2. Przykładowe architektury mikrokontrolerów 32 bitowych

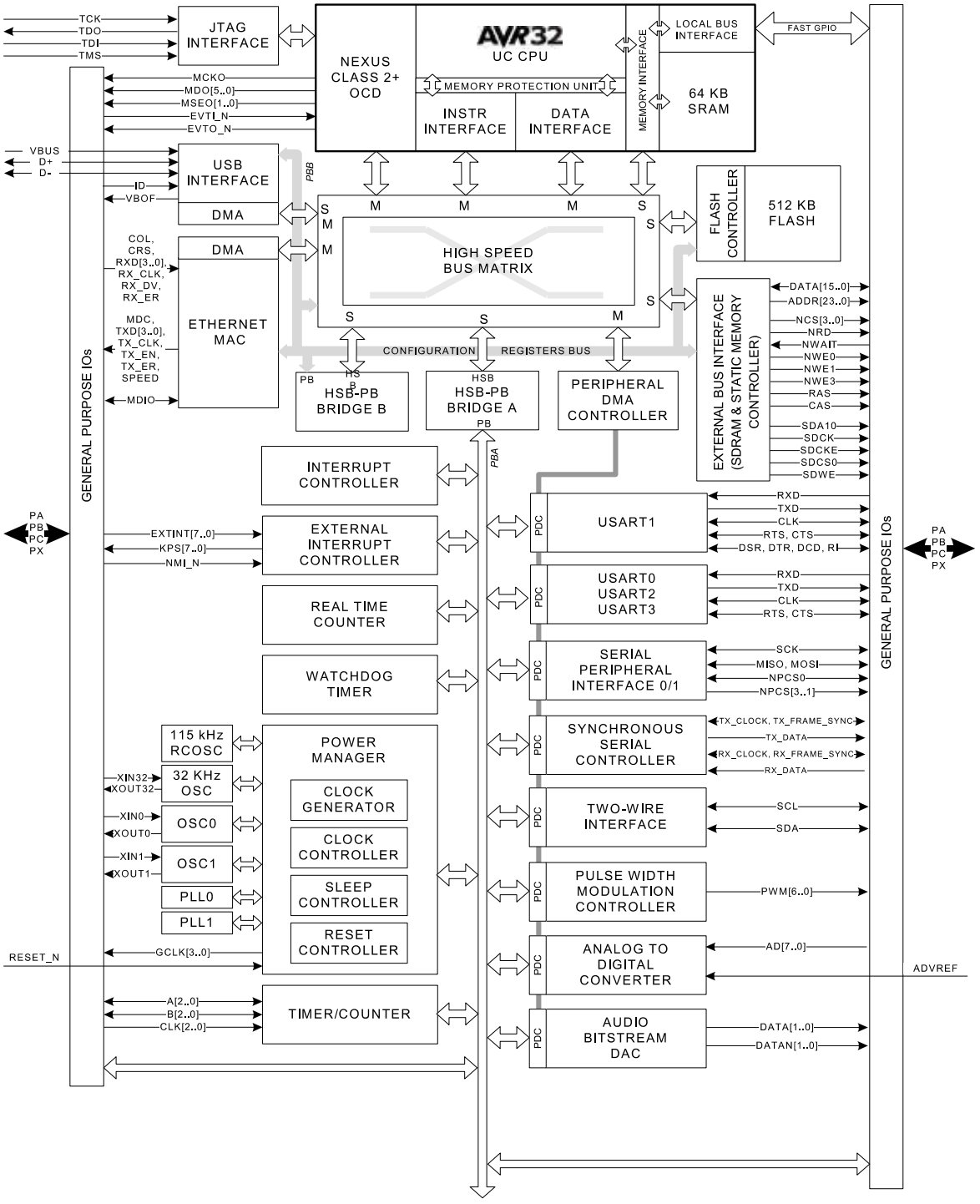
Na rynku obecnie są dostępne mikrokontrolery o architekturach 4-, 8-, 16- i 32-bitowych. Mikrokontroler 80C51 jest układem 8-bitowym, chcąc więc uzyskać skok jakościowy nowego rozwiązania należało rozważyć zastosowanie układu 16- lub 32-bitowego. Według prognoz IC Insights liczba sprzedawanych mikrokontrolerów 16-bitowych będzie systematycznie malała na rzecz znaczącego wzrostu sprzedaży mikrokontrolerów 32-bitowych [3]. Również biorąc pod uwagę liczbę dostawców i ich portfolio widać, że mikrokontrolery 32-bitowe zdominowały rynek urządzeń embedded. W związku z powyższym autor niniejszej pracy skupił się jedynie na rozwiązaniach 32-bitowych, jako przyszłościowych i jeszcze długo wspieranych.

Obecnie rynek mikrokontrolerów o architekturze 32-bitowej można podzielić na dwie kategorie: pierwsza – oparta na rozwiązaniach własnych firm produkujących mikrokontrolery, druga – mikrokontrolery produkowane na licencji udzielanej przez ARM Holdings, z czego mikrokontrolery z rdzeniem ARM stanowią 75% rynku urządzeń wbudowanych i ciągle się na nim umacniają. Konstrukcje oparte na rozwiązaniach własnych wydają się nieco zapomniane, ale na pewno warto się im bliżej przyjrzeć.

2.1. Atmel AVR32

Jednym z rozwiązań własnych są mikrokontrolery AVR32 firmy Atmel. Rodzina 8-bitowych mikrokontrolerów firmy Atmel o nazwie AVR jest szeroko stosowana w prostych systemach embedded. W niektórych przypadkach wystarczy tylko podłączenie zasilania do mikrokontrolera AVR, aby uzyskać w pełni wartościowy system mikroprocesorowy. Idąc tym tropem firma Atmel wprowadziła rozwinięcie 8-bitowych układów o 32-bitową rodzinę AVR32. Rodzina składa się z układów o różnych mikroarchitekturach. Producent na chwilę obecną oferuje mikrokontrolery AVR32 następujących serii: L, C, D, A3, A4, A0, A1, B oraz Audio. Serie różnią się od siebie wydajnością, poborem mocy, wbudowanymi peryferiami, sprzętową obsługą np. pojemnościowych paneli dotykowych, czy wyposażenia w moduł szyfrująco/deszyfrujący AES. Schemat blokowy mikrokontrolera serii AT32UC3A został przedstawiony na rys. 3.1.

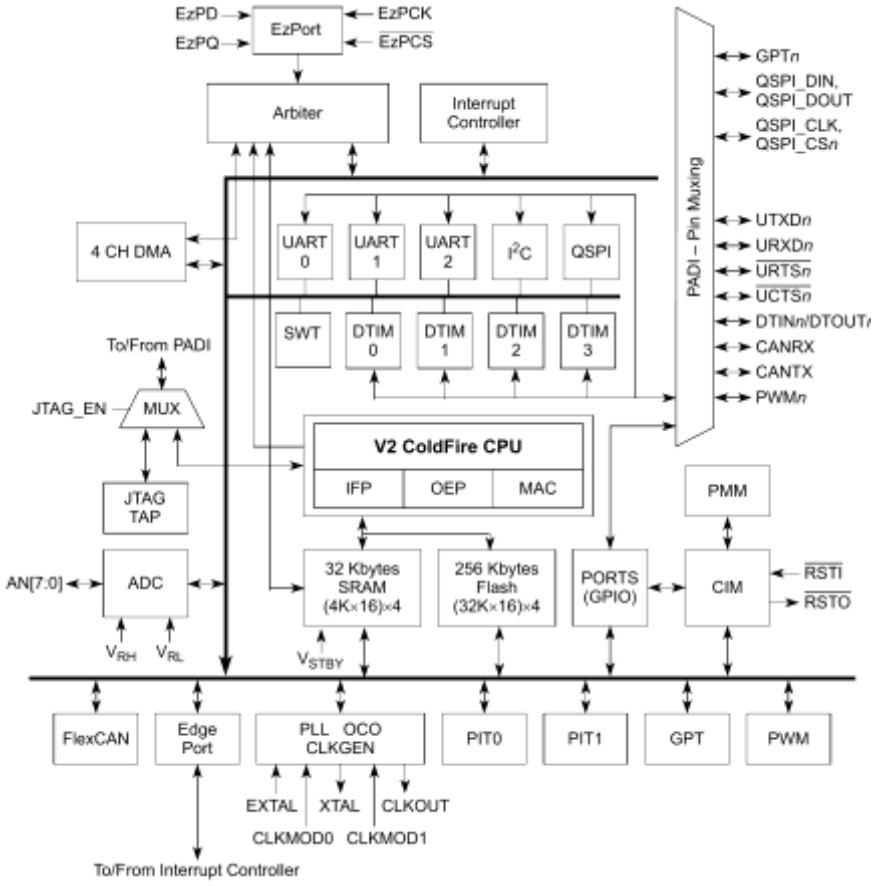
W mikrokontrolerach tych zwraca uwagę szczególnie duża wydajność CPU, która dochodzi do 1.5 MIPS/MHz. Z drugiej jednak strony serie, które charakteryzują się mniejszymi wydajnościami mają pobór prądu wynoszący 165 µA/MHz w trybie normalnej pracy i 600 nA/MHz w trybie uśpienia. Jak łatwo policzyć przy częstotliwości zegara 60 MHz pobór prądu w pracy normalnej wynosi 9,9 mA, a podczas czuwania jedynie 36 µA. Dla porównania zwykła standardowa dioda LED o rozmiarze 5 mm pobiera prąd około 10-20 mA. Firma Atmel stawia też na wysoko rozwinięte wsparcie dla użytkowników. Na jej stronie można odnaleźć wiele przykładów, narzędzi programowania oraz not aplikacyjnych, a na rynku dostępne są zestawy ewaluacyjne w przystępnych cenach dla układów rodziny AVR32. Nie mniej jednak autor pracował z zestawem ewaluacyjnym procesora AT32UC3A0512 jeszcze kilka lat temu z użyciem dedykowanej platformy i o ile wydajnościowo potwierdzały się zapewnienia producenta o tyle wsparcie techniczne i prostota programowania w analogii do 8-bitowych AVR już nie. W bibliotekach udostępnianych przez firmę Atmel znajdowały się błędy tak samo jak w dokumentacjach, co skutecznie zwiększało ilość potrzebnego czasu na uruchomienie nawet podstawowych funkcjonalności. W celu poprawnej pracy niektórych elementów należało np. w kodzie w języku C używać tak zwanych wstawek asemblerowych czy innych zabiegów. Całość spowodowała, że rodzina procesorów AVR32 została uznana za nienadającą się do użycia w celach dydaktycznych.



Rys. 2.1. Schemat blokowy mikrokontrolera AT32UC3A, opracowano wg [20]

2.2. Freescale ColdFire

Firma Freescale Semiconductor powstała w 2004 roku poprzez wydzielenie działu produkcji półprzewodników z firmy Motorola. 32-bitowe rdzenie ColdFire są kontynuacją mikrokontrolerów serii M68000 produkowanych przez Motorolę, a zestaw instrukcji asemblerowych obu tych rdzeni jest prawie kompatybilny.



Rys. 2.2. Schemat blokowy mikrokontrolera MCF5213 z rdzeniem ColdFire V2, opracowano wg [21]

Obecnie z rdzeniem ColdFire dostępne są mikrokontrolery w dwóch wersjach – oznaczanych jako MCF51 (v1) oraz MCF52 (v2). Na rys. 3.2 przedstawiony jest przykładowy schemat blokowy mikrokontrolera MCF5213 z rdzeniem v2. Oprócz tego dostępna jest także zmodernizowana wersja rdzeni v1 oznaczana jako ColdFire+. Rdzenie v1 są uproszczoną wersją rdzeni v2 i są wykorzystywane w mniej wymagających aplikacjach. Są kontynuacją architektury S08, a producent zachował nawet kompatybilność wyprowadzeń z tą serią, przez co migracja starszych rozwiązań do 32-bitowych jest prostsza. Zachowano również jedno wyprowadzeniowy interfejs debugera o nazwie BDM, przez co, jak podaje firma Freescale, możliwe jest wykorzystanie starych programów narzędziowych przeznaczonych dla mikrokontrolerów serii S08. Maksymalną wydajność jaką osiągają to 47 MIPS przy taktowaniu 50 MHz.

