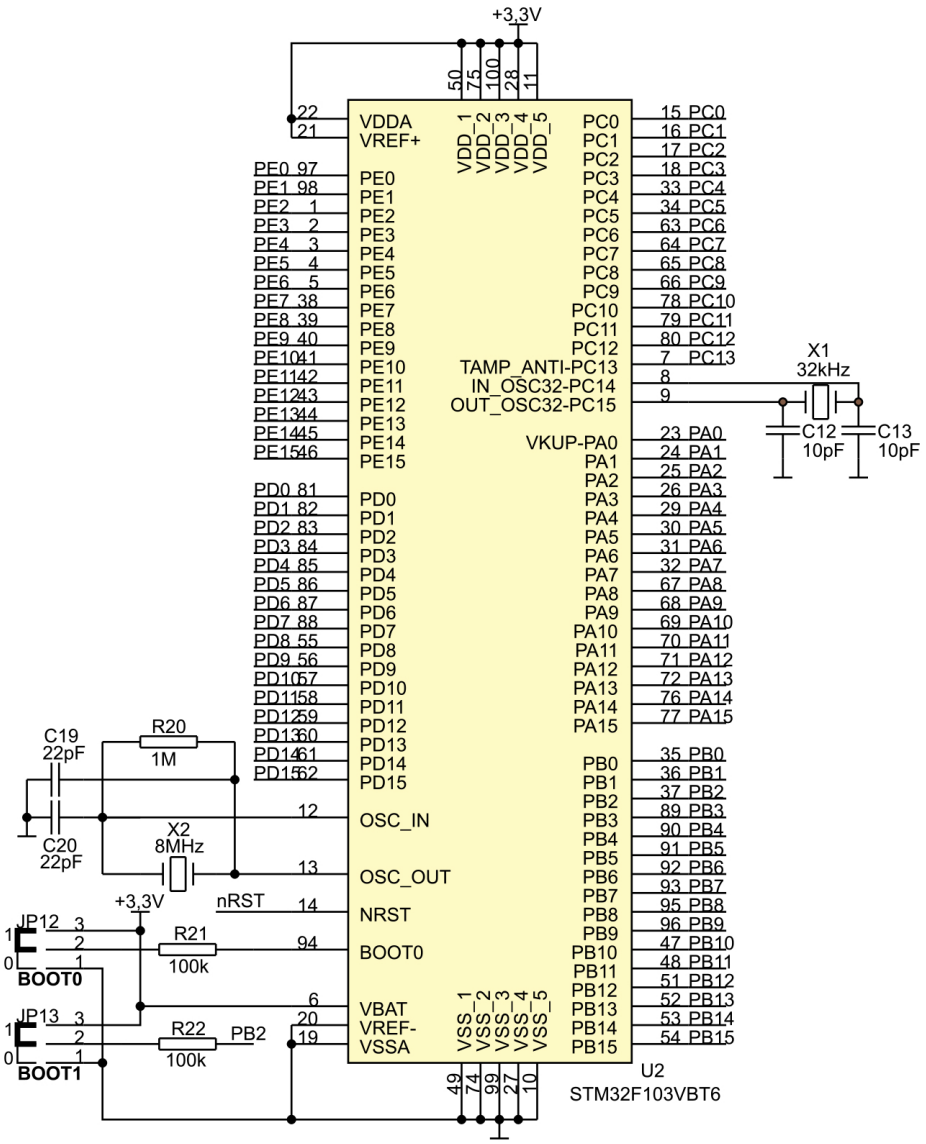
RTOS został zainicjowany za pomocą poniższych czterech linii programu

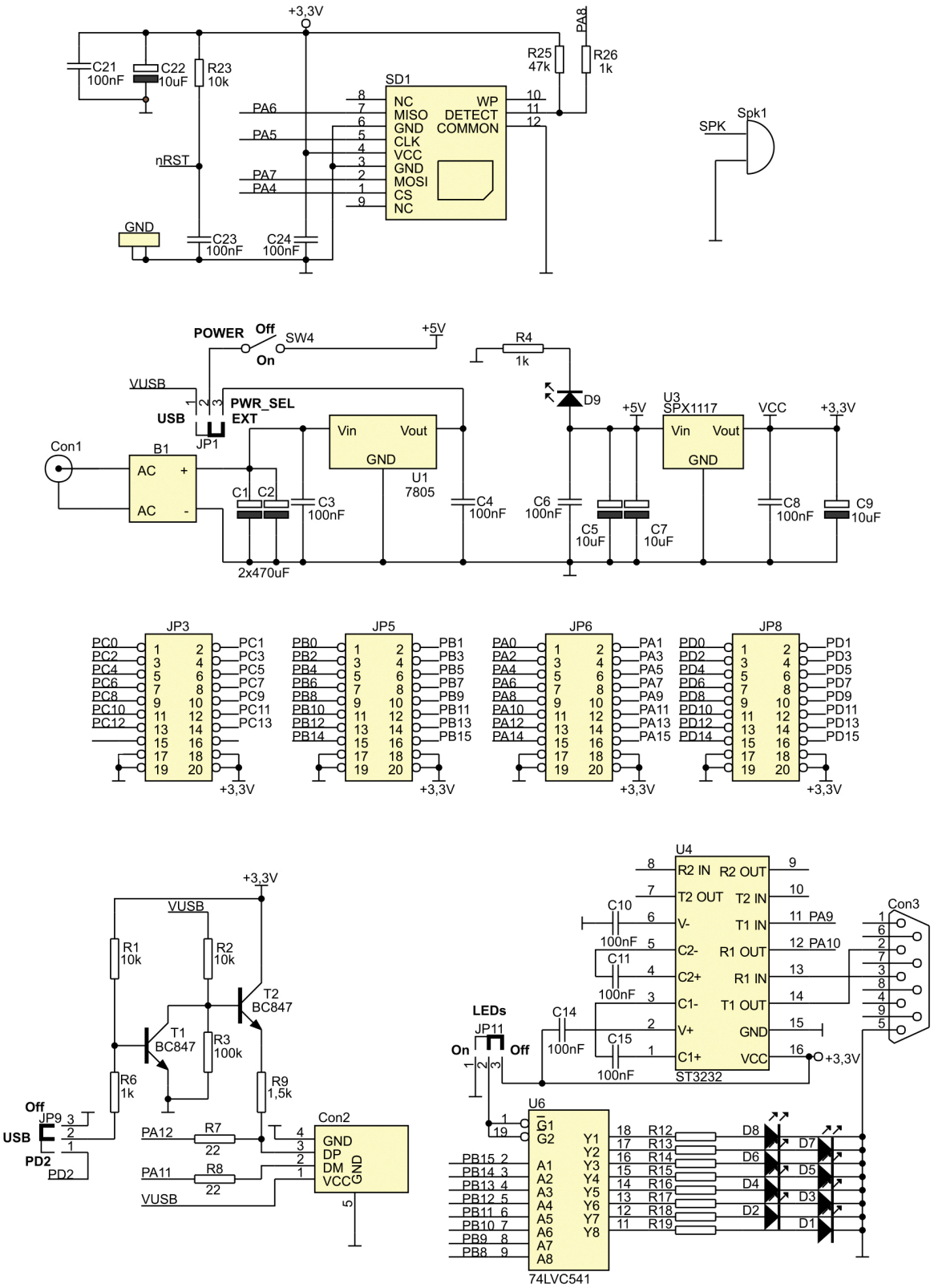
```
// inicjalizacja Multitasking
task_table[1].sp = (unsigned int) 0x20001700;
task_table[2].sp = (unsigned int) 0x20002700;
new_task(my_task_1, (uint32_t) &var1); //inicjalizuje Task1 ze wskaźnikiem
new_task(my_task_2, (uint32_t) &var2); //inicjalizuje Task2 ze wskaźnikiem
```

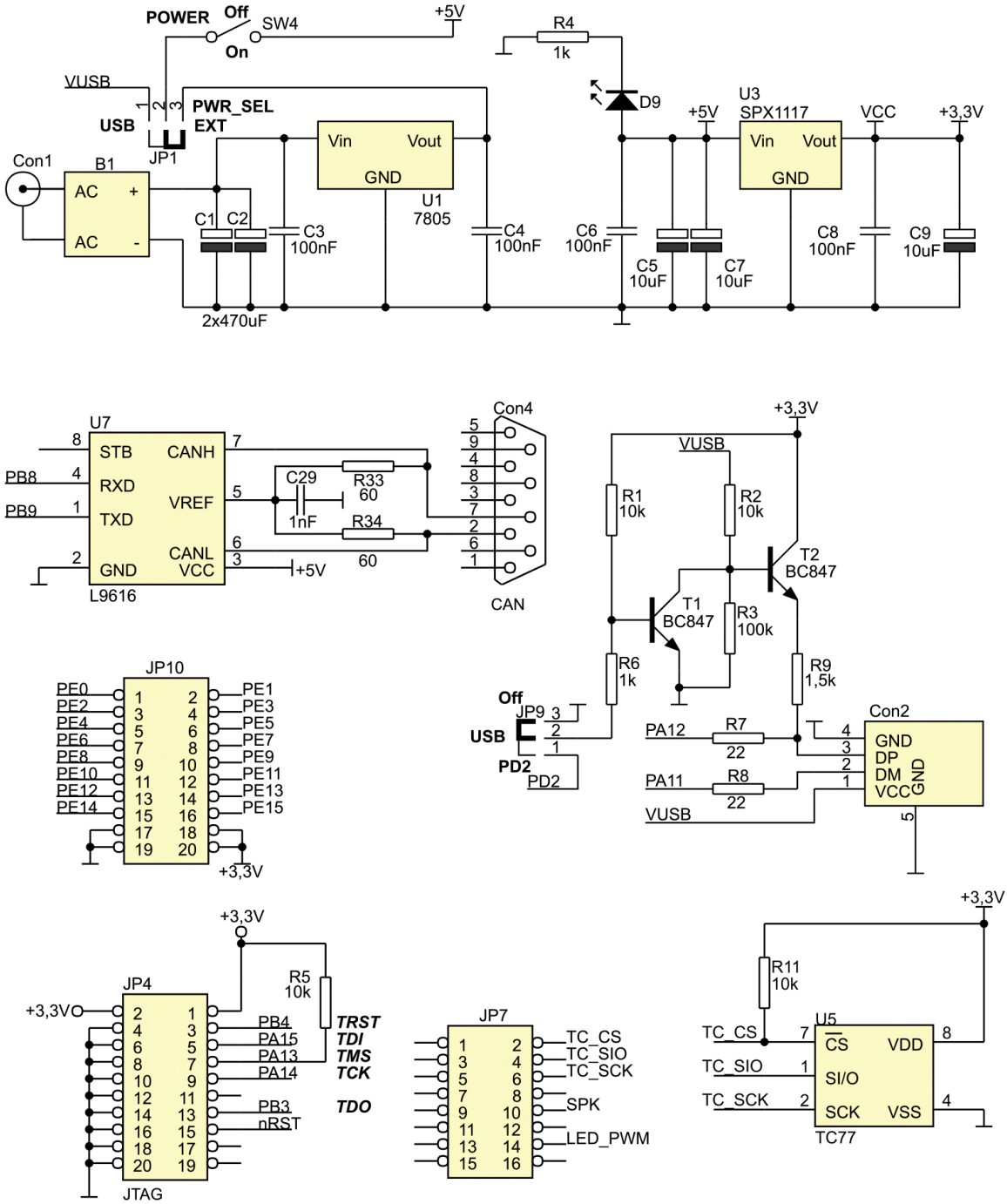
Jeżeli wyłączymy (zakomentujemy) linię trzecią, czwartą lub obie, odpowiadające im taski nie będą miały zainicjowanych kontekstów. Program powinien działać poprawnie – licznik odpowiadający niezainicjowanemu taskowi nie będzie zwiększany. Znacznie gorszy efekt uzyskamy, gdy zakomentujemy tylko linię drugą (lub pierwszą). Nastąpi próba zainicjowania kontekstu z zerową wartością wskaźnika stosu, czyli począwszy od adresu `0xFFFFFFFF`. W tym obszarze nie ma pamięci RAM. Efektem będzie zgłoszenie wyjątku `Hard Fault` i zatrzymanie systemu.

Dodatek A

Schemat zestawu uruchomieniowego ZL27ARM

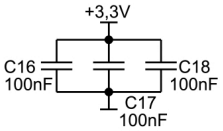


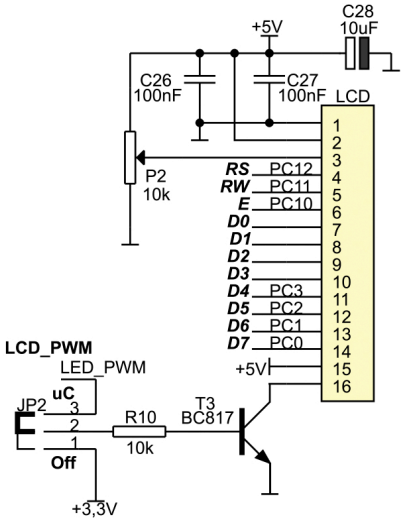

+3,3V
C21 100nF
C22 10uF
R23 10k
SD1
NC
MISO
GND
CLK
VCC
GND
MOSI
CS
NC
WP
DETECT
COMMON
R25 47k
R26 1k
PA8
PA6
PA5
PA7
PA4
nRST
GND
C23 100nF
C24 100nF
Spk1
SPK
POWER
Off
On
SW4
+5V
R4 1k
D9
VUSB
USB
PWR_SEL
EXT
JP1
Con1
B1
AC
AC
Vin
Vout
GND
U1 7805
C1
C2
C3 100nF
2x470uF
C4 100nF
C6 100nF
C5 10uF
C7 10uF
U3 SPX1117
VCC
+3,3V
C8 100nF
C9 10uF
JP3
JP5
JP6
JP8
PC0 PC1 PC2 PC3 PC4 PC5 PC6 PC7 PC8 PC9 PC10 PC11 PC12 PC13
PB0 PB1 PB2 PB3 PB4 PB5 PB6 PB7 PB8 PB9 PB10 PB11 PB12 PB13 PB14 PB15
PA0 PA1 PA2 PA3 PA4 PA5 PA6 PA7 PA8 PA9 PA10 PA11 PA12 PA13 PA14 PA15
PD0 PD1 PD2 PD3 PD4 PD5 PD6 PD7 PD8 PD9 PD10 PD11 PD12 PD13 PD14 PD15
R1 10k
R2 10k
T2 BC847
T1 BC847
R3 100k
R6 1k
R9 1,5k
Off
JP9
USB
PD2
PA12
R7 22
PA11
R8 22
Con2
GND
DP
DM
VCC
U4
R2 IN
R2 OUT
T2 OUT
T2 IN
V-
T1 IN
C2-
R1 OUT
C2+
R1 IN
C1-
T1 OUT
V+
GND
C1+
VCC
ST3232
C10 100nF
C11 100nF
C14 100nF
C15 100nF
LEDs
JP11
Con3
U6
G1
G2
A1 A2 A3 A4 A5 A6 A7 A8
Y1 Y2 Y3 Y4 Y5 Y6 Y7 Y8
R12 R13 R14 R15 R16 R17 R18 R19
D1 D2 D3 D4 D5 D6 D7 D8
74LVC541

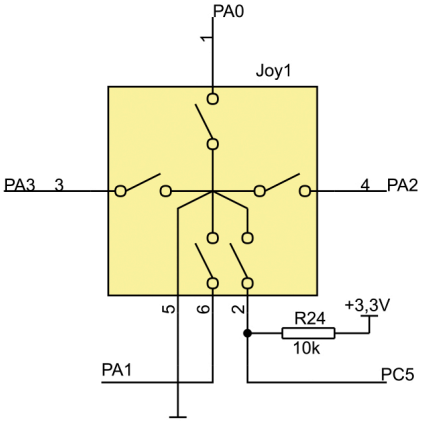

POWER
Off
SW4
On
+5V
R4
1k
D9
VUSB
USB
PWR_SEL
EXT
JP1
Con1
B1
AC
AC
Vin
Vout
GND
U1
7805
C1
C2
C3
100nF
C4
100nF
C6
100nF
C5
10uF
C7
10uF
U3
SPX1117
VCC
+3,3V
C8
100nF
C9
10uF
2x470uF
U7
STB
RXD
TXD
GND
CANH
VREF
CANL
VCC
L9616
PB8
PB9
C29
1nF
R33
60
R34
60
+5V
Con4
CAN
R1
10k
R2
10k
T2
BC847
T1
BC847
R3
100k
R6
1k
R9
1,5k
JP10
PE0
PE2
PE4
PE6
PE8
PE10
PE12
PE14
PE1
PE3
PE5
PE7
PE9
PE11
PE13
PE15
Off
JP9
USB
PD2
PA12
R7
22
PA11
R8
22
Con2
GND
DP
DM
VCC
JP4
R5
10k
PB4
PA15
PA13
PA14
PB3
nRST
TRST
TDI
TMS
TCK
TDO
JTAG
JP7
TC_CS
TC_SIO
TC_SCK
SPK
LED_PWM
R11
10k
U5
CS
SI/O
SCK
VDD
VSS
TC77

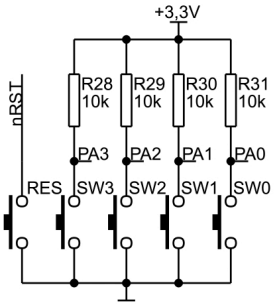
LCD 1602

Vss masa
VDD zasilanie +
VE napięcie kontrastu
RS Register Select
R/W Read/Write
E Enable
D0
D1
D2
D3
D4
D5
D6
D7
A podświetlenie +
C podświetlenie masa