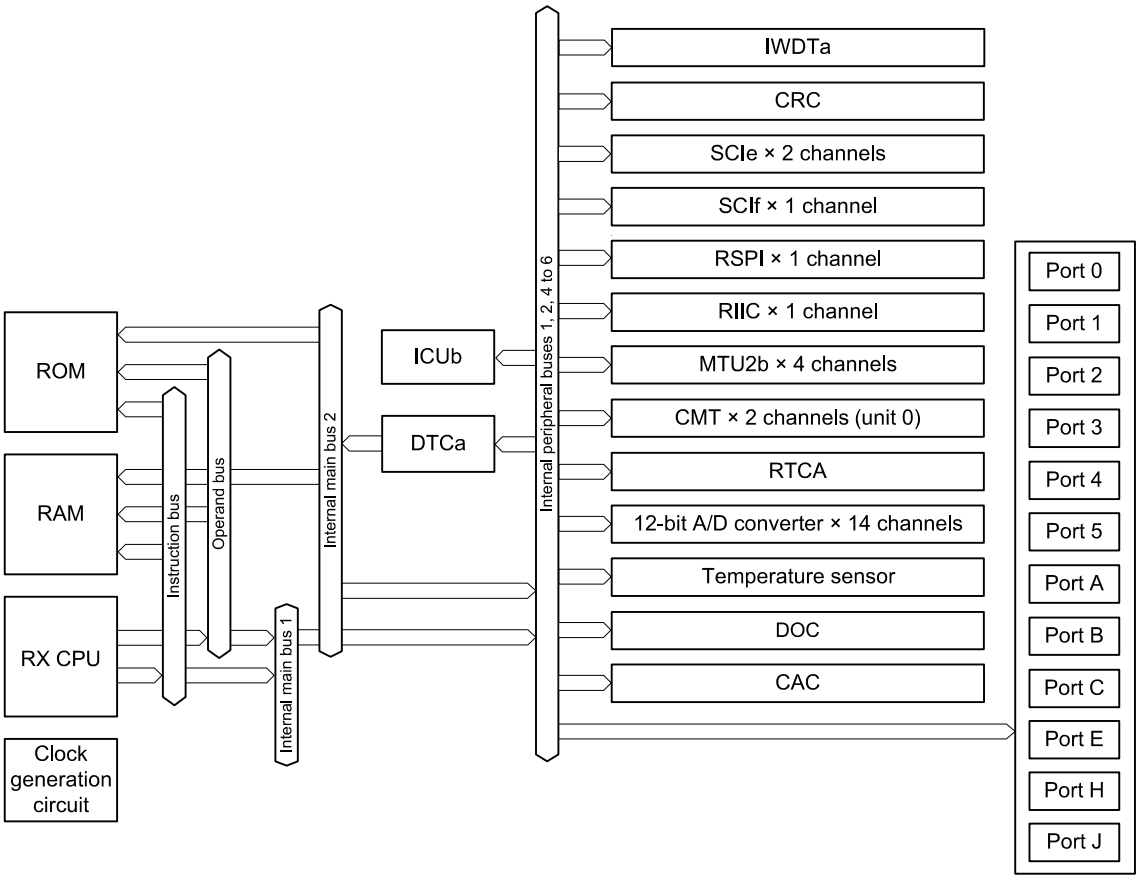
Mikrokontrolery z rdzeniem v2 są polecane do bardziej wymagających aplikacji. Większa jest wydajność i częstotliwość taktowania, które dochodzą do 80 MIPS przy 80 MHz. Mikrokontrolery posiadają większą ilość pamięci zarówno Flash jak i RAM, które dochodzą odpowiednio do 512 kB i 64 kB (w mikrokontrolerach z rdzeniem v1 do 256 kB i 32 kB), dostępne są także obudowy o większej ilości wyprowadzeń.

Rdzeń ColdFire+ został zastosowanych w rodzinach mikrokontrolerów charakteryzujących się lepszym wyposażeniem niż mikrokontrolery z rdzeniem v1 przy jednoczesnym niskim poborze prądu. Wszystkie zamknięte są w miniaturowych obudowach z przeznaczeniem głównie na rynek urządzeń przenośnych.

2.3. Renesas RX100, RX200, RX600, RX700, V850, RH850, SuperH

Tak samo jak w przypadku dwóch wcześniej opisywanych producentów, również firma Renesas swoją architekturę 32-bitową wprowadziła jako kontynuacja starszych rozwiązań. W jej przypadku rodzina mikrokontrolerów RX wywodzi się ze wcześniejszych

rodzin H8 oraz M16C i jest ona kompatybilna pod względem wyprowadzeń. Mikrokontrolery serii RX są również kompatybilne pod względem wyprowadzeń pomiędzy sobą. Seria RX100 przedstawiana jest jako rozwiązanie dla układów o ograniczonym poborze prądu i niewielkich wydajnościach. Mikrokontrolery charakteryzują się częstotliwością taktowania do 32 MHz, pamięcią Flash z przedziału 8-512 kB oraz posiadają interfejsy takie jak USB, LCD czy do obsługi paneli pojemnościowych. Schemat blokowy przykładowego mikrokontrolera serii RX110 został przedstawiony na rys. 3.3. Większą wydajnością charakteryzują się mikrokontrolery serii RX200, w których to częstotliwość taktowania sięga 54 MHz, a ilość pamięci Flash 1 MB. Seria RX200 jest też bardziej bogato wyposażona w różnorodne peryferia takie jak: interfejs CAN, przetwornik analogowo-cyfrowy typu delta-sigma czy bloki do sterownia silnikami 3-fazowymi. Producent przedstawia mikrokontrolery tej serii jako kompromis pomiędzy wydajnością, a oszczędnością energii. Energooszczędne nie są już mikrokontrolery dwóch najbardziej wydajnych serii to znaczy RX600 i RX700. Seria RX600 może być taktowana zegarem o częstotliwości do 120 MHz, natomiast RX700 dwukrotnie wyższym. W mikrokontrolerach obu serii maksymalna ilość pamięci Flash wynosi 4 MB. Seria RX600 podobnie jak RX200 posiada kontroler silników 3-fazowych, natomiast RX700 dwa kontrolery Ethernet. Pomiędzy układami występuje również różnica w rdzeniach. Firma Renesas oferuje dwa rdzenie 32-bitowe do serii mikrokontrolerów RX: RXv1 oraz RXv2. Mikrokontrolery serii RX100 wyposażone są tylko w rdzenie RXv1. W Seriach RX200 i RX600 są układy zarówno z rdzeniem jednym jak i drugim, natomiast w serii RX700 dostępne są rdzenie tylko RXv2.



Rys. 2.3. Schemat blokowy mikrokontrolera RX110, opracowano wg [22]

Mikrokontrolery serii V850 tej firmy w latach 2008 i 2009 były najczęściej sprzedawanymi mikrokontrolerami 32-bitowymi na świecie. Ich główną branżą, w której znalazły zastosowanie to motoryzacja ze względu na wysoką wydajność oraz spełnianiu

surowych norm obowiązujących w sektorze motoryzacji. Wysoką wydajność sięgającą do 1,9 MIPS/MHz osiągają dzięki rdzeniom G3. Dostępne mikrokontrolery tej serii charakteryzują się ilością wyprowadzeń od 32 do 256, pamięcią Flash do 4MB, pamięcią RAM do 256 kB oraz maksymalnym zegarem 200 MHz.

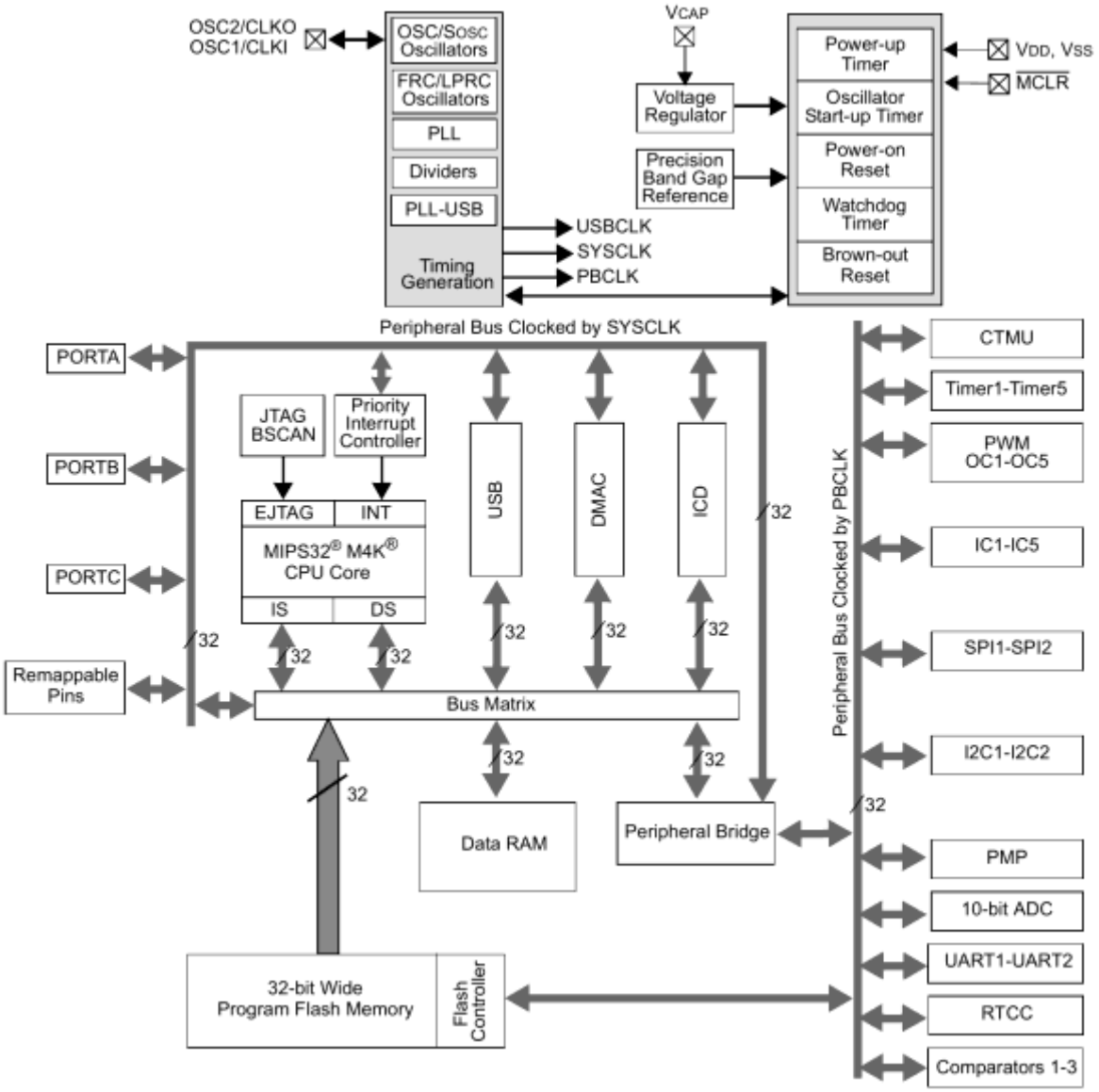
Rozwinięciem serii V850 są mikrokontrolery serii RH850. Produkowane są w technologii 40 nm, co pozwoliło na ograniczenie poboru prądu. Wyposażone są w sprzętowe wspomaganie szyfrowania danych oraz szereg interfejsów komunikacyjnych, takich jak: 6xCAN, 2xCSI, 4xQCSI, 6xUART/LIN, 10xLIN.

Ostatnimi mikrokontrolerami 32-bitowymi produkowanymi przez firmę Renesas są układy wyposażone w rdzeń SuperH. Sam rdzeń został opracowany w latach 90-tych przez firmę Hitachi. Renesas stworzył serię kompatybilnych rdzeni z oznaczeniami SH-1…SH-7, niektóre z nich z jednostkami DSP i koprocesorem arytmetycznym. Mikrokontrolery tej serii znalazły zastosowanie przede wszystkim jako układy wbudowane w napędy CD-ROM, większe urządzenia AGD czy konsole do gier. Ciekawostką są rdzenie SH-2, które pomimo 32-bitowej architektury, używają 16-bitowych rozkazów w celu oszczędności pamięci.

2.4. Mikrochip PIC32

Firma Mikrochip jest znana przede wszystkim ze swoich bardzo popularnych mikrokontrolerów 8- i 16-bitowych. Wprowadziła do swojej oferty mniej znane mikrokontrolery 32-bitowe o nazwie PIC32. Mikrokontrolery posiadają rdzeń z rodziny MIPS32 o nazwie M4K. Rdzenie te są oferowane również jako rozwiązania do implementacji przez innych producentów. Charakteryzują się wydajnością do 1,53 MIPS/MHz, a częstotliwości taktowania sięgają 80 MHz. Rdzeń M4K posiada rozdzielone magistrale danych i instrukcji w związku z czym są to konstrukcje o architekturze harvardzkiej. Schemat blokowy przykładowego mikrokontrolera rodziny PIC32 został przedstawiony na rys. 3.4.

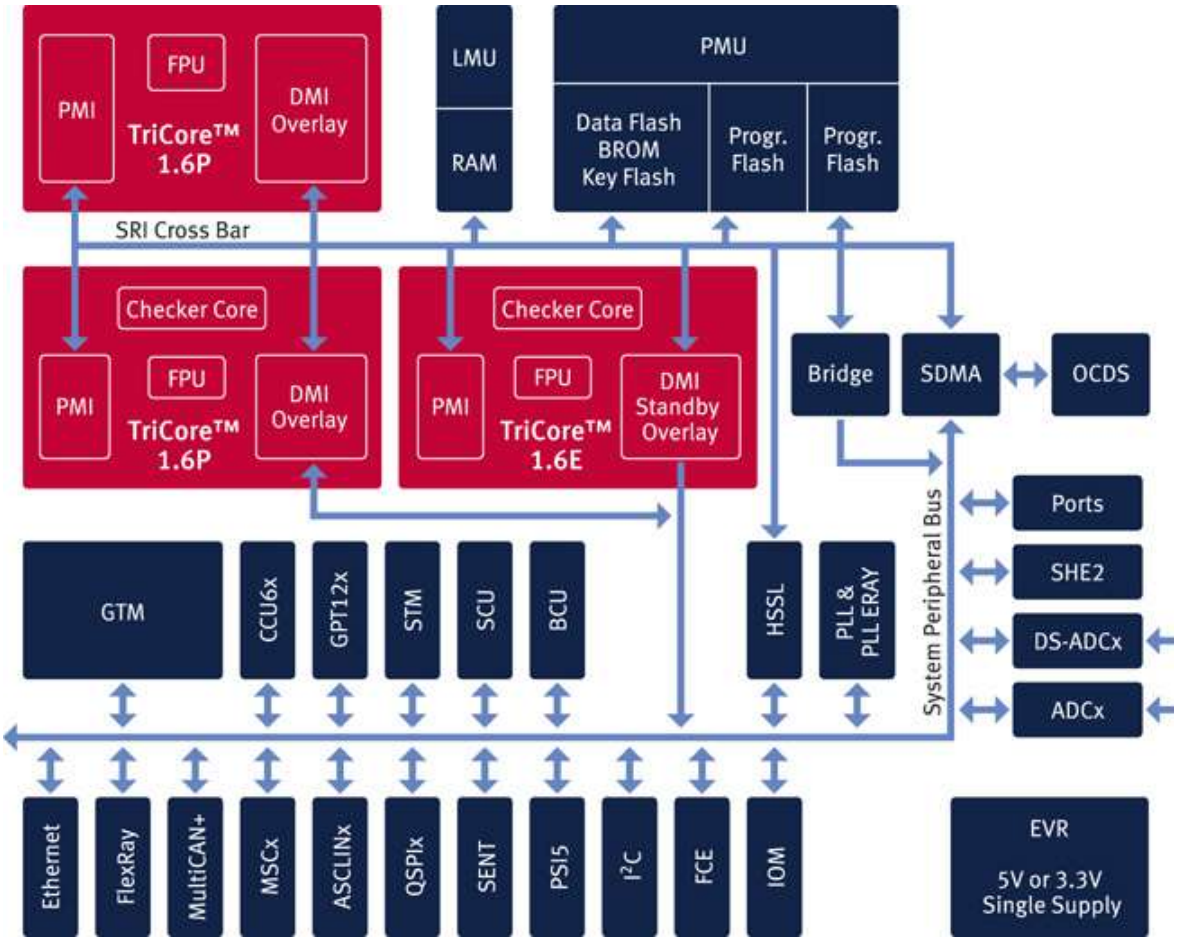
Producent inaczej niż u konkurencji rozwiązał sprawę obsłużenia przerwania, gdyż dzięki podwójnym zestawom rejestrów, które są naprzemiennie przełączane, nie ma potrzeby zapisywania na stosie aktualnego stanu procesora, dzięki czemu skraca się czas obsługi przerwania. Rdzenie posiadają moduł sprzętowego mnożenia i dzielenia, dzięki czemu mnożenie i dzielenie wykonywane są w pojedynczym cyklu zegarowym. Mikrokontrolery posiadają od 32 do 512 kB pamięci Flash. Wyposażone są w szereg interfejsów szeregowych (do 6xUART, 5xI^2C, 4xSPI), a także wyprowadzenie magistrali zewnętrznej, gdzie magistrala adresowa może być maksymalnie 16-bitowa, a magistrala danych 8- lub 16-bitowa. Oprócz tego posiadają interfejsy USB oraz Ethernet MII/RMII. Mikrokontrolery mogą pracować w kilku trybach oszczędzania energii. Producent udostępnia szerokie wsparcie dla użytkowników, w tym darmowe biblioteki.



Rys. 2.4. Schemat blokowy mikrokontrolera z rodziny PIC32, opracowano wg [23]

2.4.1. Infineon AUDO, AURIX

Mikrokontroler AUDO został wyprodukowany przez firmę Infineon w 1999 roku. Jego rdzeń o nazwie TriCore charakteryzował się łączeniem cech: RISC, MCU oraz DSP. Od tego czasu architektura została unowocześniona, na bazie której powstała rodzina procesorów AURIX. Schemat blokowy przykładowego mikrokontrolera rodziny AURIX został przedstawiony na rys. 3.5.



Rys. 2.5. Schemat blokowy mikrokontrolera z rodziny AURIX, opracowano wg [24]

Producent poleca ją do zastosowań w branży głównie motoryzacyjnej, choć znajdują zastosowanie również w innych dziedzinach. Rdzeń TriCore został opracowany z myślą o realizacji skomplikowanych algorytmów obliczeniowych takich jak obsługą wytrysków paliwa czy sterowaniem niskiej emisji spalin. Znajdują zastosowanie także w samochodach elektrycznych i hybrydowych. W Polsce mikrokontrolery nie cieszą się dużą popularnością.

2.5. Mikrokontrolery z rdzeniem ARM® - zarys historii

Jeszcze kilkanaście lat temu producenci półprzewodników oferowali mikroprocesory z rdzeniami własnymi bądź z architekturami opracowanymi przez pionierów rynku takimi jak Intel czy Motorola. Sytuacja diametralnie zmieniła się za sprawą firmy ARM Holdings.

Firma powstała w 1990 roku pod ówczesną nazwą Advanced RISC Machines z porozumienia kilku przedsiębiorstw (Apple Computer, Acorn Computer Group, VLSI Technology). Następnie w 1998 roku nazwa została zmieniona na ARM Holdings. Firma nie zajmuje się produkowaniem mikrokontrolerów ani żadnych półprzewodników. Opracowuje architektury i rdzenie mikroprocesorów, a następnie sprzedaje projekty, a dokładniej bloki własności intelektualnej IP (ang. *Intellectual Property*) producentom mikroprocesorów. Producenci półprzewodników obudowują rdzenie w różne ilości pamięci Flash oraz RAM, różnorodne interfejsy, przetworniki A/C, C/A, kontrolery LCD, paneli dotykowych oraz wiele, wiele innych oferując setki kombinacji mikrokontrolerów z rdzeniami ARM różniące się między sobą wyposażeniem, pamięcią, obudową i w związku z tym ceną. Klientami firmy ARM są znani producenci półprzewodników,

między innymi [4]: Altera, Analog Devices, Atmel, Freescale, Fujitsu, Infineon, nVidia, NXP (dawny Philips), Renesas, Samsung, Silicon Labs, STMicroelectronics, Texas Instruments, Toshiba, VIA czy Xilinx. Obecnie ARM podaje, że udzielonych ma kilkaset licencji na produkowanie opracowanych przez nich rdzeni.

Pierwszy rdzeń ARM powstał jeszcze przed założeniem ARM Holdings w roku 1985, a został opracowany przez firmę Acorn Computer Group. Szybko okazało się, że nowe rozwiązanie jest bardzo ciekawe i należy kontynuować jego rozwój. Sukces rynkowy rozpoczął się w 1993 roku za sprawą rdzenia ARM7TDMI, który został opracowany już przez ARM Holdings. Charakteryzował się wysoką wydajnością przy niewielkim poborze prądu, a zarazem niewielkim kosztem implementacji. Rdzeń szybko znalazł zastosowanie we wszelakich urządzeniach przenośnych w tym telefonach komórkowych, palmtopach czy grach multimedialnych. Pięć lat później, bo w roku 1998 ARM Holdings opracowuje rdzeń ARM9, który w porównaniu z ARM7 jest bardziej energochłonny, ale i znacznie bardziej wydajny. Jak widać hasłowe mikrokontrolery ARM są konstrukcją już z dość długą historią jednak do roku 1998 dużo się o nich słyszało i mówiło, były wykorzystywane w różnych urządzeniach jednak konstruktor w zasadzie nie miał możliwości opracowania projektu bazującego na jakimkolwiek mikrokontrolerze ARM. Wynikało to z faktu, iż popularni producenci półprzewodników nie mieli w swoich ofertach mikrokontrolerów z rdzeniami ARM i promowali swoje własne rozwiązania. Taktykę zmieniła dopiero firma Atmel w 1998 roku wprowadzając do produkcji mikrokontrolery z rdzeniem ARM7TDMI o nazwie AT91M. Nie posiadały one jednak wewnętrznej pamięci Flash, co sprawiło, że nie osiągnęły dużej popularności, ale coś na rynku drgnęło, choć konstruktorom przyszło jeszcze czekać kolejne 6 lat na prawdziwy przełom. Przełom nastąpił w 2004 roku za sprawą firmy Philips (obecnie NXP), która wprowadziła do sprzedaży procesory serii LPC2000 bazujące na rdzeniu rodziny ARM7. Mikrokontrolery posiadały już wewnętrzną pamięć Flash i RAM, wyposażone również były w bogate peryferia takie jak interfejsy SPI, I^2C, UART czy przetworniki A/C. Konkurencja widząc rynkowy sukces zaczęła powoli wprowadzać do swojej oferty podobne rozwiązania.

Oczywiście w latach 1998-2004 ARM Holdings nie próżnuje i oferuje coraz to nowe konstrukcje rdzeni, powstaje ARM11. W roku 2004 do portfolio dochodzi całkiem nowa rodzina rdzeni o nazwie Cortex, a dokładnie rdzeń Cortex-M3. Charakteryzował się w porównaniu z klasycznymi rdzeniami ARM niższym kosztem produkcji (powierzchnia na krzemie zajmowana przez rdzeń to jedyne 0,56 mm^2 przy technologii 0,18 μm), szybszą obsługą przerwań, mniejszym poborem mocy oraz mniejszym rozmiarem kompilowanego kodu [5]. Jako pierwsza wprowadziła do swojej oferty mikrokontrolery z rdzeniem Cortex firma Luminary Micro (w 2009 roku wykupiona przez Texas Instrumensts) tworząc wiosną 2007 roku rodzinę Stellaris. Następnie z tym samym rdzeniem w drugiej połowie 2007 roku na rynek wchodzą mikrokontrolery serii STM32 produkcji STMicroelectronics, a pod koniec 2008 roku również firma NXP wprowadza do sprzedaży mikrokontrolery LPC1700 także z rdzeniem Cortex-M3. Decyzje tych firm zachęciły konkurencję to analogicznych posunięć. Rodzina rdzeni Cortex stała się na tyle popularna, że od 2006 roku ARM Holdings do oferty wprowadza już tylko nowe rdzenie będące rdzeniami serii Cortex. Obecnie dostępne są mikrokontrolery z rdzeniami ARM Cortex pochodzące od ponad 20 producentów.

2.6. Mikrokontrolery z rdzeniem ARM® Cortex™-M

Mając na uwadze powyższe, aby zastosować współczesne rozwiązania mikrokontrolerów z rdzeniem ARM należy sięgnąć po jeden ze rdzeni Cortex.

Rys. 2.6. Logo rodziny rdzeni Cortex ARM

Rodzina Cortex składa się z trzech podrodzin: Cortex-A, Cortex-M, Cortex-R. Krótka ich charakterystyka została przedstawiona w tab. 3.1.

Tablica 2.1 Charakterystyka podrodzin rdzeni Cortex

Rodzina	Opis
Cortex-A	Najbardziej wydajne i zarazem najdroższe ze rdzeni Cortex, zoptymalizowane do wykorzystania w urządzeniach, na których instalowane są systemy operacyjne, takie jak Linux, Windows Embedded, Android. Obsługują dużą ilość pamięci RAM, posiadają interpreter języka JAVA. Wykorzystywane głównie w telefonach komórkowych i tabletach.
Cortex-R	Rdzenie Cortex przeznaczone do systemów czasu rzeczywistego, tam gdzie szybkość obsługi zdarzeń jest najbardziej krytyczny (np. systemy ABS).
Cortex-M	Wersja zoptymalizowana pod względem ceny przy jednoczesnym zachowaniu dużej wydajności. Zalecana do zastosowań w systemach embedded konsumenckich i przemysłowych.

Z myślą o mikrokontrolerach, które mogłyby z powodzeniem zastępować starsze rodziny takie jak 51', PIC, czy AVR w urządzeniach wbudowanych została stworzona rodzina Cortex-M. Z tego też względu ARM Holdings sprzedał najwięcej licencji, na produkcję rdzeni Cortex-M przy mniejszej liczbie licencji na rdzenie Cortex-A oraz Cortex-R. Czasami producenci oferują procesory wielordzeniowe jak np. OMPA4470 firmy Texas Instruments, gdzie zaimplementowany jest rdzeń Cortex-A9 z dwoma pomocniczymi rdzeniami Cortex-M3 lub też OMAP5 z jednym Cortex-A15 oraz dwoma Cortex-M4. Nie mniej jednak mikrokontrolery z rdzeniem Cortex-M są według autora niniejszej pracy najbardziej odpowiednie w kwestii zachowania dydaktycznego charakteru oraz odpowiedniej niskiej ceny (kilkanaście – kilkadziesiąt PLN) w stosunku do wydajności. Stąd też dalszy opis będzie skupiony tylko na tej rodzinie.

Seria rdzeni Cortex-M składa się z kilku wersji, których główne cechy zostały zestawione w tab. 3.2. Różnic pomiędzy poszczególnymi rdzeniami jest bardzo dużo. Chociażby rdzenie różnią się między sobą rodzajem architektury – rdzenie M0/M0+/M1 posiadają architekturę Von Neumann'a, a rdzenie M3/M4/M4F/M7 są zbudowane na architekturze Harvardzkiej. Różnią się także obecnością jednostki MPU (*ang. Memory Protection* Unit), czy też timerem SysTick, możliwością wykonywania operacji Bit-banding (operacji na pojedynczych bitach w pamięci). Co więcej, część z wyposażenia może być stosowana jako opcjonalna przez producentów, to znaczy, że np. mikrokontrolery z rdzeniem Cortex-M3 jednego producenta mogą posiadać jednostkę MPU, a drugiego już nie. Przed wyborem odpowiedniego, należy więc starannie przejrzeć noty katalogowe w celu weryfikacji wszystkich wymaganych cech.

Tablica 2.2 Wersje rdzeni ARM Cortex-M

Wersja	Rok opracowania	Opis
Cortex-M0	2009	Najtańszy, najbardziej energooszczędny rdzeń Cortex-M przeznaczony głównie jako zastępnik za mikrokontrolery 8-/16-bitowe w prostszych aplikacjach. Charakteryzują się niewielką wydajnością oraz niewielkim poborem mocy, lista rozkazów asemblera wynosi 56 instrukcji.
Cortex-M0+	2012	Zmodernizowana wersja rdzenia Cortex-M0 z tą samą listą instrukcji, bardziej energooszczędna (ARM reklamuje mikrokontrolery z rdzeniem M0+ jako pracujące 15 lat na jednej baterii)
Cortex-M1	2007	Dostępny jest tylko jako wersja do implementacji w układach FPGA (np. firm Altera czy Xilinx)
Cortex-M3	2004	Rdzenie oferujące dużą prędkość wykonywania kodu, ale jednocześnie oferujące wspomaganie dla oszczędności energii. Posiadają rozszerzoną listę instrukcji w stosunku do M0/M0+/M1
Cortex-M4/M4F	2010	Konstrukcja oparta na Cortex-M3, ale posiadające CPU z rozszerzeniem DSP oraz możliwą osobną jednostką FPU (rdzenie M4F). Posiadają rozszerzoną listę instrukcji w stosunku do Cortex-M3.
Cortex-M7	2014	Rozbudowana wersja Cortex-M4 dla najbardziej wymagających aplikacji. Charakteryzuje się szybszą pamięcią SRAM, szybszymi magistralami, ale i wyższą ceną.