Dodatek B

Znaki ASCII7 i operatory języka C

Tabela B.1: Kody ASCII

Dec	**Hex**	**Char**	**Dec**	**Hex**	**Char**	**Dec**	**Hex**	**Char**	**Dec**	**Hex**	**Char**
0	00	NUL	32	20		64	40	@	96	60	′
1	01	SOH	33	21	!	65	41	A	97	61	a
2	02	STX	34	22	”	66	42	B	98	62	b
3	03	ETX	35	23	#	67	43	C	99	63	c
4	04	EOT	36	24	$	68	44	D	100	64	d
5	05	ENQ	37	25	%	69	45	E	101	65	e
6	06	ACK	38	26	&	70	46	F	102	66	f
7	07	BEL	39	27	‘	71	47	G	103	67	g
8	08	BS	40	28	(	72	48	H	104	68	h
9	09	HT	41	29	)	73	49	I	105	69	i
10	0A	LF	42	2A	*	74	4A	J	106	6A	j
11	0B	VT	43	2B	+	75	4B	K	107	6B	k
12	0C	FF	44	2C	,	76	4C	L	108	6C	l
13	0D	CR	45	2D	-	77	4D	M	109	6D	m
14	0E	SO	46	2E	.	78	4E	N	110	6E	n
15	0F	SI	47	2F	/	79	4F	O	111	6F	o
16	10	DLE	48	30	0	80	50	P	112	70	p
17	11	DC1	49	31	1	81	51	Q	113	71	q
18	12	DC2	50	32	2	82	52	R	114	72	r
19	13	DC3	51	33	3	83	53	S	115	73	s
20	14	DC4	52	34	4	84	54	T	116	74	t
21	15	NAK	53	35	5	85	55	U	117	75	u
22	16	SYN	54	36	6	86	56	V	118	76	v
23	17	ETB	55	37	7	87	57	W	119	77	w
24	18	CAN	56	38	8	88	58	X	120	78	x
25	19	EM	57	39	9	89	59	Y	121	79	y
26	1A	SUB	58	3A	:	90	5A	Z	122	7A	z
27	1B	ESC	59	3B	;	91	5B	[	123	7B	{
28	1C	FS	60	3C	<	92	5C	\	124	7C	|
29	1D	GS	61	3D	=	93	5D	]	125	7D	}
30	1E	RS	62	3E	>	94	5E	∧	126	7E	∼
31	1F	US	63	3F	?	95	5F	_	127	7F	DEL

Upper 4 Bits / Lower 4 Bits	0000	0001	0010	0011	0100	0101	0110	0111	1000	1001	1010	1011	1100	1101	1110	1111
xxxx0000	CG RAM (1)			0	@	P	`	p				ー	タ	ミ	α	p
xxxx0001	(2)		!	1	A	Q	a	q			。	ア	チ	ム	ä	q
xxxx0010	(3)		"	2	B	R	b	r			「	イ	ツ	メ	ß	θ
xxxx0011	(4)		#	3	C	S	c	s			」	ウ	テ	モ	ε	∞
xxxx0100	(5)		$	4	D	T	d	t			、	エ	ト	ヤ	μ	Ω
xxxx0101	(6)		%	5	E	U	e	u			・	オ	ナ	ユ	σ	ü
xxxx0110	(7)		&	6	F	V	f	v			ヲ	カ	ニ	ヨ	ρ	Σ
xxxx0111	(8)		'	7	G	W	g	w			ァ	キ	ヌ	ラ	g	π
xxxx1000	(1)		(	8	H	X	h	x			ィ	ク	ネ	リ	√	x̄
xxxx1001	(2)		)	9	I	Y	i	y			ゥ	ケ	ノ	ル	¹	y
xxxx1010	(3)		*	:	J	Z	j	z			ェ	コ	ハ	レ	j	千
xxxx1011	(4)		+	;	K	[	k	{			ォ	サ	ヒ	ロ	ˣ	万
xxxx1100	(5)		,	<	L	¥	l	|			ャ	シ	フ	ワ	¢	円
xxxx1101	(6)		−	=	M	]	m	}			ュ	ス	ヘ	ン	£	÷
xxxx1110	(7)		.	>	N	^	n	→			ョ	セ	ホ	゛	ñ	
xxxx1111	(8)		/	?	O	_	o	←			ッ	ソ	マ	゜	ö	▒

Rys. B.1: Generator znaków LCD HD44870, wersja standardowa (ROM Code: A00)

Lower 4 Bits \ Upper 4 Bits	0000	0001	0010	0011	0100	0101	0110	0111	1000	1001	1010	1011	1100	1101	1110	1111
xxxx0000	CG RAM (1)															
xxxx0001	(2)															
xxxx0010	(3)															
xxxx0011	(4)															
xxxx0100	(5)															
xxxx0101	(6)															
xxxx0110	(7)															
xxxx0111	(8)															
xxxx1000	(1)															
xxxx1001	(2)															
xxxx1010	(3)															
xxxx1011	(4)															
xxxx1100	(5)															
xxxx1101	(6)															
xxxx1110	(7)															
xxxx1111	(8)															

Rys. B.2: Generator znaków LCD HD44870, wersja europejska (ROM Code: A02)

Tabela B.2: Operatory języka C

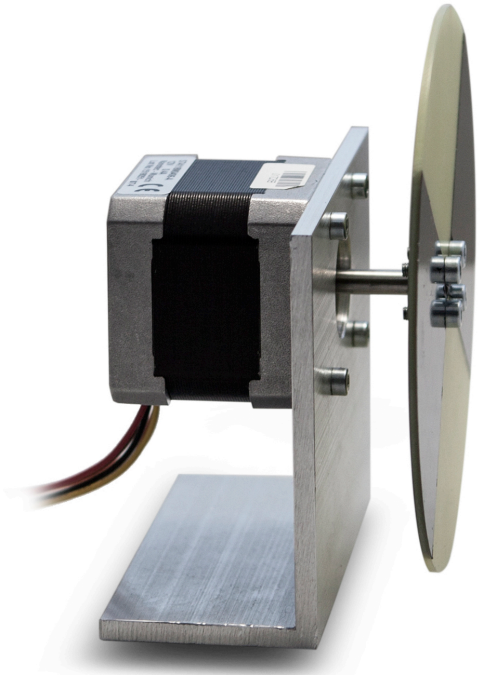
Priorytet	**Nazwa**	**Symbol**	**Łączność**
1	indeksowanie tablic	`[ ]`	lewostronna
1	wywołanie funkcji	`( )`	lewostronna
1	odwołanie do składowej struktury lub unii	`. −>`	lewostronna
1	inkrementacja przyrostkowa	`i++`	lewostronna
1	dekrementacja przyrostkowa	`i--`	lewostronna
2	inkrementacja przedrostkowa	`++i`	prawostronna
2	dekrementacja przedrostkowa	`--i`	prawostronna
2	pobranie adresu	`&`	prawostronna
2	wyłuskanie	`*`	prawostronna
2	plus (jednoargumentowy)	`+`	prawostronna
2	minus (jednoargumentowy)	`-`	prawostronna
2	negacja bitowa	`~`	prawostronna
2	negacja logiczna	`!`	prawostronna
2	określenie rozmiaru	`sizeof`	prawostronna
3	rzutowanie	`( )`	prawostronna
4	mnożenie, dzielenie, reszta z dzielenia	`* / %`	lewostronna
5	dodawanie, odejmowanie	`+ -`	lewostronna
6	przesunięcia bitowe	`<< >>`	lewostronna
7	relacje	`< <= > >=`	lewostronna
8	porównania	`== !=`	lewostronna
9	iloczyn bitowy (*and*)	`&`	lewostronna
10	exclusive OR (*xor*)	`^`	lewostronna
11	suma bitowa (*or*)	`|`	lewostronna
12	iloczyn logiczny (*and*)	`&&`	lewostronna
13	suma logiczna (*or*)	`||`	lewostronna
14	operator trójwartościowy	`?:`	prawostronna

15	przypisanie	`= += -=`
		`*= /= %=`
		`&= ∧= |=`
		`<<= >>= ==`
			prawostronna