Po przeanalizowaniu różnic, z punktu widzenia zastosowania w laboratorium systemów mikroprocesorowych najbardziej odpowiedni wydają się być mikrokontrolery z rdzeniami Cortex-M3/M4.

2.7. Przykładowe zestawy ewaluacyjne dla mikrokontrolerów z rdzeniem ARM Cortex-M3/M4

2.7.1. Zestaw EasyMxPRO™v7 for STELLARIS® ARM®

Zestawów ewaluacyjnych dla mikrokontrolerów z rdzeniem ARM Cortex-M3/M4 jest na rynku bardzo wiele, a różnią się przede wszystkim ceną i wyposażeniem, od takich zawierających dwa przyciski, tyle samo diod LED i wyprowadzone porty po bardzo rozbudowane zestawy z wbudowanymi kolorowymi dotykowymi panelami LCD, różnego typu sensorami i pełną gamą interfejsów. Autor niniejszej pracy magisterskiej dokonał przeglądu z jakich zestawów korzystają inne politechniki w ramach laboratorium układów mikroprocesorowych. Ciekawe zajęcia mają studenci Katedry Radiokomunikacji Politechniki Poznańskiej. Na laboratorium wykorzystują zestaw ewaluacyjny EasyMxPRO™v7 for STELLARIS® ARM®, wraz z mikroprocesorem TM4C123GH6PGE produkcji Texas Instruments z rdzeniem Cortex-M4. Zestaw przedstawiony został na rys. 3.10. Obecnie cena zestawu w sklepie internetowym TME wynosi 207 PLN, gdzie jeszcze 3 miesiące temu, to jest w październiku 2014 roku wynosiła 625 PLN. Zestaw to bogato wyposażona platforma obsługująca 270 mikrokontrolerów rodziny ARM z rdzeniami Cortex-M3/M4 [6]. Zawiera wiele różnorodnych modułów, w tym między innymi kolorowy dotykowy ekran LCD, kodek mp3, czytnik kart pamięci, USB, Ethernet, CAN, brzęczyk, szereg diod i mikrołączników podłączonych do portów I/O. Wejścia/wyjścia podzielone są na 8 grup (A…I) po 8 linii I/O. W ramach każdej z grup dostępne jest złącze z rastrem 2,54 mm do wyprowadzenia linii I/O na zewnątrz zestawu, trójstanowe przełączniki do załączania rezystorów podciągających, 8 sztuk diod LED sygnalizujących stan danej linii oraz mikrołączniki,

które po naciśnięciu mogą wymuszać na linii stan niski bądź wysoki. Brzęczyk piezoelektryczny znajdujący się w zestawie podłączony jest do portu PA6 mikrokontrolera. Płyta EasyMxPRO posiada też dwa złącza USB, do których przyłączone są konwertery USB-UART, a następnie do interfejsów UART mikrokontrolera. Posiada także złącze USB DEVICE dostosowane do wtyczki typu B, służące do komunikacji z komputerem PC.

Rys. 2.7. Zdjęcie zestawu ewaluacyjnego EasyMxPRO™v7 for STELLARIS® ARM®

Mając na uwadze złącza USB należy jeszcze wspomnieć o wyposażeniu zestawu w złącze USB HOST dla wtyczki typu A. Do złącza tego można podłączyć np. myszkę czy klawiaturę i obsługiwać w mikrokontrolerze. Płyta ewaluacyjna posiada także złącze standardowe Ethernet typu RJ-45 podłączone bezpośrednio do mikrokontrolera. Oprócz tego w zestawie znajdują się wejścia/wyjścia audio gdyż płytę wyposażono w koder/dekoder VS1053 zintegrowany z przetwornikami A/C i C/A. EasyMxPRO zawiera także gniazdo kart mikroSD, gdzie poprzez magistralę SPI podłączone jest bezpośrednio z mikrokontrolerem. Najbardziej widocznym (i zapewne najdroższym elementem) jest kolorowy wyświetlacz LCD TFT o rozdzielczości 320x240 pikseli. Wyświetlacz dodatkowo wyposażony jest w rezystancyjną matryce dotykową wyposażoną w kontroler oraz regulowane za pomocą PWM podświetlenie. Na płycie znajdują się także dżojstik oraz miejsce na podłączenie czujników temperatury: DS1820 z cyfrową magistralą 1-wire i analogowym LM35. W zestawie zamontowane są także dwie pamięci: Flash o pojemności 8 Mb oraz EEPROM o pojemności 1024 bajtów. Ostatnim elementem jest potencjometr, którego za pomocą zworek można przyłączyć do jednego z pięciu wejść analogowych mikrokontrolera.
Sam zaś mikrokontroler TM4C123GH6PGE charakteryzuje się cechami:

- rdzeń Cortex-M4,
- 256 kB pamięci Flash,
- 32 kB pamięci SRAM,
- 2 kB pamięci EEPROM,
- jednostka zmiennoprzecinkowa (FPU),
- 8xUART,
- 4xSPI,
- 6xI^2C,
- 2xCAN,
- USB,
- 12xTimer,
- 2x 12-bitowy przetwornik A/C 1MSPS,
- 32 kanałowy interfejs DMA,
- sterownik przerwań NVIC.

Charakteryzują go dobre parametry wystarczające do zastosowań embedded w warunkach domowych i przemysłowych.

Zestaw EasyMxPRO™v7 for STELLARIS® ARM® pomimo tak bogatego wyposażenia został uznany przez autora niniejszej pracy magisterskiej, jako nienadający się do zastosowań w laboratorium Podstaw Techniki Mikroprocesorowej Politechniki Warszawskiej z kilku powodów. Po pierwsze, nie zawiera kilku rodzajów klawiatur ani wyświetlaczy: siedmiosegmentowego i 2x16 znaków LCD tak jak w zestawie DSM-51. Po drugie urządzenie przestaje być w sprzedaży w internetowym sklepie TME (największy polski sklep internetowy z szeroko rozumianą elektroniką). Po trzecie koszt rzędu 207 PLN za urządzenie jest relatywnie wysoki.

2.7.2. Raspberry Pi

Urządzeniem, które dość mocno wpłynęło na rynek zestawów uruchomieniowych jest platforma komputerowa Raspberry Pi stworzona przez Raspberry Pi Foundation. Gdy pierwsze egzemplarze pojawiały się w sklepie internetowym Farnell, należało się zapisywać w kilkutygodniowych kolejkach w celu uzyskania możliwości zakupu. O popularności może świadczyć też fakt, że po wpisaniu hasła „Raspberry Pi" w serwisie YouTube pojawia się ponad 430 tysięcy wyników. Premiera urządzenia odbyła się początkiem 2012 roku.

Rys. 2.8. Zdjęcie komputera Raspberry Pi model B2, z procesorem z rdzeniem Cortex-A7

Komputer z założenia miał posiadać przystępną cenę stąd też nie jest wyposażony w szereg peryferii pożądanych w typowych zestawach ewaluacyjnych jednak obecność portów USB, Ethernet, HDMI oraz czytnika kart SD, na której można wgrać obraz jednego z kilku dostępnych systemów operacyjnych sprawiła, że taka platforma dostępna za 100-200 PLN cieszy się duża popularnością. Zdjęcie komputera Raspberry Pi model B2 został przedstawiony na rys. 3.11. Pierwotnie, w pierwszych modelach (model A, B, B+ oraz Zero) platforma była wyposażona w procesor rodziny ARM11, a dokładnie mikroprocesor z rdzeniem ARM1176JZF-S. Trzy lata później, bo na wiosnę 2015 roku prowadzona na rynek została wersja B2 posiadająca dużo wydajniejszy procesor z rodziny ARM Cortex-A7. Programowanie najczęściej odbywa się w ten sposób, że na karcie SD nagrywa się obraz jednego z dostępnych systemów operacyjnych (np. Raspbian – specjalna dystrybucja systemu Linux Debian), a programy wykorzystujące zasoby procesora, linie I/O pisze się jako programy uruchamiane pod kontrolą systemu operacyjnego wykorzystując odpowiednie biblioteki czy moduły jądra.

Platforma z uwagi jednak na inna filozofię pracy niż na laboratorium Podstaw Techniki Mikroprocesorowej, brak odpowiedniego wyposażenia w standardzie oraz

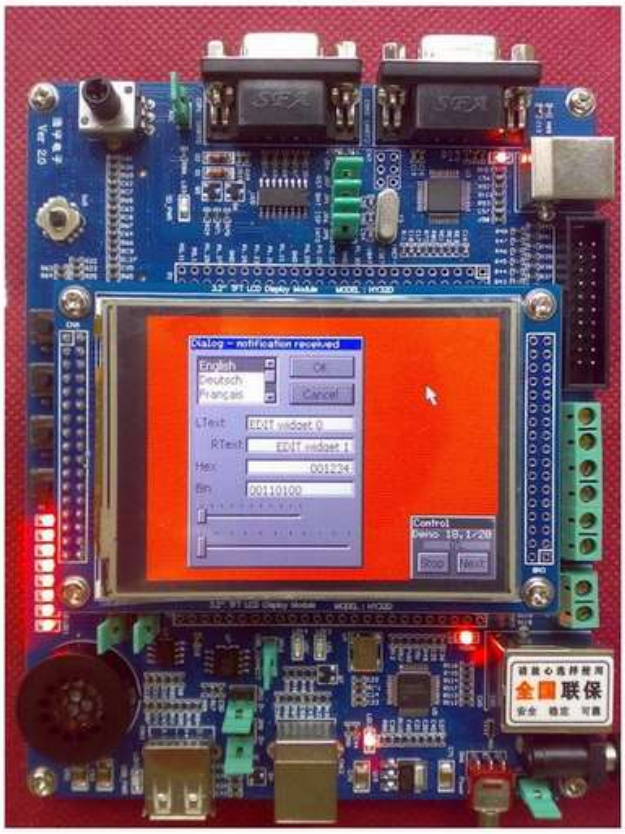
procesor ARM inny niż z rodziny Cortex-M nie jest rozważany jako platforma dydaktyczna zastępująca DSM-51. Przyczynił się on jednak do wielkiej popularyzacji procesorów z rdzeniami ARM wśród studentów i młodzieży i należało o nim wspomnieć.

2.7.3. Pozostałe zestawy ewaluacyjne dla Cortex-M3/M4

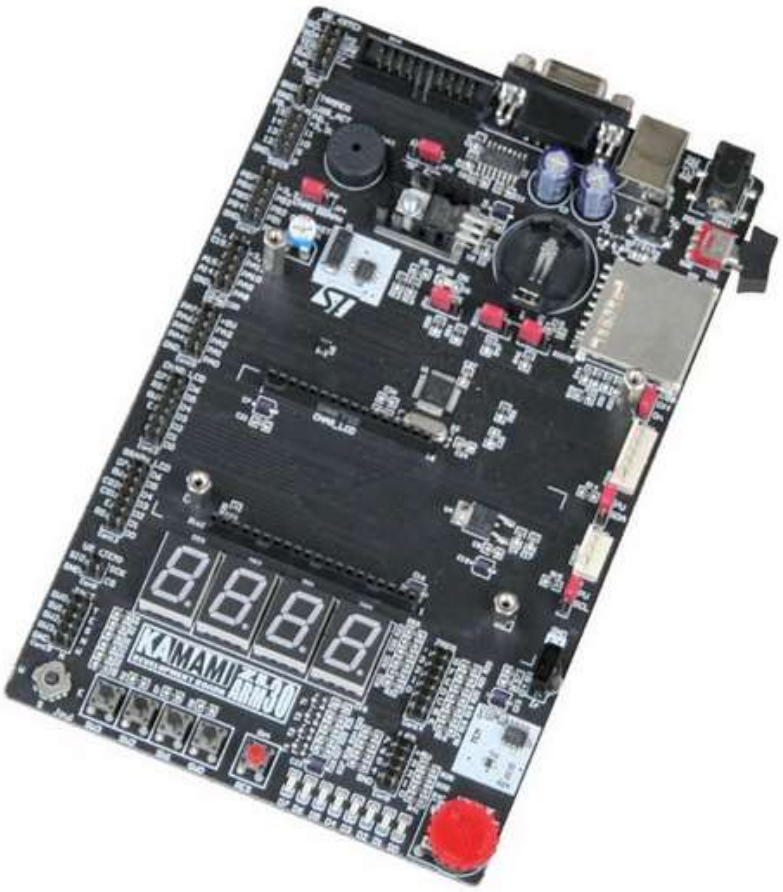
Na rynku dostępne są różne rodzaje zestawów ewaluacyjnych dla mikrokontrolerów z rdzeniem Cortex-M3 lub Cortex-M4. Kilka wybranych zostało przedstawione na rysunkach rys. 3.12 do rys. 3.15. Widać, że żaden z nich podobnie jak zestawy EasyMxPRO™v7 for STELLARIS® ARM® lub Raspberry Pi nie posiadają odpowiedniego wyposażenia pod kątem dydaktycznym. System DSM-51 został zaprojektowany z myślą o nauce programowania, poznawania mikroprocesorów i urządzeń peryferyjnych natomiast obecne na rynku zestawy ewaluacyjne nakierowane są na prezentacje możliwości danego mikrokontrolera i to dla profesjonalistów, a przynajmniej dla osób, które znają już podstawy mikrokontrolerów.

Rys. 2.9. Zestaw ewaluacyjny STM32429I-EVAL produkcji STMicroelectronic z procesorem STM32F429 z rdzeniem ARM®Cortex™-M4, w sklepie KAMAMI w cenie 1962 PLN brutto

Rys. 2.10. Zestaw uruchomieniowy Blueboard-LPC1768 COMBO produkcji NGX z procesorem LPC1768 (Cortex-M3)



Rys. 2.11. Zestaw uruchomieniowy EB-LPC1768_01 z mikrokontrolerem LPC1768 firmy NXP (Cortex-M3)



Rys. 2.12. Zestaw uruchomieniowy ZL30ARM dla mikrokontrolerów STM32F103 (Cortex-M3)

Zaniechano w związku z tym dalszego poszukiwania zestawu ewaluacyjnego, który możliwościami dorównałby systemowi DSM-51 i przystąpiono do poszukiwań rozwiązania alternatywnego – to znaczy wymiana jedynie procesora w systemie DSM-51 na mikrokontroler z rdzeniem Cortex-M3/M4.

3. Budowa układu ARM Cortex

3.1. Budowa mikrokontrolera STM32F100RBT6B

Mikrokontroler STM32F100RBT6B z rdzeniem ARM Cortex-M3 produkowany jest przez firmę STMicroelectronics na licencji udzielonej przez ARM Holdings. Oznaczenie mikrokontrolera należy rozumieć w następujący sposób:

- **STM32** – rodzina 32-bitowych mikrokontrolerów z rdzeniami ARM,
- **F** – mikrokontroler ogólnego przeznaczenia,
- **100** – podrodzina 100 (oznaczane w dokumentacji też, jako **F1**),
- **R** – obudowa z 64 wyprowadzeniami,
- **B** – 128 kB pamięci Flash,
- **T** – obudowa LQFP,
- **6** – temperatura stosowania: -40..+85 °C,
- **B** – wewnętrzne oznaczenie producenta.

Rodzina 100 (F1) jest rodziną mikrokontrolerów z rdzeniem Cortex-M3 o średniej wydajności. Poglądowa charakterystyka wydajności i wyposażenia w rdzenie Cortex przedstawia rys. 4.12. Sama rodzina F1 również podzielona jest na kilka typów mikrokontrolerów, są to: F100, F101, F102, F103, F105 i F107 różniące się między sobą maksymalną częstotliwością taktowania, ilością dostępnej pamięci Flash i RAM, oraz niektórymi peryferiami. Mikrokontrolery różnych typów są ze sobą kompatybilne pod względem wyprowadzeń i wyposażenia w standardowe peryferia. Generalnie w rodzinach STM32 panuje zasada, że wraz ze wzrostem numeru zwiększa się wydajność i wyposażenie mikrokontrolera. Seria F100 jest więc serią podstawową o maksymalnym taktowaniu 24 MHz. Osobną kwestią jest podział mikrokontrolerów produkcji STMicroelectronics ze względu na tak zwaną gęstość. Kryterium podziału stanowi ilość pamięci Flash. W mikrokontrolerach serii F100 podział ze względu na ilość pamięci Flash przebiega następująco:

- 16..32 kB – mikrokontrolery niskiej gęstości (ang. *low-density*),
- 64..128 kB – mikrokontrolery średniej gęstości (ang. *medium-density*),
- 256..512 kB – mikrokontrolery wysokiej gęstości (ang. *high-density*).

Pomiędzy mikrokontrolerami różnej gęstości zachodzą niewielkie różnice w budowie, ilości peryferii i dystrybucji sygnałów zegarowych wewnątrz układu. Zastosowany mikrokontroler posiada 128 kB pamięci Flash, zaliczany jest więc do mikrokontrolerów średniej gęstości, stąd tylko takich dotyczyć będzie dalszy opis zawarty w niniejszej pracy.

Wysoka wydajność		**STM32 F2** 120 MHz 150 DMIPS	**STM32 F4** 180 MHz 225 DMIPS	**STM32 F7** 216 MHz 462 DMIPS
Uniwersalne	**STM32 F0** 48 MHz 38 DMIPS	**STM32 F1** 72 MHz 61 DMIPS	**STM32 F3** 72 MHz 90 DMIPS	
Niskie zużycie energii	**STM32 L0** 32 MHz 26 DMIPS	**STM32 L1** 32 MHz 33 DMIPS	**STM32 L4** 80 MHz 100 DMIPS	
	Cortex-M0/M0+	Cortex-M3	Cortex-M4	Cortex-M7

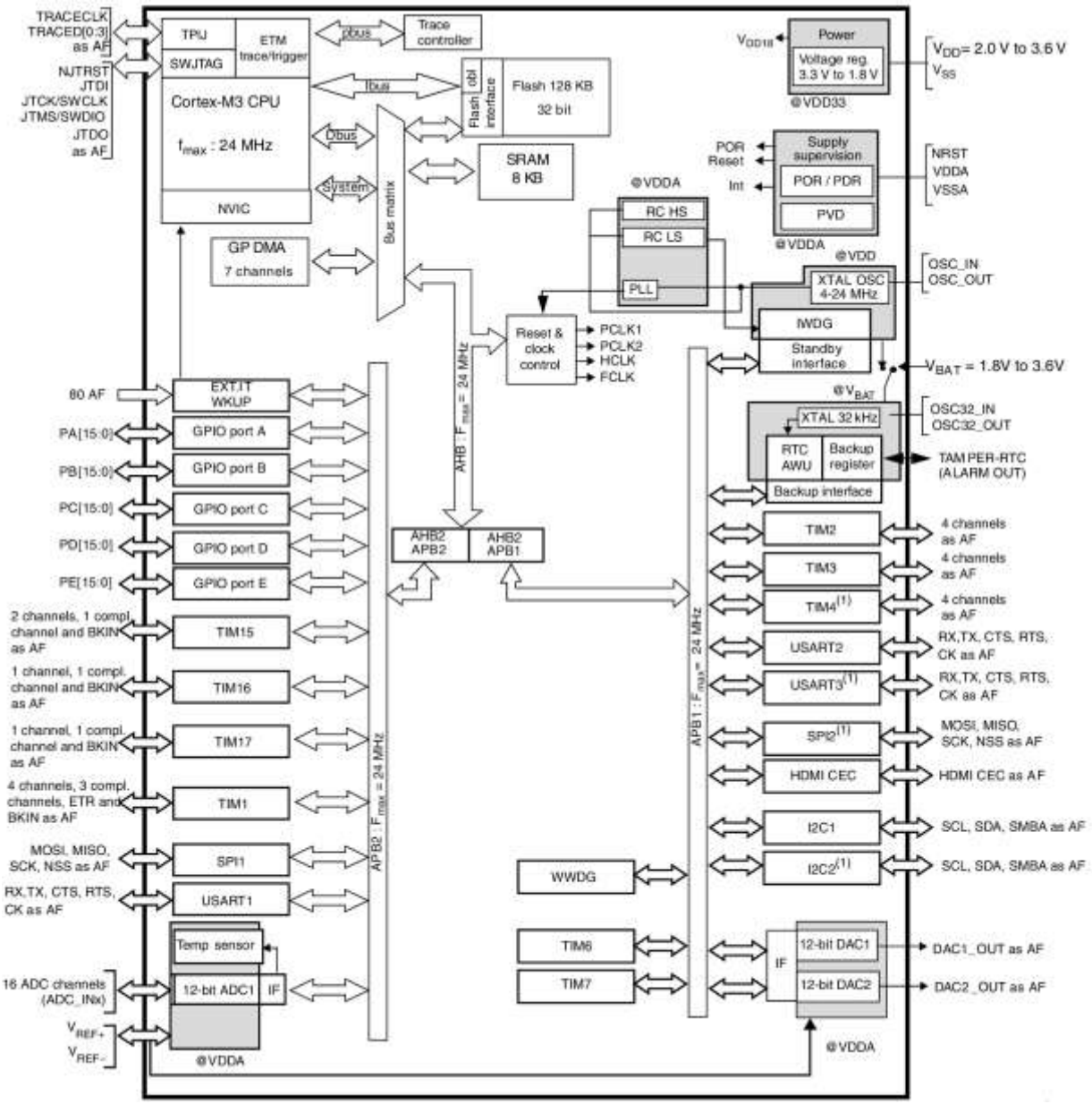
Rys. 3.1. Zestawienie rodzin STM32 z uwzględnieniem wydajności i wyposażenia w rdzeń Cortex

Omawiany mikrokontroler charakteryzuje się cechami zestawionymi w tab. 4.4.

Tablica 3.1. Najważniejsze parametry mikrokontrolera STM32F100RBT6B

Parametr	Wartość
Rdzeń	ARM® 32-bit Cortex®-M3
Architektura	ARMv7-M
Maksymalna częstotliwość taktowania	24 MHz
Wydajność	1.25 DMIPS/MHz
Napięcie zasilania	2,0…3,6 V (linie WE/WY tolerują 5 V)
Pamięć Flash	128 kB
Pamięć SRAM	8 kB
Interfejsy debugera	JTAG, SWD
Tryby oszczędzania energii	Sleep, Stop, Standby
DMA (ang. *Direct Memory Access*)	7-kanałowy
Przetwornik A/C	1 x 12-bitowy, 16-kanałowy
Przetwornik C/A	2 x 12-bitowy
Liczba Timerów	9
Interfejsy komunikacyjne	2xI^2C, 2xSPI, 3xUART, 1xHDMI-CEC
Pozostałe	Wewnętrzny generator RC, RTC, kontroler napięcia

Schemat blokowy mikrokontrolera STM32F100RBT6B został przedstawiony na rys. 4.13. Schemat ten prawdziwy jest dla mikrokontrolerów małej i średniej gęstości, z zastrzeżeniem, że w mikrokontrolerach serii STM32F100 małej gęstości nie występują elementy oznaczone symbolem (1). Mikrokontroler składa się z kilkudziesięciu elementów połączonych ze sobą różnego rodzaju magistralami wewnętrznymi. Najważniejszym elementem jest sam rdzeń Cortex-M3 wraz z sterownikiem przerwań NVIC (ang. *Nested vectored interrupt controller*) oraz interfejsami debugera (SW, JTAG) i jednostką debugującą ETM (ang. *Embedded Trace Macrocell*) służąca do rekonstrukcji przebiegu wykonywania programu. Rdzeń komunikuje się z pozostałymi elementami poprzez trzy magistrale, są to: Ibus (I-code bus), Dbus (D-code bus) oraz System (System bus). Fizycznie, wszystkie trzy są 32-bitowymi magistralami typu AMBA AHB-Lite (ang. *Advanced High-performance Bus*) natomiast przeznaczenie mają różne. W związku z tym, iż rdzenie Cortex-M3 są zaprojektowane w architekturze Harwardzkiej rozdzielone są pamięci programu oraz danych, a komunikacja z nimi odbywa się poprzez rozdzielne magistrale. Magistrala Ibus łączy bezpośrednio rdzeń mikrokontrolera z interfejsem pamięci Flash, stąd przeznaczona jest do odczytywania kolejnych instrukcji procesora zawartych w tej pamięci. Obsługuje adresy z zakresu od 0x00000000 do 0x1FFFFFFF. Do tej magistrali nie mają dostępu żadne interfejsy debugera. Druga z magistral – Dbus również obsługuje adresy z zakresu od 0x00000000 do 0x1FFFFFFF, ale łączy się z interfejsem pamięci Flash poprzez przełącznik Bus Matrix. Zadaniem tej magistrali jest dostęp do danych z przestrzeni pamięci programu oraz zapewnia dostęp mechanizmom debugera do pamięci Flash.



Rys. 3.2. Schemat blokowy budowy wewnętrznej mikrokontrolera STM32F100RBT6B

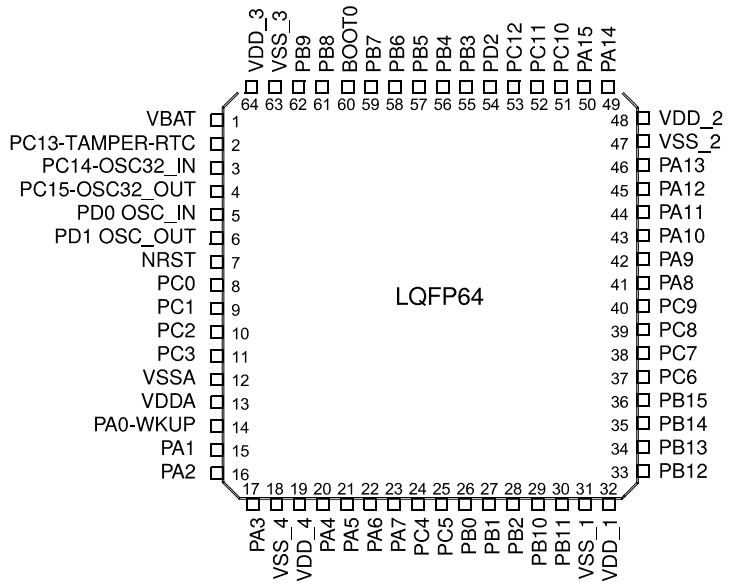
W samym interfejsie pamięci Flash priorytet obsługi posiada magistrala Dbus. Magistrala System bus zapewnia dostęp do pamięci SRAM oraz urządzeń peryferyjnych mikrokontrolera. Obsługuje adresy z zakresu od 0x20000000 do 0xDFFFFFFF oraz od 0xE0100000 do 0xFFFFFFFF. Zapewnia też dostęp mechanizmom debugera do elementów z tych przestrzeni adresowych.