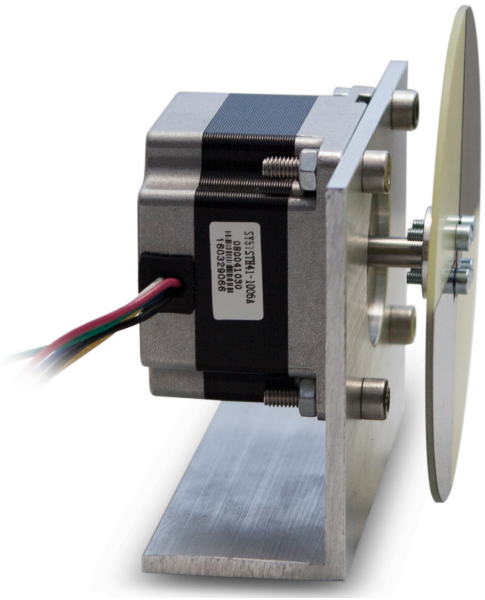
16	przecinek	`,`	lewostronna

Dodatek C

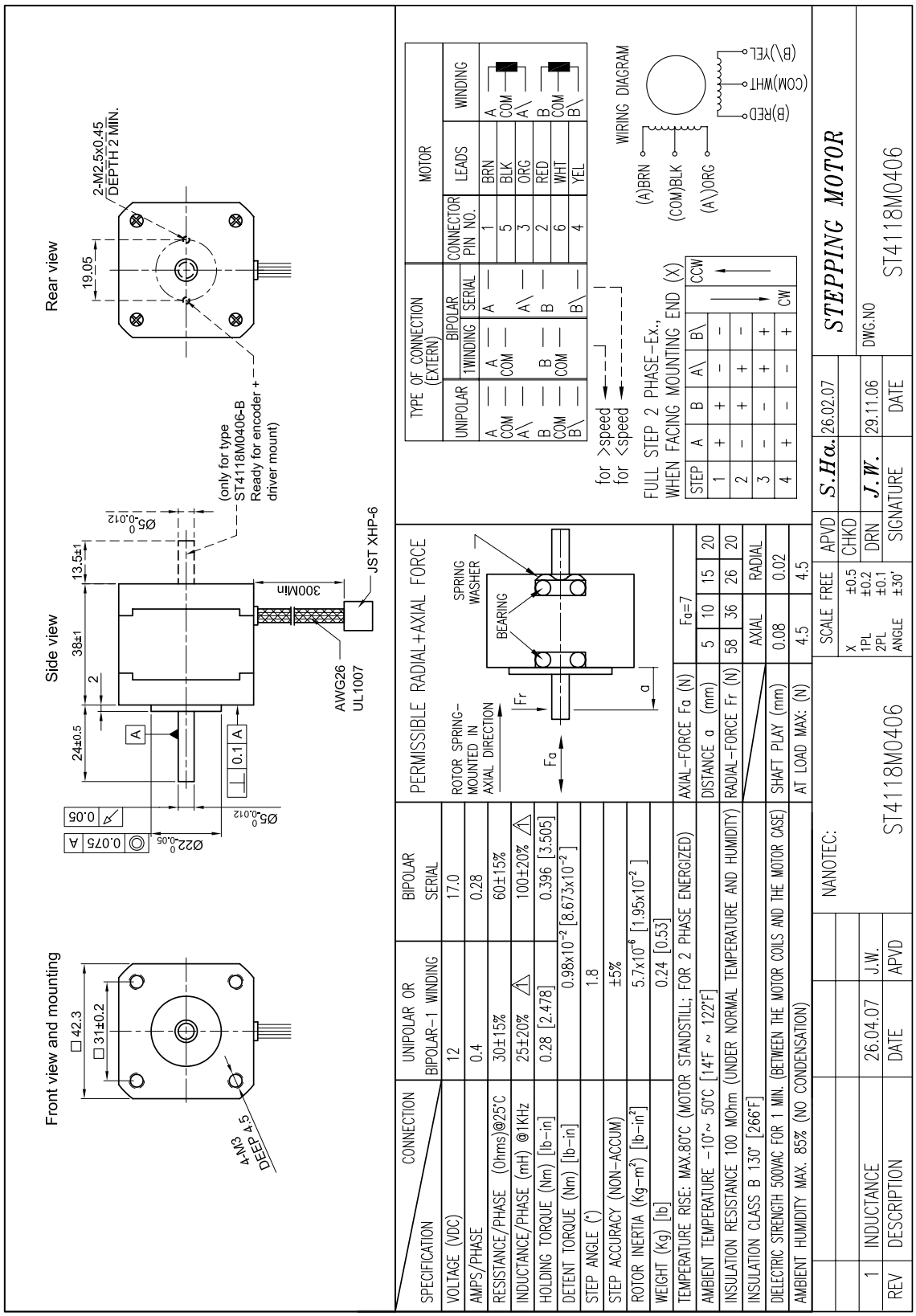
Karty katalogowe silników krokowych



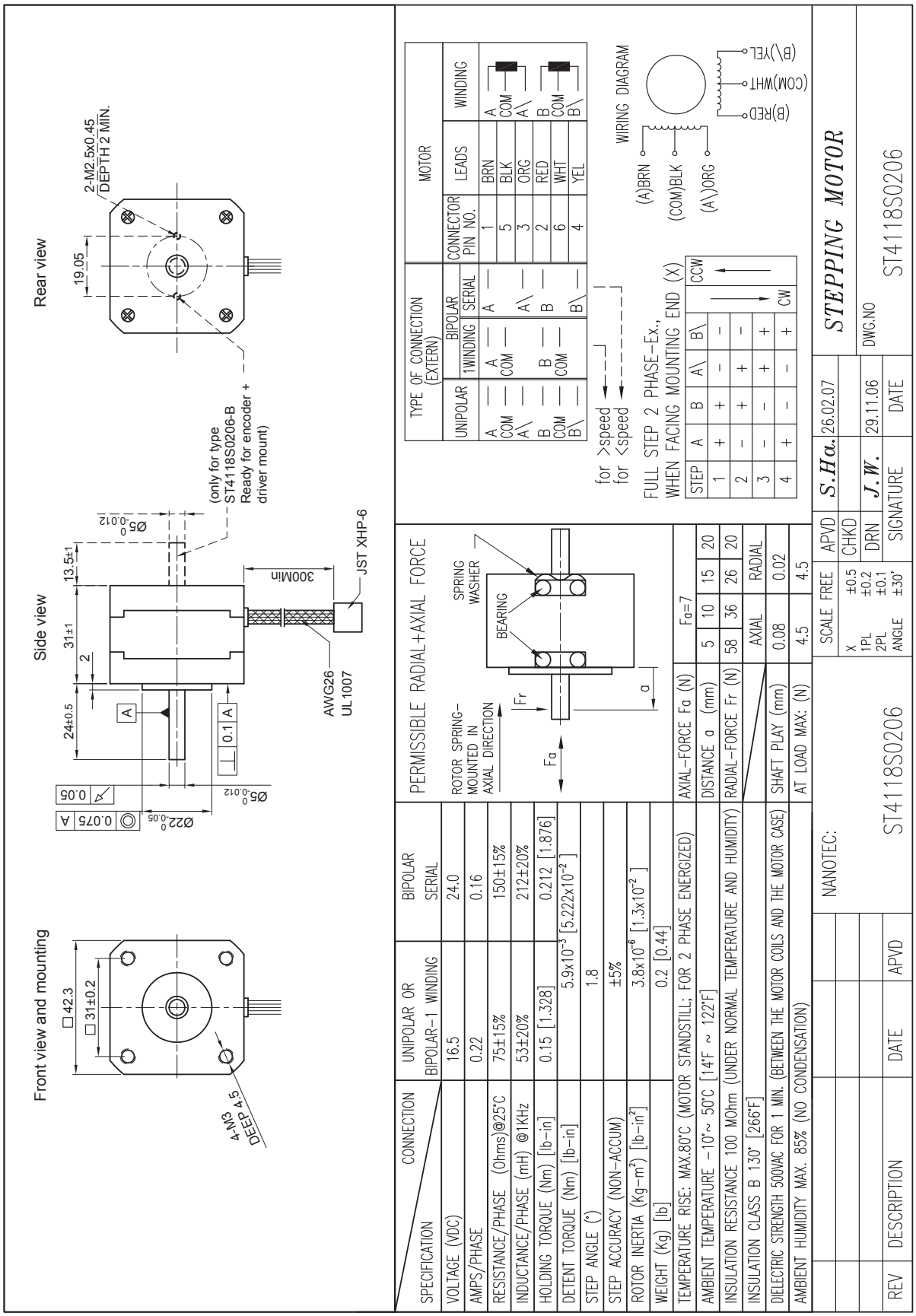
Rys. C.1: Silnik hybrydowy ST4118M0406



Rys. C.2: Silnik hybrydowy SY57STH41-1006A



Rys. C.3: Karta katalogowa silnika ST4118M0406

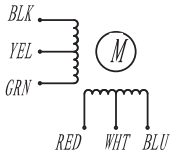


Rys. C.4: Karta katalogowa silnika ST4118S0206

HIGH TORQUE HYBRID STEPPING MOTOR SPECIFICATIONS

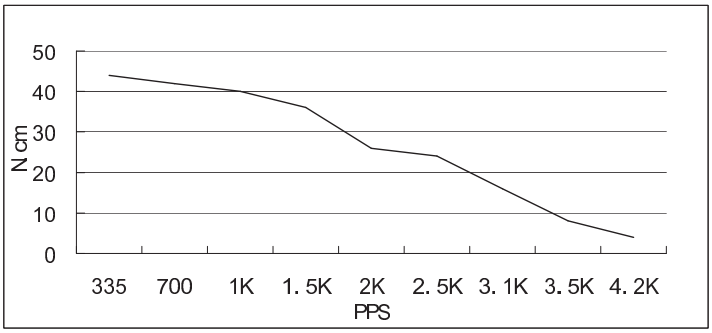
General specifications		Electrical specifications	
Step Angle (°)	1.8	Rated Voltage (V)	5.7
Temperature Rise (℃)	80 Max (rated current、2 phase on)	Rated Current (A)	1.0
Ambient Temperature (℃)	-20～+50	Resistance Per Phase (±10% Ω)	5.7 (25℃)
Number of Phase	2	Inductance Per Phase (±20% mH)	5.4
Insulation Resistance (MΩ)	100 Min (500VDC)	Holding Torque (N.cm)	39
Insulation Class	Class B	Detent Torque (Kg.cm)	0.21
Max.radial force (N)	75 (20mm from the flange)	Rotor Torque (g.cm²)	120
Max.axial force (N)	15	Weight (Kg)	0.45

● Wiring Diagram :



● Pull out torque curve :

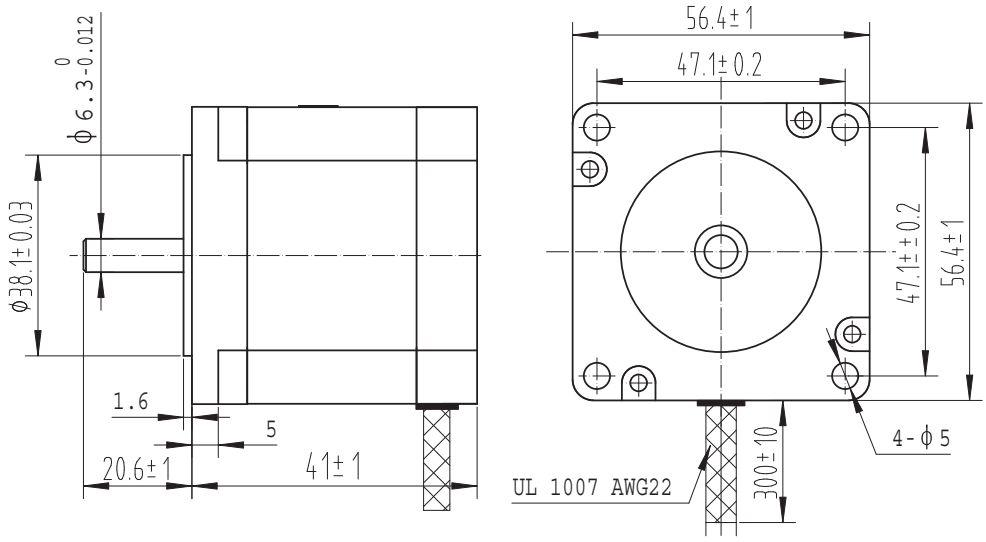
VOLTAGE: 30VDC, CONSTANT CURRENT:1.0A,HALF STEP



●

Dimensions:

(unit=mm)

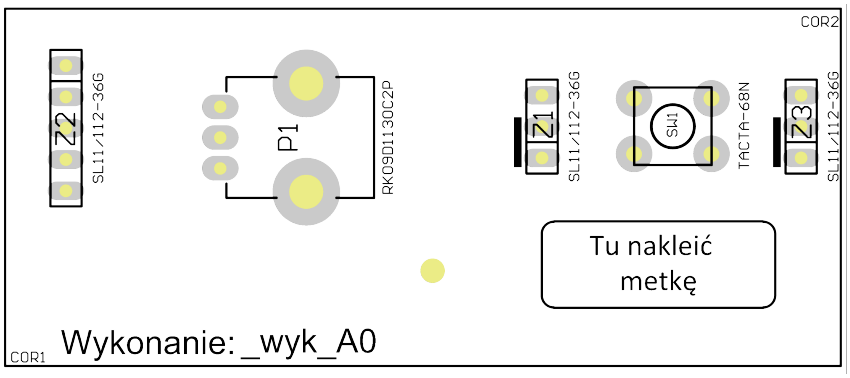


					SY57STH41-1006A	TECHNICAL CONDITIONS
REV	REVISIONS	DESCRIPTION	BY	DATE		
DRAW	孔祥振 2011.08.05				CHANGZHOU SONGYANG MACHINERY & ELECTRONICS NEW TECHNIC INSTITUTE	080041000
CHECK						
APPROVE						

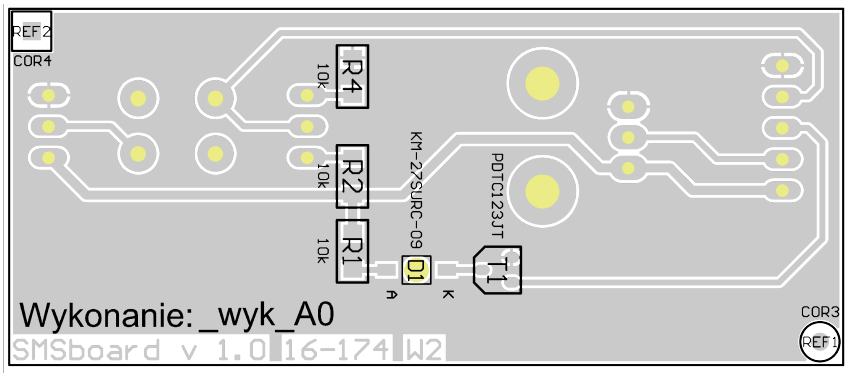
Rys. C.5: Karta katalogowa silnika SY57STH41-1006A