Jak wspomniano, magistrale System i Dbus łączą się z pozostałymi elementami systemu poprzez przełącznik Bus Matrix. Element ten zapewnia wydajny i równomierny dostęp do elementów systemu jednostce centralnej i kontrolerowi DMA. Od strony Bus Matrix masterem są magistrale System, Dbus oraz DMAbus pochodzącą od kontrolera DMA. DMAbus fizycznie jest również 32-bitową magistralą AHB-Lite. Elementami typu slave są z punktu widzenia przełącznika Bus Matrix: interfejs pamięci Flash, pamięć SRAM, wewnętrzna magistrala AHB obsługująca dwa mostki (ang. *Bridges*) pomiędzy magistralami AHB i APB1/APB2 (ang. *Advanced Peripheral Bus*) oraz kontroler sygnału zerowania i zegarów. Bus Matrix używa algorytmu round-robin do przydzielania dostępu do poszczególnych zasobów. Jest to prosty algorytm przydzielania zasobów bez

uwzględniania priorytetów. W swoich strukturach zawiera także dekodery adresów w celu wyboru odpowiedniej magistrali do przesyłania danych.
Wszystkie układy peryferyjne takie jak Timery, interfejsy SPI, UART są przełączone do jednej z dwóch magistral APB1 lub APB2. Magistrale te są wolniejsze od AHB, zoptymalizowane pod kątem mniejszego poboru mocy ze względu na to, iż elementy przyłączone do tych magistral nie wymagają tak dużej prędkości przesyłu danych jak komunikacja pomiędzy rdzeniem Cortex-M3, a pamięciami. Wśród elementów peryferyjnych mikrokontrolera STM32F100RBT6B przyłączonych do magistral APB można wyróżnić następujące:

- GPIO A…D – kontrolery portów wejść/wyjść, porty A, B, C posiadają po 16 linii, natomiast port D w omawianym mikrokontrolerze tylko 3 linie, razem 51 linii,
- ADC1 – 12-bitowy, 16-kanałowy przetwornik analogowo-cyfrowy z sukcesywną aproksymacją o czasie przetwarzania 1,17 µs przy taktowaniu 24 MHz,
- DAC1, DAC2 – dwa 12-bitowe przetworniki cyfrowo-analogowe,
- TIM1 – 16-bitowy, zawansowany timer/licznik z możliwością zliczania w górę i w dół, z 4 niezależnymi kanałami,
- TIM2, TIM3, TIM4 – trzy 16-bitowe, 4-kanałowe timery/liczniki, każdy z możliwością zliczania w górę i w dół,
- TIM15 – 16-bitowy, 2-kanałowy timer/licznik ze zliczaniem tylko w górę,
- TIM16, TIM17 – dwa 16-bitowe, 1-kanałowe timery/liczniki ze zliczaniem tylko w górę,
- TIM6, TIM7 – dwa podstawowe, 16-bitowe timery ze zliczaniem tylko w górę,
- RTC – niezależny zegar czasu rzeczywistego, z możliwością taktowania z osobnego źródła zegarowego i podtrzymywania napięciem bateryjnym,
- IWDG – standardowy układ Watchdog, może być taktowany z osobnego źródła zegarowego,
- WWDG – okienkowy układ Watchdog, którego reset musi nastąpić w odpowiednim przedziale czasowym pomiędzy wystąpieniem poprzedniego resetu, taktowany z magistrali APB,
- SPI1, SPI2 – dwa interfejsy zewnętrznej magistrali SPI,
- I^2C1, I^2C2 – dwa interfejsy zewnętrznej magistrali I^2C,
- USART1, USART2, USART3 – trzy asynchroniczne interfejsy USART, mogące pracować w trybie full-duplex z prędkością transmisji do 4,5 Mbit/s, obsługują też standardy IrDA oraz LIN,
- HDMI CEC – interfejs wykorzystywany m.in. w urządzeniach RTV do sterowania jednym urządzeniem poprzez drugie,
- Reset & Clock Control – jednostka zarządzania sygnałami Reset oraz sygnałami zegarowymi,
- Voltage regulator – wewnętrzny układ zasilający dostarczający napięcia 1,8 V dla rdzenia, pamięci, układów peryferyjnych,
- Supply supervision – moduł kontroli napięcia zasilającego.

Magistrale APB1 i APB2 łączą się z główna magistralą systemową AHB poprzez dwa mostki: AHB/APB1 oraz AHB/APB2. Wszystkie wymienione elementy zamknięte są w 64-wyprowadzeniowej obudowie LQFP, której wygląd przedstawiony jest na rys. 4.14.



Rys. 3.3. Wyprowadzenia mikrokontrolera STM32F100RBT6B

Przedstawione na rys. 4.14. funkcje poszczególnych wyprowadzeń zmieniają się w zależności od załączonych wewnętrznych urządzeń peryferyjnych. Należy przez to rozumieć, że standardowo po resecie mikrokontrolera nie są włączone żadne urządzenia peryferyjne i wszystkie linie WE/WY są liniami ogólnego przeznaczenia (GPIO) przypisane do poszczególnych portów (A, B, C lub D) mikrokontrolera. Dopiero załączenie danego urządzenia peryferyjnego (np. interfejsu SPI1) powoduje, że niektóre wyprowadzenia przestają pełnić funkcję standardowych linii WE/WY, a przyjmują funkcje danego urządzenia peryferyjnego. W dokumentacji producenta nazwane jest to funkcją alternatywną (ang. *AF – Alternate Function*) danego wyprowadzenia. Oprócz powyższego należy mieć na uwadze, że niektóre z peryferii nie są na stałe przypisane do danego wyprowadzenia, to znaczy w zależności od konfiguracji mogą być dostępne na różnych portach mikrokontrolera. Producent w tym celu udostępnił rejestry do przeprzyporządkowywania (ang. *Remap*) wyprowadzeń urządzeń peryferyjnych na inne niż standardowe wyprowadzenia. Zestawienie wyprowadzeń mikrokontrolera wraz opisem funkcji zostało przedstawione w tab. 4.5.

Tablica 3.2. Zestawienie funkcji wyprowadzeń mikrokontrolera STM32F100RBT6B, opracowano wg [7]

Nr	Nazwa	Typ	Funkcja główna	Funkcja alternatywna	Remap
1	V_{BAT}	Z	V_{BAT}	-	-
2	PC13-TAMPER-RTC	WE/WY	PC13	TAMPER-RTC	-
3	PC14-OSC32_IN	WE/WY	PC14	OSC32_IN	-
4	PC15-OSC32_OUT	WE/WY	PC15	OSC32_OUT	-
5	OSC_IN	WE	OSC_IN	-	PD0
6	OSC_OUT	WY	OSC_OUT	-	PD1
7	NRST	WE/WY	NRST	-	-
8	PC0	WE/WY	PC0	ADC1_IN10	-
9	PC1	WE/WY	PC1	ADC1_IN11	-
10	PC2	WE/WY	PC2	ADC1_IN12	-
11	PC3	WE/WY	PC3	ADC1_IN13	-
12	V_{SSA}	Z	V_{SSA}	-	-
13	V_{DDA}	Z	V_{DDA}	-	-
14	PA0-WKUP	WE/WY	PA0	WKUP/USART2_CTS / ADC1_IN0 / TIM2_CH1_ETR	-
15	PA1	WE/WY	PA1	USART2_RTS /	-

				ADC1_IN1 / TIM2_CH2	
16	PA2	WE/WY	PA2	USART2_TX / ADC1_IN2 / TIM2_CH3 / TIM15_CH1	-
17	PA3	WE/WY	PA3	USART2_RX / ADC1_IN3 / TIM2_CH4 / TIM15_CH2	-
18	V_{SS_4}	Z	V_{SS_4}	-	-
19	V_{DD_4}	Z	V_{DD_4}	-	-
20	PA4	WE/WY	PA4	SPI1_NSS / ADC1_IN4 / USART2_CK / DAC1_OUT	-
21	PA5	WE/WY	PA5	SPI1_SCK / ADC1_IN5 / DAC2_OUT	-
22	PA6	WE/WY	PA6	SPI1_MISO / ADC1_IN6 / TIM3_CH1	TIM1_BKIN / TIM16_CH1
23	PA7	WE/WY	PA7	SPI1_MOSI / ADC1_IN7 / TIM3_CH2	TIM1_CH1N/ TIM17_CH1
24	PC4	WE/WY	PC4	ADC1_IN14	-
25	PC5	WE/WY	PC5	ADC1_IN15	-
26	PB0	WE/WY	PB0	ADC1_IN8 / TIM3_CH3	TIM1_CH2N
27	PB1	WE/WY	PB1	ADC1_IN8 / TIM3_CH4	TIM1_CH3N
28	PB2	WE/WY	PB2/BOOT1	-	-
29	PB10	WE/WY	PB10	I2C2_SCL / USART3_TX	TIM2_CH3 / HDMI_CEC
30	PB11	WE/WY	PB11	I2C2_SDA / USART3_RX	TIM2_CH4
31	V_{SS_1}	Z	V_{SS_1}	-	-
32	V_{DD_1}	Z	V_{DD_1}	-	-
33	PB12	WE/WY	PB12	SPI2_NSS / I2C2_SMBA / TIM1_BKIN / USART3_CK	-
34	PB13	WE/WY	PB13	SPI2_SCK / TIM1_CH1N / USART3_CTS	-
35	PB14	WE/WY	PB14	SPI2_MISO / TIM1_CH2N / USART3_RTS	TIM15_CH1
36	PB15	WE/WY	PB15	SPI2_MOSI / TIM1_CH3N / TIM15_CH1N	TIM15_CH2
37	PC6	WE/WY	PC6	-	TIM3_CH1
38	PC7	WE/WY	PC7	-	TIM3_CH2
39	PC8	WE/WY	PC8	-	TIM3_CH3
40	PC9	WE/WY	PC9	-	TIM3_CH4
41	PA8	WE/WY	PA8	USART1_CK / MCO / TIM1_CH1	-
42	PA9	WE/WY	PA9	USART1_TX / TIM1_CH2 / TIM15_BKIN	-
43	PA10	WE/WY	PA10	USART1_RX / TIM1_CH3 / TIM17_BKIN	-
44	PA11	WE/WY	PA11	USART1_CTS / TIM1_CH4	-
45	PA12	WE/WY	PA12	USART1_RTS / TIM1_ETR	-
46	PA13	WE/WY	JTMS-SWDIO	-	PA13
47	V_{SS_2}	Z	V_{SS_2}	-	-
48	V_{DD_2}	Z	V_{DD_2}	-	-
49	PA14	WE/WY	JTCK/SWCLK	-	PA14
50	PA15	WE/WY	JTDI	-	TIM2_CH1_

					ETR / PA15 / SPI1_NSS
51	PC10	WE/WY	PC10	-	USART3_TX
52	PC11	WE/WY	PC11	-	USART3_RX
53	PC12	WE/WY	PC12	-	USART3_CK
54	PD2	WE/WY	PD2	TIM3_ETR	-
55	PB3	WE/WY	JTDO	-	TIM2_CH2 / PB3 / TRACESWO/ SPI1_SCK
56	PB4	WE/WY	NJTRST	-	PB4 / TIM3_CH1 / SPI1_MISO
57	PB5	WE/WY	PB5	I2C1_SMBA / TIM16_BKIN	TIM3_CH2 / SPI1_MOSI
58	PB6	WE/WY	PB6	I2C1_SCL / TIM4_CH1 / TIM16_CH1N	USART1_TX
59	PB7	WE/WY	PB7	I2C1_SDA / TIM17_CH1N TIM4_CH2	USART1_RX
60	BOOT0	WE	BOOT0	-	-
61	PB8	WE/WY	PB8	TIM4_CH3 / TIM16_CH1 / CEC	I2C1_SCL
62	PB9	WE/WY	PB9	TIM4_CH4 / TIM17_CH1	I2C1_SDA
63	V_{SS_3}	Z	V_{SS_3}	-	-
64	V_{DD_3}	Z	V_{DD_3}	-	-

W kolumnie Typ podano rodzaj wyprowadzenia: Z – zasilające, WE – wejście, WY – wyjście, WE/WY – wejście/wyjście. Poszczególne wyprowadzenia są omówione w ramach opisu kolejnych układów peryferyjnych i urządzeń wewnętrznych mikrokontrolera.

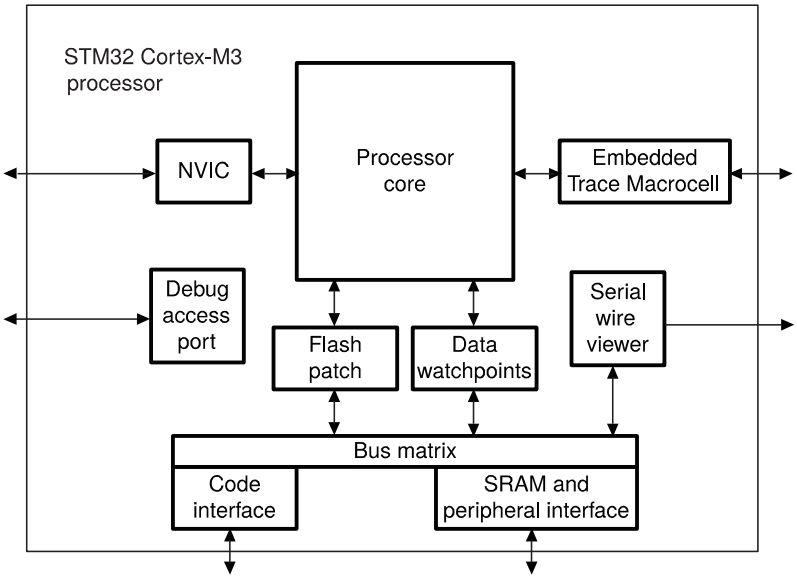
3.2. Rdzeń Cortex-M3

Rdzeń Cortex-M3 został zaprojektowany przez ARM Holdings implementując architekturę ARMv7-M również opracowaną przez tą samą firmę. Pewnego wyjaśnienia wymaga tu operowania nazwami rdzeni i ich architekturami, gdyż nie jest to jednoznaczne. W tab. 4.6 zestawiono przykładowe nazwy architektur, rdzeni zaprojektowanych przez ARM Holdings na ich podstawie oraz przykłady rdzeni zaprojektowanych przez firmy zewnętrzne. Tak jak zostało wspomniane wcześniej, firma ARM Holding sprzedaje zarówno projekty gotowych rdzeni jak i opisy architektur, na których podstawie zewnętrzne firmy mogą projektować swoje własne konstrukcje.
Architektura ARMv8-A jest architekturą 64-bitową, wszystkie pozostałe są 32-bitowe. Przy posługiwaniu się nomenklaturą należy zwrócić szczególną uwagę czy podaje się nazwę architektury czy rdzenia, gdyż w przeciwnym razie może dojść do nieporozumienia. Na przykład rdzeń ARM7 oparty jest na architekturze ARMv3, natomiast rdzeń ARM7TDMI już na architekturze ARMv4T, a rdzeń ARM7EJ na jeszcze kolejnej, to znaczy ARMv5TE. Podobnie rzecz ma się z rdzeniami opracowanymi przez firmę Apple, gdzie na przykład rdzeń Apple A9 jest oparty na architekturze ARMv8-A czyli tej samej, co rdzeń Cortex-A72, natomiast rdzeń Cortex-A9 zaprojektowany jest na architekturze ARMv7-A czyli tej, co Apple A6. Należy zatem z rozwagą operować hasłami na przykład „ARM 7” zaznaczając zawsze co autor wypowiedzi ma na myśli.

Tablica 3.3. Architektury ARM i rdzenie oparte na danych architekturach

Nazwa architektury	Rdzenie opracowane przez ARM	Rdzenie opracowane przez firmy zewnętrzne
ARMv3	ARM6, ARM7	-
ARMv4	ARM8	StrongARM
ARMv4T	ARM7TDMI, ARM9TDMI	-
ARMv5TE	ARM7EJ, ARM9E, ARM10E	XScale
ARMv6	ARM11	-
ARMv6-M	ARM Cortex-M0, ARM Cortex-M0+, ARM Cortex-M1	-
ARMv7-M	ARM Cortex-M3	-
ARMv7E-M	ARM Cortex-M4, ARM Cortex-M7	-
ARMv7-R	ARM Cortex-R4, ARM Cortex-R5, ARM Cortex-R7	-
ARMv7-A	ARM Cortex-A5, ARM Cortex-A7, ARM Cortex-A8, ARM Cortex-A9, ARM Cortex-A12, ARM Cortex-A15, ARM Cortex-A17	Apple A6/A6X
ARMv8-A	ARM Cortex-A35, ARM Cortex-A53, ARM Cortex-A57 ARM Cortex-A72	AMD K12, Apple A7/A8/A8X/A9/A9X, Nvidia Project Denver

Implementacja rdzenia Cortex-M3 zastosowana przez firmę STMicroelectronic została przedstawiona na rys. 4.15.



Rys. 3.4. Schemat implementacji rdzenia Cortex-M3 w mikrokontrolerze STM32F100RBT6B, opracowano wg [9]

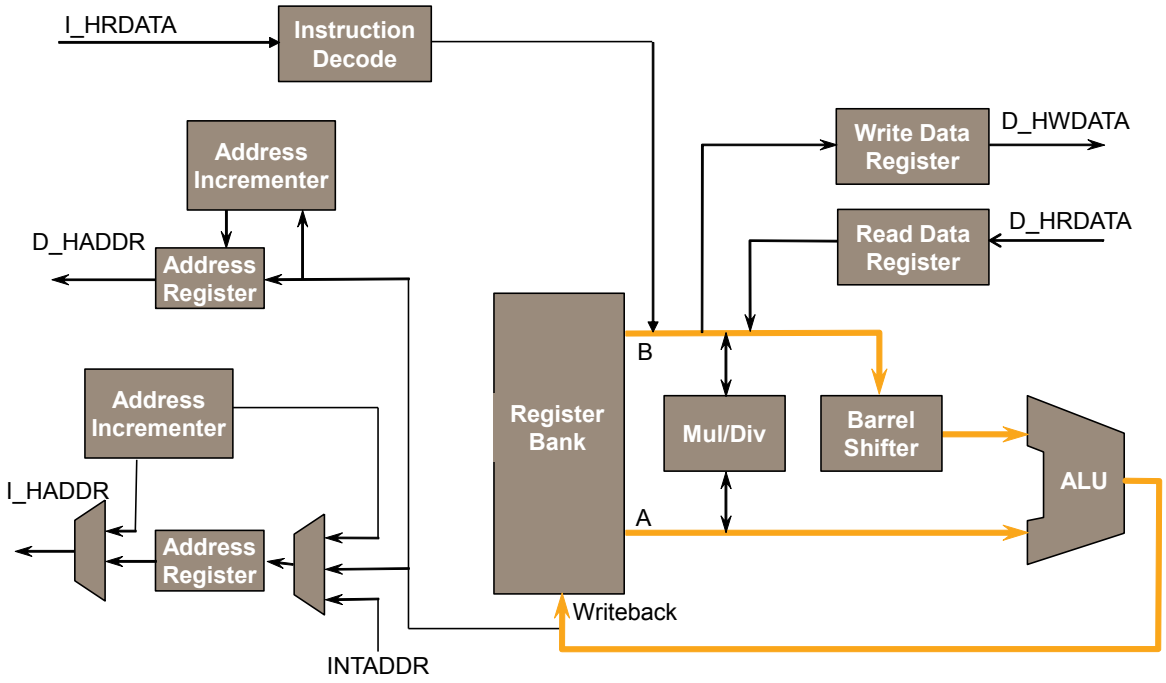
Głównym elementem jest jednostka CPU (Processor core), która łączy się z pozostałymi elementami za pomocą magistral. Wśród ukazanych elementów, kilka z nich zastosowanych jest ze względu na zwiększenie funkcjonalności debugera i nie wpływa na działanie rdzenia podczas normalnej jego pracy. Elementami tymi są:

- Flash patch – implementuje możliwość wprowadzania sprzętowych punktów przerwań oraz możliwość poprawy błędów w kodzie programu,
- Data watchpoints – implementuje możliwość wprowadzania sprzętowych punktów przerwań dla zmiennych w pamięci programu oraz wyzwalania zasobów,
- Serial wire viewer – umożliwia przesyłanie strumienia informacji generowanych w systemie oraz podglądu zmiennych poprzez pojedyncze wyprowadzenie mikrokontrolera,

- Debug access port – oznaczany też, jako AHB-AP jest portem dostępowym dla zewnętrznych magistral debugera: JTAG oraz SWD, który łączy się magistralą AHB do wewnętrznego Bus Matrix i uzyskuje dzięki temu dostęp do wszystkich rodzajów pamięci i rejestrów włączając w to rejestry procesora,
- Embedded Trace Macrocell – jednostka debugera umożliwiająca rekonstrukcję przebiegu programu, poprzez śledzenie kolejnych wykonywanych instrukcji.

Poprzez urządzenia Flash patch i Data watchpoint CPU łączy się z wewnętrznym elementem Bus Matrix, z którego to wyprowadzone są na zewnątrz rdzenia magistrale ICode bus, DCode bus, System bus. Oprócz powyższych w ramach rdzenia znajduje się też kontroler przerwań NVIC.

Jednostka CPU to 32-bitowe sekwencyjne urządzenie cyfrowe odpowiedzialne za pobieranie kolejnych kodów instrukcji z pamięci programu, danych z rejestrów podręcznych, pamięci SRAM lub urządzeń peryferyjnych, wykonywania odpowiednich obliczeń i zapisie wyników we wskazane miejsce. Składa się z 32-bitowego układu wykonawczego (ang. *datapath*), banku rejestrów 32-bitowych, i 32-bitowej magistrali pamięci. Układ wykonawczy rdzenia Cortex-M3 został przedstawiony na rys. 4.16.



Rys. 3.5. Układ wykonawczy rdzenia Cortex-M3, opracowano wg [10]

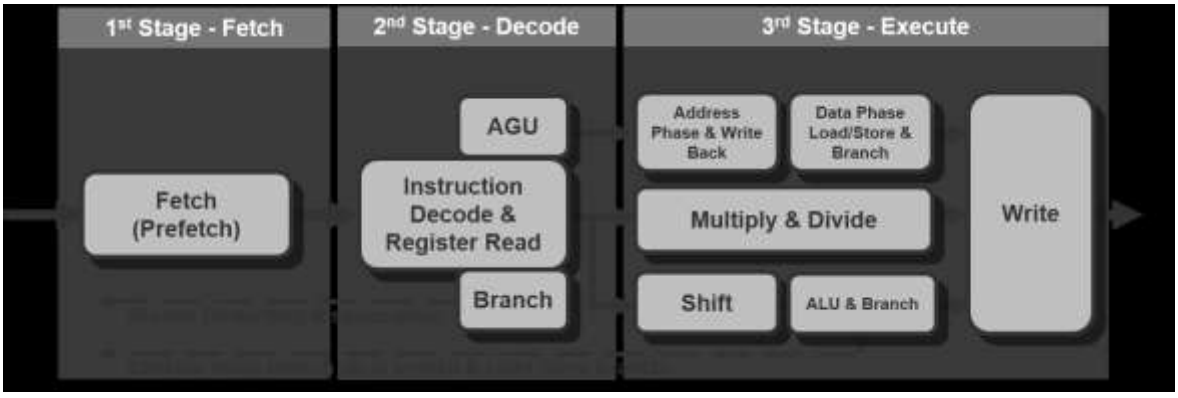
Instrukcje trafiają do Dekodera Rozkazów poprzez szynę danych magistrali odczytu instrukcji I_HRDATA. I_HADDR to szyna adresowa magistrali odczytu instrukcji. Magistrala danych składa się z trzech części – D_HADDR – szyny adresowej, D_HWDATA – szyny zapisu danych, D_HRDATA – szyny odczytu danych. Rdzeń zapewnia osobny odczyt i zapis danych oraz zapis danych do rejestrów i do magistrali danych. Na diagramie widoczne są tylko dwie magistrale: odczytu instrukcji oraz odczytu i zapisu danych. Podział na magistrale: ICode, DCode oraz System bus zachodzi poza samym CPU w przełączniku Bus Matrix zawartym w rdzeniu. Cortex-M3 wykonany jest w architekturze LDR-STR (ang. *Load Register – Store Register*), co oznacza, że operacje mogą być wykonywane tylko na danych zawartych w rejestrach znajdujących się w banku rejestrów. Przetwarzanie rozkazów odbywa się według 3-etapowego mechanizmu potokowego (ang. *pipeline*). Poprzez potokowanie przetwarzania należy rozumieć rozbicie wykonywania instrukcji na etapy. Każdy z etapów odbywa się w jednym cyklu

zegarowym, co w przypadku 3-etapowego przetwarzania oznacza, że jedna instrukcja wykonywana jest w ciągu trzech cyklów zegarowych. Należy mieć jednak na uwadze, że w przetwarzaniu potokowym w każdej chwili kolejny z etapów zajmuje się inną instrukcją, to znaczy etap I przetwarza instrukcję n-2, etap II przetwarza instrukcję n-1, a etap III przetwarza w tej chwili instrukcję n-tą. Powoduje to, że średnio przetwarzana jest jedna instrukcja w jednym cyklu zegarowym. Schemat ideowy 3-etapowego przetwarzania potokowego przedstawiony jest na rys. 4.17.

Cykl maszynowy	Etap I	Etap II	Etap III
m	Instrukcja n	Instrukcja n-1	Instrukcja n-2
m+1	Instrukcja n+1	Instrukcja n	Instrukcja n-1
m+2	Instrukcja n+2	Instrukcja n+1	Instrukcja n
m+3	Instrukcja n+3	Instrukcja n+2	Instrukcja n+1

Rys. 3.6. Schemat ideowy przetwarzania potokowego

Widać na nim, że w każdym cyklu zegarowym pracują wszystkie etapy przetwarzania potokowego, z tym, że każdy z nich zajmuje się inną instrukcją. Przetwarzanie potokowe ma na celu zmniejszenie ilości operacji (uproszczeniu) przeprowadzanych przez każdy z etapów w celu umożliwienia zastosowania szybszego taktowania. Dzięki takiemu rozwiązaniu możliwe stają się częstotliwości pracy sięgające setek megaherców.
Schemat blokowy przetwarzania potokowego w rdzeniach Cortex-M3 został przedstawiony na rys. 4.18.