Dodatek D

Schemat płyty SMSboard



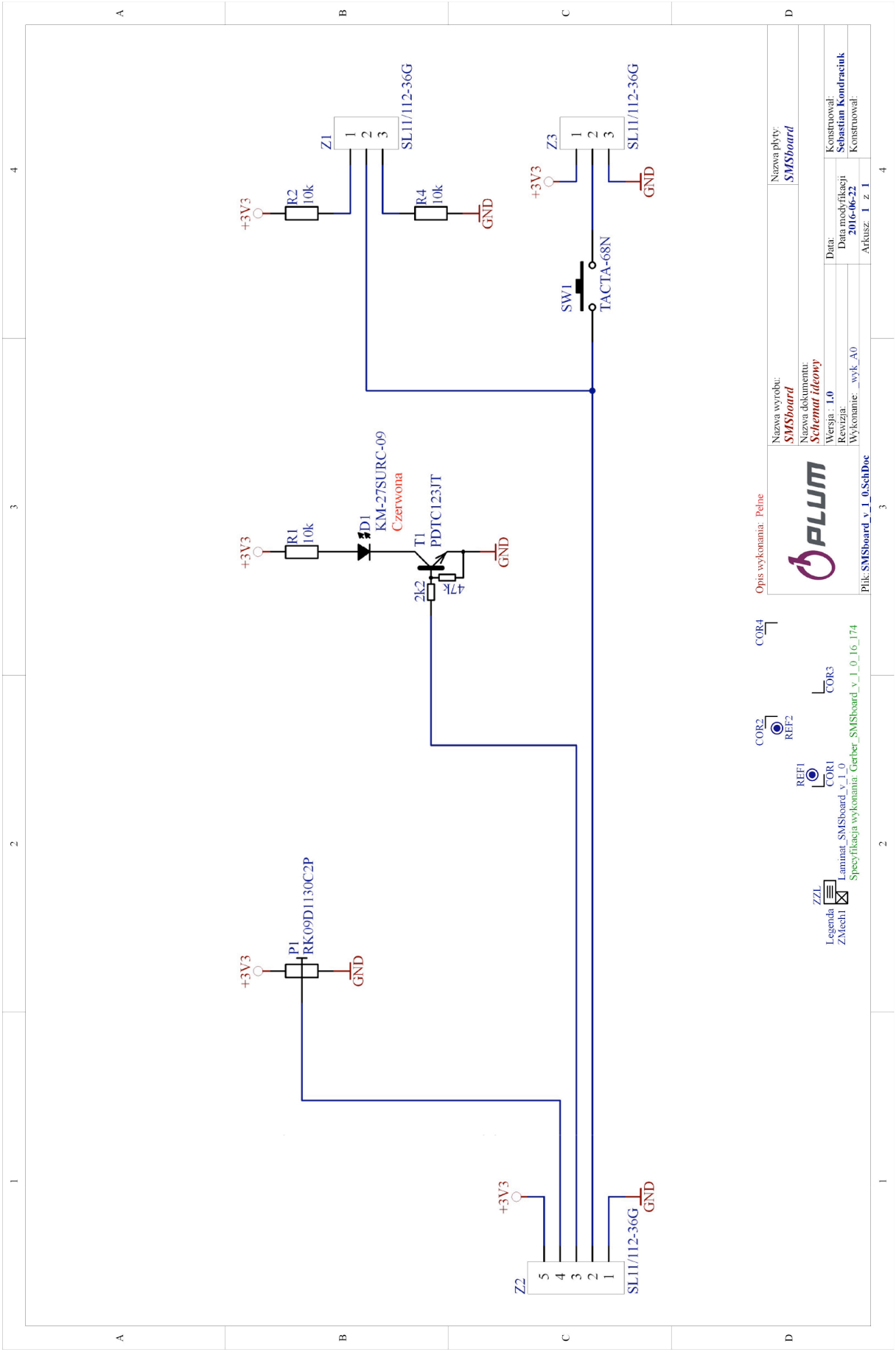
Rys. D.1: SMS_board_v1.0 warstwa 1



Rys. D.2: SMS_board_v1.0 warstwa 2

Tabela D.1: SMS_board_v1.0 - wykaz elementów

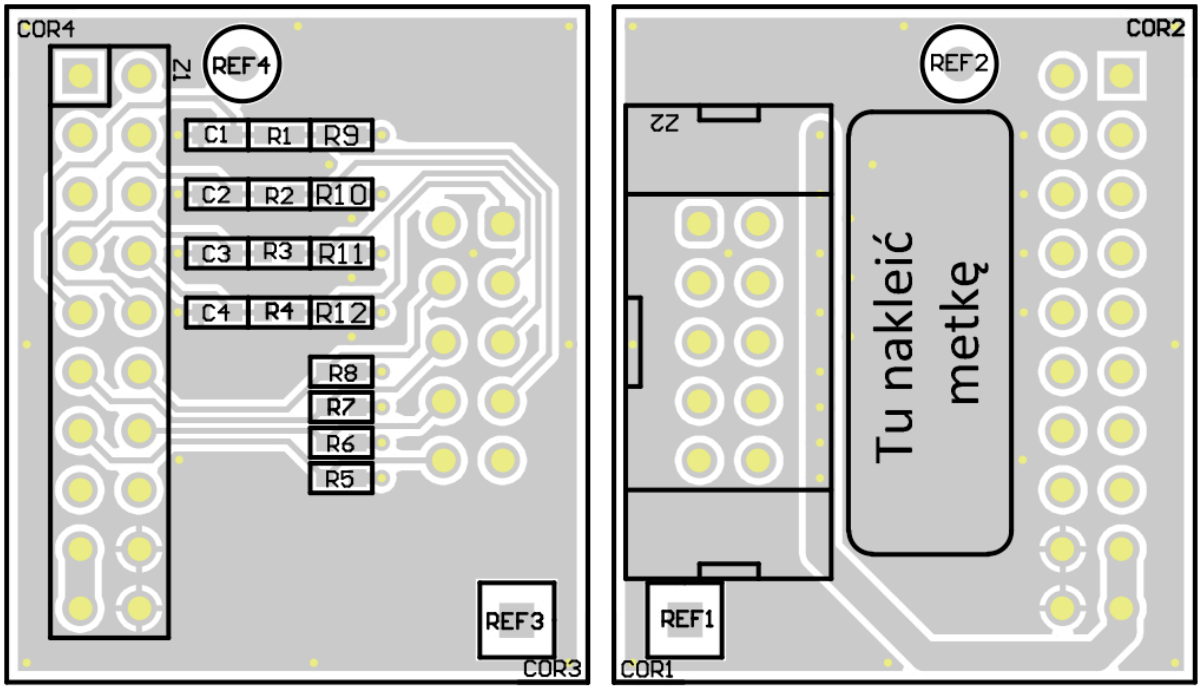
Lp.	Element	Wartość	Obudowa	Ilość
1	D1	KM-27SURC-09	KM-27SURC-09	1
2	P1	RK09D1130C2P	RK09	1
3	R1, R2, R4	10k	1206	3
4	SW1	TACTA-68N	SW-PHAP3301F	1
5	T1	PDTC123JT	SOT-23_3d	1
6	Z1, Z3	SL/112-36G	SIP3	2
7	Z2	SL/112-36G	SIP5	1
8	ZMech1	Laminat_SMSboard_v_1_0		1



Rys. D.3: SMS_board_v1.0 schemat

Dodatek E

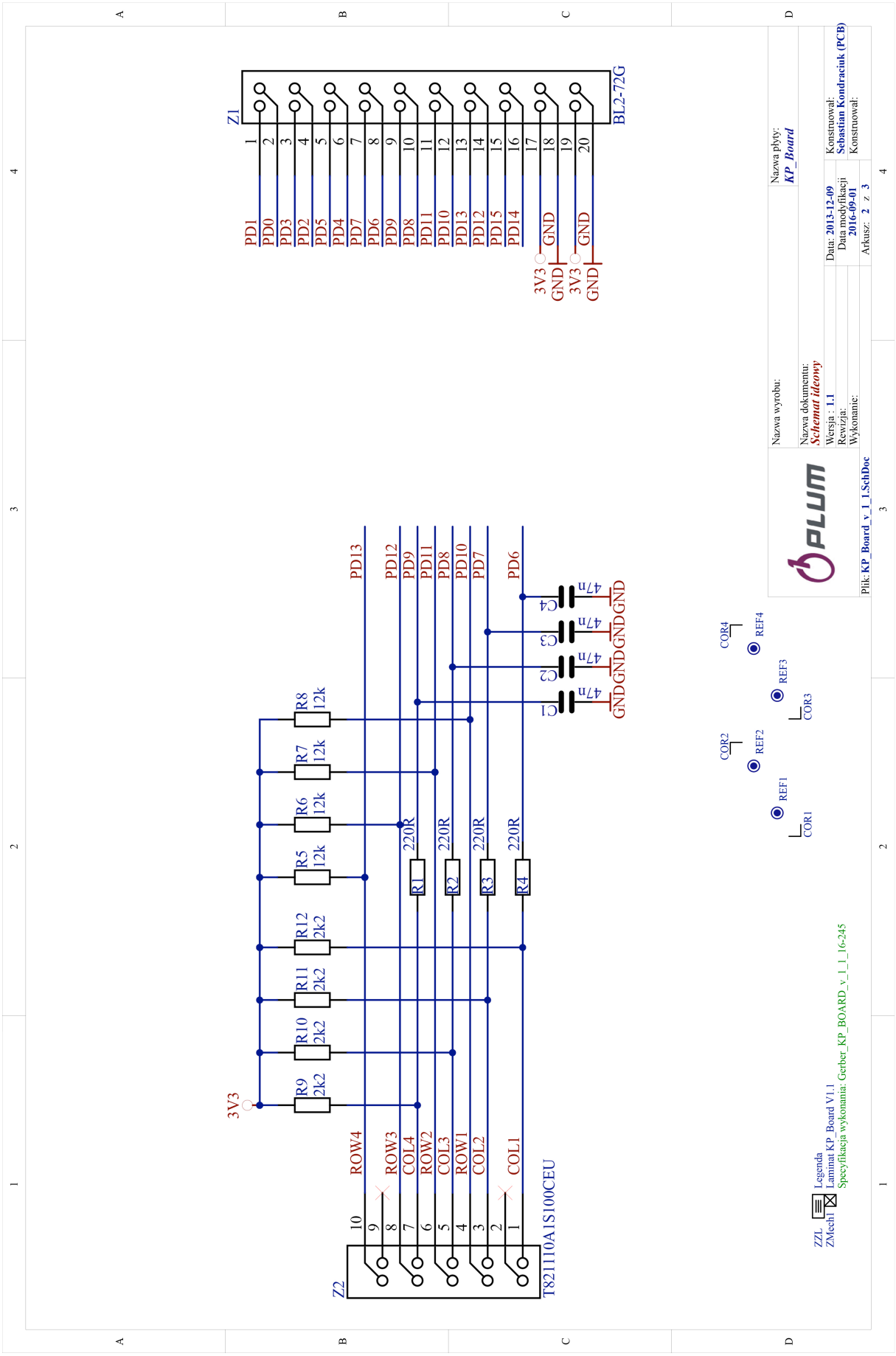
Schemat płyty KP_board



Rys. E.1: KP_board_v1.1 warstwy

Tabela E.1: KP_board_v1.1 - wykaz elementów

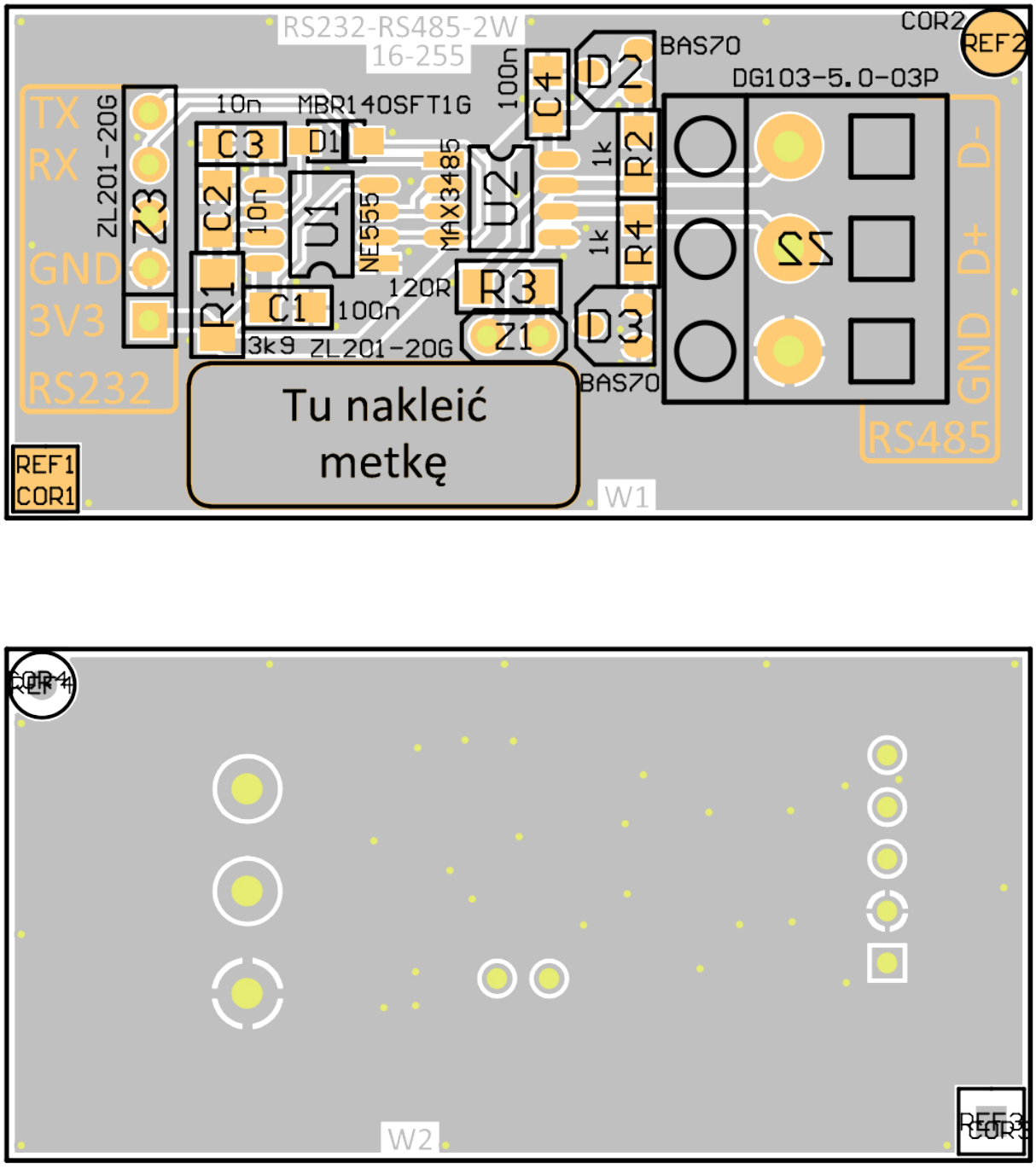
Lp.	Element	Wartość	Obudowa	Ilość
1	C1, C2, C3, C4	47n	C0603-0.6	4
2	R5, R6, R7, R8	12k	R0603	4
3	R1, R2, R3, R4	220R	R0603	4
4	R9, R10, R11, R12	2k2	R0603	4
5	Z1	BL2-72G	PIN20-S/M	1
6	Z2	T821110A1S100CEU	T821110A1S100CEU	1
7	ZMech1	Laminat KP_board V1.1	Temp	1



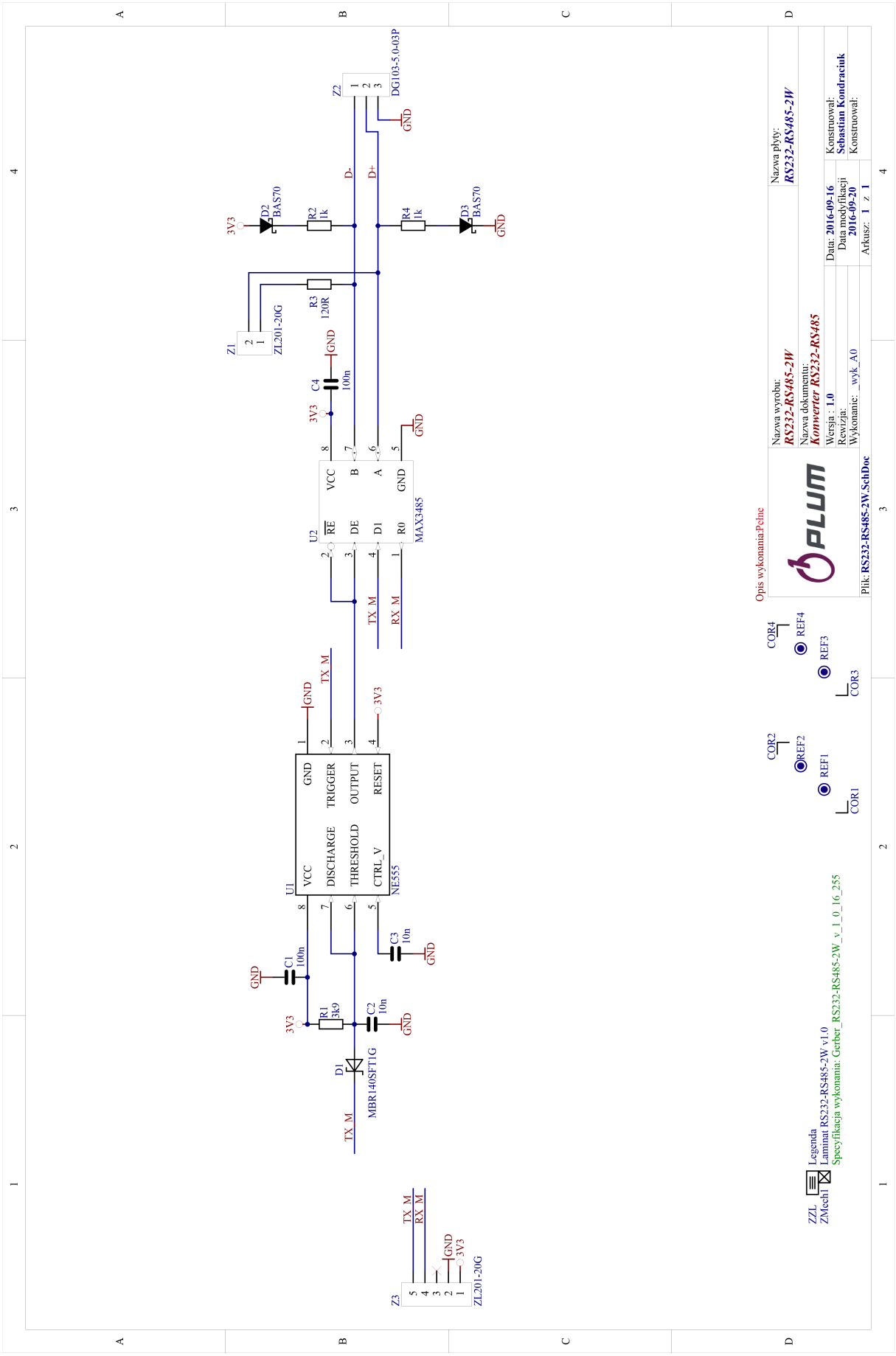
Rys. E.2: KP_board_v1.1 schemat

Dodatek F

Schemat płyty RS232/485



Rys. F.1: Konwenter RS232-RS485 v1.1 warstwy



Rys. F.2: Konwenter RS232-RS485 v1.1 schemat

Bibliografia

Elektronika

Paul Scherz, Simon Monk (2013): *Practical Electronics for Inventors* Third Edition. Mc Graw Hill Education

Mikrokontrolery

Dr. Yifeng Zhu (2015): *Embedded Systems with ARM Cortex-M. Microcontrollers in Assembly Language and C.* E-Man Press LLC 2015

Marek Galewski (2011): *STM32 Aplikacje i ćwiczenia w języku C.* Wydawnictwo btc

Joseph Yiu ARM Ltd. (2014): *The Definitive Guide to ARM Cortex-M3 and Cortex-M4 Processors.* Third Edition, Elsevier

Automatyka

Antoniou A. i Wu-Sheng Lu (2010): *Practical Optimization. Algotithms and Engineering Applications.* Springer Science+Business Media

Astrom K.J. i Wittenmark B. (1997): *Computer-Controlled Systems. Theory and Design.* Third Edition Prentice Hall, New Jersey

Astrom K.J. i Wittenmark B. (2008): *Adaptive Control.* Second Edition, Dover Publications, Inc. Mineola, New York

Bemporad A. i Morari M. (1999): *Robust model predictive control: a survey.* Robustness in identification and control, Lecture Notes in Control and Information Sciences, red. Garulli A., Tesi A., t.245, s. 207-226, Springer, Berlin

Bertsekas D.P. (1999): *Nonlinear Programming.* Second Edition, Athena Scientific, Belmont, Massachusetts

Bertsekas D.P. (2003): *Convex Analysis and Optimization.* Athena Scientific, Belmont, Massachusetts

Bertsekas D.P. (2008): *Introduction to Probability.* Second Edition, Athena Scientific, Belmont, Massachusetts

Camacho E.F. i Bordons C. (2007): *Model Predictive Control.* Second Edition, Springer, Londyn

Chaparro L.F. (2011): *Signals and Systems Using MatLab.* Elsevier

Clarke D.W. i Mohtadi C. (1989): *Properties of Generalized Predictive Control* Automatica t.25, nr 6, s. 859 - 875

Cristianini N. i Shawe-Taylor J. (2000): *An Introduction to Support Vector Machines.* Cambridge University Press

Dorf R.C. i Bishop R.H. (2008): *Modern Control Systems.* Eleventh Edition, Pearson Prentice Hall

Fadali M.S. i Visioli A. (2009): *Digital Control Engineering. Analysis and Design.* Elsevier

Fine T.L. (1999): *Feedforward Neural Network Methodology.* Springer-Verlag New York

Franklin G.F. i Powell J.D. i Workman M.L. (1998): *Digital Control of Dynamic Systems.* Addison Wesley Longman

Garcia C.E. i Morshedi A.M. (1986): *Quadratic programming solution of dynamic matrix control (QDMC)* Shell Development Company, Chemical Egineering Commun. Vol.46 pp. 073 - 087

Golub G.H. i Van Loan C.F. (1996): *Matrix Computation.* The Johns Hopkins University Press

Haykin S. (2009): *Neural Networks and Learning Machines.* Third Edition, Pearson Prentice Hall

Hestenes M. i Stiefel E. (1952): *Methods of Conjugate Gradients for Solving Linear Systems.* Journal of Research of the National Bureau of Standards, Vol.49, No.6, December 1952

Higham N.J. (2002): *Accuracy and Stability of Numerical Algorithms.* Second Edition, Society for Industrial and Applied Mathematics, Philadelphia

Isermann R. i Munchhof M. (2011): *Identification of Dynamic Systems. An Introduction with Applications.* Springer-Verlag Berlin Heidelberg

Jańczak A. (2005): *Identyfication of linear Systems Using Neural Networks and Polynomial Models.* Springer-Verlag Berlin Heidelberg

Ljung L. (1999): *System Identification. Theory for the User.* Second Edition, Prentice Hall

Ławryńczuk M. (2011): *Sieci Neuronowe w Efektywnych Obliczeniowo Algorytmach Regulacji Predykcyjnej.* Oficyna Wydawnicza Politechniki Warszawskiej

Maciejowski J.M. (1989): *Multivariable Feedback Design.* Addison-Wesley

Maciejowski J.M. (2002): *Predictive Control with Constraints.* Prentice Hall, Harlow

Marquis P. i Broustail J.P. (1988): *SMOC, a bridge between state space and model predictive controllers: application to the automation of hydrotreating unit.* Proceedings of the 1988 IFAC Workshop on Model Based Process Control, Oxford, s. 37-43

Ogata K. (2002): *Modern Control Engineering.* Pearson Prentice Hall

Piegat A. (2003): *Modelowanie i Sterowanie Rozmyte.* Wydanie II, Akademicka Oficyna Wydawnicza EXIT, Warszawa

Proakis J.G. i Manolakis D.G. (2007): *Digital Signal Processing.* Fourth Edition, Pearson Prentice Hall, New Jersey

Qin S.J. i Badgwell T.A. (2003): A survey of industrial model predictive control technology. *Control Engineering Practice*, t. 11, nr 7, s. 733–764.

Rawlings J.B. i Mayne D.Q. (2009): *Model Predictive Control: Theory and Design.* Nob Hill Publishing, Madison

Rossiter J.A. (2003): *Model-Based Predictive Control. A Practical Approach.* CRC Press, Boca Raton

Simon D. (2006): *Optimal State Estimation.* Wiley-Interscience

Soderstrom T. i Stoica P. (1989): *System Identification.* Prentice Hall International (UK)

Tatjewski P. (2007): *Advanced Control of Industrial Processes, Structures and Algorithms.* Springer, Londyn

Tatjewski P. (2002): *Sterowanie Zaawansowane Obiektów Przemysłowych.* Akademicka Oficyna Wydawnicza EXIT, Warszawa

Wang L. i Cluett W.R (2000): *From Plant Data to Process Control.* Taylor & Francis

Wang L. (2009): *Model Predictive Control System Design and Implementation Using MatLab.* Springer-Verlag London

Watkins D.S. (2010): *Fundamentals of Matrix Computation.* Third Edition, John Wiley

Venkataraman P. (2009): *Applied Optimization with MatLab Programming.* Second Edition, John Wiley

Zarchan P. i Musoff H. (2009): *Fundamentals of Kalman Filtering: A Practical Approach.* American Institute of Aeronautics and Astronautics

Skorowidz

Algorytm przydziału procesora
– *Earliest Deadline First*, 337
– *round-robin sheduling*, 337
Algorytm regulacji
– uchyb statyczny, 299
Asembler, 23
– dyrektywy, 24, 79, 80
– etykieta, 23, 78, 79
– format instrukcji, 78
– komentarz, 23, 78, 80
– mnemonik, 23, 78
– operandy, 23, 78
– pseudoinstrukcje, 100

Bajt, 13
– kolejność Big Endian, 102
– kolejność Little Endian, 102
Bieguny dzwoniące, 325
Bit, 12
– LSB, 13
– MSB, 13, 16
Bramka
– NAND, 12
– NOR, 12

CMSIS, 5, 45

Debugowanie
– *Trace ETM*, 4, 193
Delta Diraca, 314
Dioda, 7
– LED, 8, 68, 73
Dokumentacja do pobrania
– STM3210E-EVAL, 188
– STM32F103, 42
Działania logiczne, 12
– NAND, 12
– NOR, 12

ELF, 24, 25

Flaga
– Borrow, 19, 108
– Carry, 17, 19, 20, 85, 108
– Negative, 19, 85, 108
– Overflow, 19, 20, 85, 108
– Saturation, 85
– Zero, 19, 85

IDE Keil μVision, 44

Kod uzupełnień
– do dwóch, 17
– do jedności, 16
Kolejka
– FIFO, 30
– LIFO, 30
Konfiguracja
– ADC1, 254
– DAC, 269
– DMA, 244, 260
– GPIO, 72
– NVIC, 150, 282, 285
– RCC, 70
– RTC, 161
– SysTick, 151
– TIM2, 173
– TIM4, 168
– USART1, 280
Krzem, 7

Liczby
– binarne, 15
– całkowite bez znaku, 16
– całkowite ze znakiem, 16
– hexadecymalne, 15
– stałopozycyjne bez znaku, 216
– stałopozycyjne wymierne, 215
– stałopozycyjne ze znakiem, 217
Licznik programu, 28, 33, 134
Link Register, 30, 33, 35, 37, 134, 339
Lokowanie biegunów, 321
– z redukcją szumu pomiarowego, 329, 334
– z wewnętrznym integratorem, 327, 329, 334

Mikrokontroler
– Cortex-M, 76

– schemat STM32F103VB, 242
– schemat STM32F103ZE, 243
– STM32F103VB, 40
– STM32F103ZE, 187

Obiekt regulacji
– oscylacyjny, 295, 334
– wieloinercyjny, 296
– z podwójnym całkowaniem, 323
Opcje
– asemblera, 55
– kompilatora C, 54
– linkera, 55
– programatora, 56
– projektu, 49
Operator opóźnienia q^{-1}, 315
Opornik, 68

Półprzewodnik, 7
Pamięć
– Flash, 25, 27, 29, 30
– RAM, 25, 27
– ROM, 25
– S-RAM, 29, 30
Potokowość, 34
Programator
– j-link, 43
– ULINKpro, 188, 193
Projekt
– Projekt00, 46
– Projekt01_LED, 69, 189
– Projekt02_LCD, 122
– Projekt03_asm, 138
– Projekt04_SysTick, 149
– Projekt05_keypad, 155
– Projekt06_scanf, 159
– Projekt07_RTC, 162
– Projekt08_toggleLED, 166
– Projekt09_TIM2switch, 172
– Projekt10_RadioControl, 175
– Projekt11_6WheelDrive, 180
– Projekt12_DistanceSensor, 184
– Projekt14_TFTLCD, 189
– Projekt15_StepperMotor, 201
– Projekt16_StepperMotor, 211
– Projekt17_FixedPoint_sin, 230
– Projekt18_cube3D, 232
– Projekt19_cube3D_FixedPoint, 238
– Projekt20_DMA_M2M, 244
– Projekt21_ADC1, 253
– Projekt22_4-20mA, 258
– Projekt23_DAC_sin, 268
– Projekt24_1_Modbus_Master, 278
– Projekt24_2_Modbus_Master, 284
– Projekt25_Modbus_slave, 282
– Projekt26_I2C, 290
– Projekt27_SPI, 292
– Projekt28_obiekt, 296
– Projekt29_PI, 303
– Projekt30_PID, 307
– Projekt31_1_pole_placement, 325
– Projekt31_2_pole_placement, 333
– Projekt31_3_pole_placement, 334
– Projekt32_Multitasking, 339
Przekazywanie parametrów, 137
Przerwania, 4
– aktywne, 146
– oczekujące, 146
– priorytety, 146
– zabranianie, 145
– zewnętrzne, 151
– zezwalanie, 145

Równanie diofantyczne, 321
Regulacja
– cel, 299
– obiekt, 299
– PID – czas wyprzedzenia, 305
– PID – czas zdwojenia, 303
– równanie PID, 299
– uchyb, 299
– wartość zadana, 299
RISC, 1
RTOS, 4, 337, 341

Silnik krokowy
– bipolarny, 198
– hybrydowy, 197
– o zmiennej reluktancji, 196
– sterowanie dwufazowe, 200
– sterowanie jednofazowe, 199
– sterowanie L/nR, 208
– sterowanie półkrokowe, 200
– sterowanie z kluczowaniem, 209
– unipolarny, 198
– z magnesem trwałym, 197
Sterta, 29, 30, 45
Stos, 29, 30
– *ascending*, 135
– *descending*, 135
– *empty*, 135
– *full*, 135
– *interrupt stacking*, 144, 338
– *interrupt unstacking*, 144, 339

– wskaźnik, 28, 30
– wskaźnik MSP, 33, 142, 338
– wskaźnik PSP, 33, 142, 338

Task, 341
– kontekst, 339
Transmisja szeregowa
– CRC, 277
– I^2C, 63, 66, 287
– INTEL HEX Format, 275
– Modbus, 275
– prędkość bodowa, 273
– ramka, 272
– SPI, 291
– USART, 272
Tranzystor
– bipolarny, 9, 68, 73
– CMOS, 11
– MOSFET, 10

Wejście GPIO
– Schmitta, 64
Wydajność procesora, 3
– CoreMark, 3
– DMIPS, 3
– FLOPS, 3
– MIPS, 3
Wyjście GPIO, 65
– komplementarne, 65
– obciążalność prądowa, 68
– otwarty dren, 66
Wzory Viéte'a, 322

Zestaw uruchomieniowy
– STM3210E-EVAL, 187
– ZL27ARM, 40
Zmienna zespolona z, 315
Zmienne
– globalne, 29, 30
– lokalne, 29
Znak
– ASCII, 22
– CR, 22
– LF, 22
– NUL, 22
Zworki BOOT, 41, 60