Rys. 3.7. Potokowe przetwarzanie instrukcji w rdzeniach Cortex-M3, opracowano wg [10]

Rdzenie Cortex-M3 posiadają 3-stopniowe przetwarzanie potokowe podobne do zastosowanego w rdzeniach ARM7, z tą różnicą, że w każdym z etapów przeprowadzane jest więcej operacji w celu optymalizacji i zwiększenia wydajności dla mniejszych częstotliwości taktowania. Pierwszym z etapów jest pobieranie instrukcji (ang. *Fetch*). Jednostka, która jest za to odpowiedzialna posiada możliwość wstępnego pobierania instrukcji (ang. *Prefetch*) w ilości trzech instrukcji 32-bitowych lub sześciu instrukcji 16-bitowych lub ich kombinacji i umieszczenia ich w buforze. Oznacza to, że jednostka pobiera z wyprzedzeniem kilka instrukcji do przodu zanim jeszcze CPU stwierdzi, jaki numer instrukcji potrzebuje. Za określnie, spod którego adresu należy z wyprzedzeniem sięgnąć po instrukcje odpowiedzialny jest etap II. Jego zadaniem jest dekodowanie instrukcji otrzymanych z pierwszego stopnia przetwarzania potokowego oraz odczyt

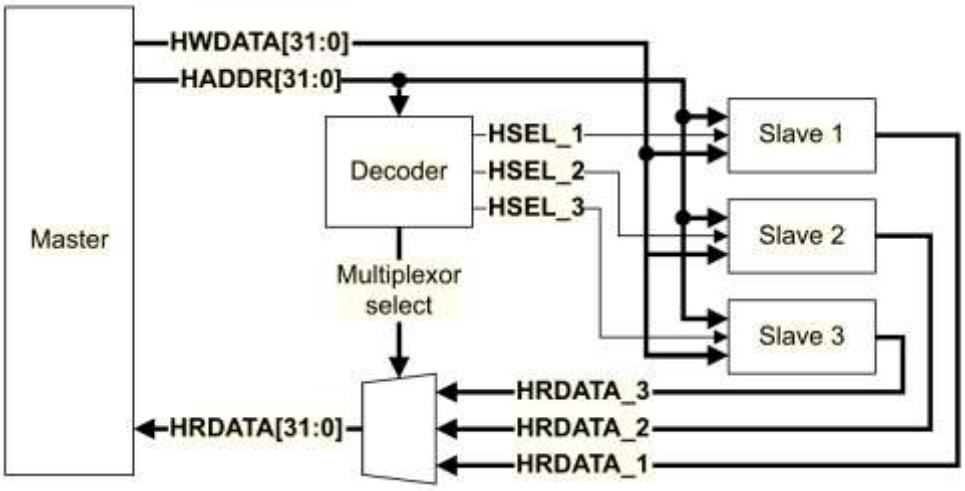
danych z potrzebnych rejestrów. Zawiera także jednostkę obliczania adresów instrukcji. Drugi etap nie wykonuje jednak instrukcji, więc w przypadku, gdy we fragmencie kodu nie ma skoków, to pobierane są z wyprzedzeniem po prostu kolejne instrukcje. W chwili, gdy została zdekodowana instrukcja skoku warunkowego to jednostka pobierania instrukcji pobiera instrukcje spod obu adresów, które mogą być wynikiem operacji (ang. *branch speculation*). Dzięki takim zabiegom CPU nie musi czekać na pobranie kolejnej instrukcji z pamięci programu (która jest wolniejsza niż sam CPU) niezależnie od wyniku wykonania rozkazu. Trzeci etap to rzeczywiste wykonanie instrukcji z wykorzystaniem jednostki arytmetyczno-logicznej (ang. *ALU*), osobnej jednostki mnożenia i dzielenia oraz szybkiego rejestru przesuwnego (Barrel Shifter) oraz zapis wyniku.
W trakcie wykonywania programu są oczywiście sytuacje, w których jednostka centralna ma za zadanie wykonanie fragmentu programu spod adresu całkiem innego niż wcześniej przewidywanego. Przykładem takiego zdarzenia może być skok niebezpośredni BX lub BLX. W takim przypadku bufor instrukcji jak i potoki są czyszczone z tego, co się w nich znajduje niezależnie od stopnia wykonania instrukcji i pobierana jest z pamięci programu instrukcja spod wskazanego adresu, a następnie przetwarzana potokowo według powyższego opisu. Powoduje to, że częściowe przetworzenie wcześniej przewidywanych operacji było zbędne, a cały potok potrzebuje, co najmniej trzech cykli zegara do wykonania pierwszej instrukcji spod adresu skoku. Jest to największa wada przetwarzania potokowego.

3.3. Wewnętrzne magistrale systemowe

Jak wspomniano wcześniej, rdzeń łączy się z pozostałymi elementami mikrokontrolera za pomocą trzech 32-bitowych magistral AHB-Lite: Ibus, Dbus oraz System. Magistrale Dbus oraz System wchodzą do przełącznika Bus Matrix, z którego wyprowadzone są dopiero magistrale AHB do dwóch mostków AHB/APB, pamięci Flash oraz pamięci SRAM. Do elementu Bus Matrix dochodzi również magistrala od strony kontrolera DMA. Za mostkami AHB/APB wyprowadzone są już magistrale APB do komunikacji z poszczególnymi urządzeniami peryferyjnymi.

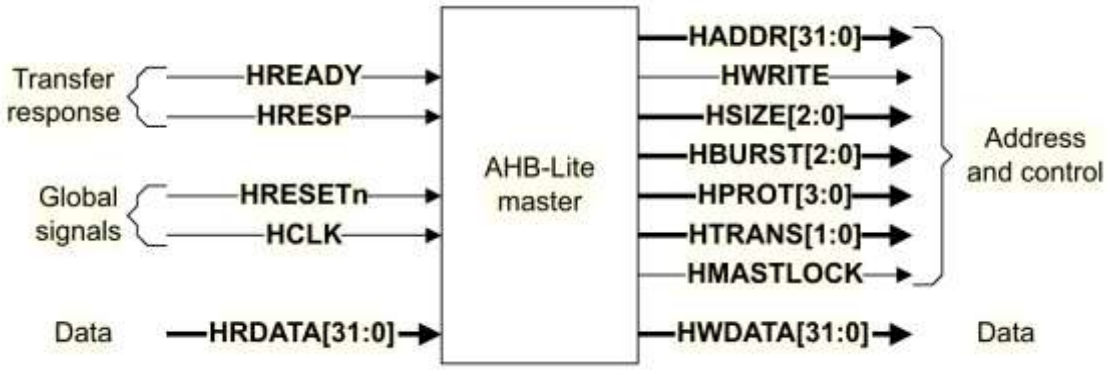
3.3.1. Magistrala AHB-Lite

Magistrala AHB jest szybką, synchroniczną magistralą danych z osobną 32-bitową szyną adresową, 32-bitową szyną danych do zapisu, 32-bitową szyną danych do odczytu oraz wieloma liniami kontrolnymi stosowaną w przypadku potrzeby uzyskania wysokiej przepustowości pomiędzy łączonymi elementami. Takiej przepustowości wymaga na przykład łączność jednostki centralnej CPU z pamięciami. Elementy przyłączone do magistrali są w topologii master-slave, to znaczy jedno z urządzeń jest urządzeniem nadrzędnym (ang. *master*), natomiast pozostałe (jedno lub więcej) są urządzeniami podrzędnymi (ang. *slave*). Zadaniem urządzenia typu master jest generowanie adresów i sygnałów kontrolnych, natomiast urządzenia slave czekają aż na magistrali pojawią się odpowiednie sygnały, na skutek których wykonują określone operacje (np. wystawiają dane na szynę danych spod określonego adresu). Przykład układu połączenia jednego urządzenia master i trzech urządzeń slave do jednej magistrali AHB-Lite został przedstawiony na rys. 4.19. Oprócz wymienionych wcześniej elementów widoczne są także dekoder adresów (Decoder) oraz Multiplekser danych. Zadaniem dekodera adresów jest generowanie sygnałów HSELx, czyli wyboru odpowiedniego układu slave spośród kilku dostępnych w zależności od adresu wystawionego przez urządzenie master.



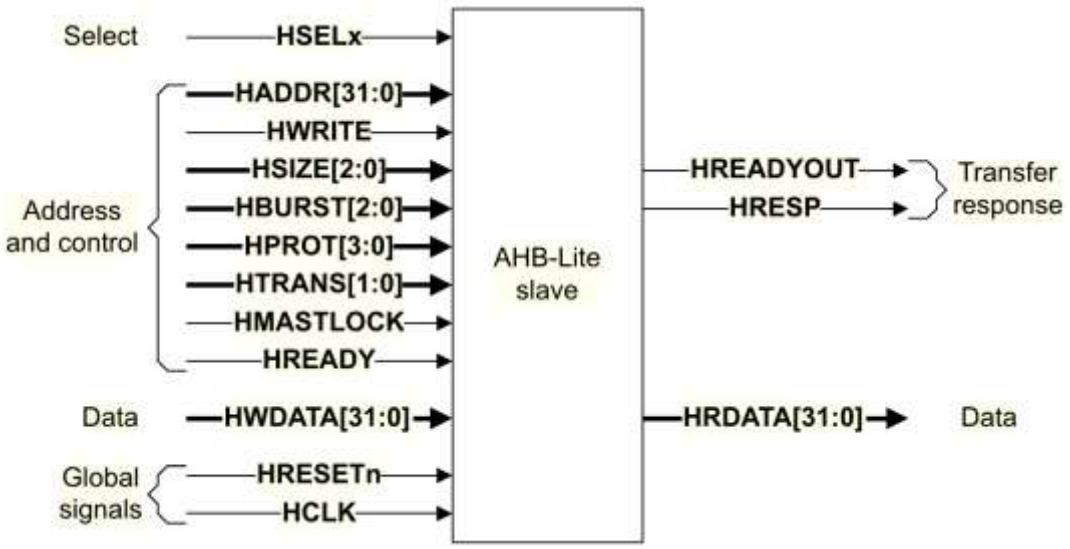
Rys. 3.8. Przykład wykorzystania magistrali AHB-Lite z trzema urządzeniami slave, opracowano wg [11]

Dekoder adresów spełnia tu analogiczną funkcję jak dekoder w Dydaktycznym Systemie Mikroprocesorowym DSM-51. Multiplekser danych wybiera natomiast jedno spośród kilku źródeł danych (jedno spośród kilku urządzeń slave) jako źródło danych na linii HRDATA[31:0] przyłączonej do urządzenia master. Urządzeniem sterującym całą komunikacją w magistrali AHB-Lite jest urządzenie master. Widok takiego urządzenia wraz z przyłączonymi szynami i liniami magistrali AHB został przedstawiony na rys. 4.20. Z prawej strony mastera zostały umieszczone szyny i linie, którymi to urządzenie steruje, czyli wymusza na nich stany wysokie lub niskie, natomiast z lewej strony zostały zobrazowane szyny i linie, których stan zależy od urządzeń slave (linie HREDY, HRESP oraz szyna HRDATA[31:0]) lub od układu zegarowego mikroprocesora (linie HCLK, HRESETn).



Rys. 3.9. Widok urządzenia AHB-Lite master, opracowano wg [11]

Grubszymi liniami oznaczono szyny (więcej niż jedna linia) w nawiasie kwadratowym oznaczając ilość linii w danej szynie. Zapis [31:0] należy rozumieć jako 32 linie o numerach od 0 do 31. Nazwy wszystkich sygnałów zaczynają się od litery „H” w celu zaznaczenia, że sygnał dotyczy magistrali AHB, dla odróżnienia sygnałów z magistrali APB, których nazwy zaczynają się od litery „P”. Na rys. 4.21 przedstawiono urządzenie typu AHB-Lite slave wraz z przyłączonymi liniami i szynami. Większość linii i szyn po lewej stronie pochodzi z urządzenia master, za wyjątkiem linii HSELx, która ma źródło w dekoderze adresów oraz linii HRESETn i HCLK, które pochodzą od układu zegarowego mikroprocesora.



Rys. 3.10. Widok urządzenia AHB-Lite slave, opracowano wg [11]

Urządzenie slave może jedynie wymuszać stan na liniach HREADYOUT i HRESP oraz na szynie danych HRDATA[31:0].

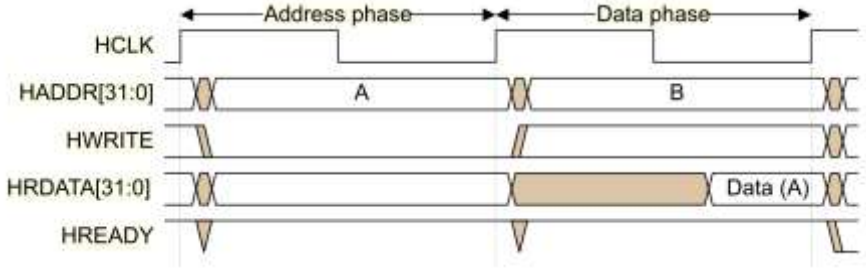
Opis poszczególnych linii i szyn magistrali AHB-Lite przedstawiono w tab. 4.7.

Tablica 3.4. Linie i szyny magistrali AMBA AHB-Lite

Nazwa	Źródło sygnału	Opis
HCLK	Układ zegarowy	Linia z sygnałem zegarowym dla operacji przeprowadzanych na magistrali. Wszystkie sygnały próbkowane są przy narastającym zboczu sygnału.
HRESETn	Układ resetu	Sygnał resetu dla wszystkich elementów podłączonych do magistrali AHB-Lite. Jedyny sygnał w magistrali, którego stanem aktywnym jest stan niski. Po otrzymaniu tego sygnału urządzenie master powinno wystawić linie i szyny na prawidłowych poziomach, a urządzenia slave wystawić stan wysoki na linii HREADYOUT.
HADDR[31:0]	Master	32-bitowa szyna adresowa. Adres wystawiony na szynie analizuje dekoder adresów i urządzenia slave.
HBURST[2:0]	Master	3-bitowa szyna informująca o rodzaju transferu ciągłego danych. Domyślnie jest to 000, co oznacza transfer pojedynczy – na przykład zapytanie o daną spod konkretnego adresu i otrzymanie jej wartości. Natomiast w innych stanach tych linii możliwe są transfery danych z kilku następujących po sobie adresów.
HMASTLOCK	Master	Ustawienie stanu wysokiego przez mastera na tej linii oznacza, że operacja na magistrali, która jest obecnie przeprowadzana nie może być wstrzymana, ani przerwana przez żaden inny transfer. Ma to znaczenie przy urządzeniach slave możliwych do sterowania przez kilka urządzeń master.
HPROT[3:0]	Master	4-bitowa szyna dodatkowych linii informujących o poziomach ochrony (ang. *protection*) operacji na magistrali. Sygnalizuje na przykład czy dana operacja pochodzi z instrukcji użytkownika, czy programu głównego, albo czy może być buforowana. Sygnały przeznaczone głównie dla modułów kontroli pamięci (MMU), którego w omawianym mikrokontrolerze nie ma.
HSIZE[2:0]	Master	3-bitowa szyna informująca o wielkości bloku danych odczytywanych spod jednego adresu. Typowo jest to jeden bajt (stan 000) ale mogą też być inne na przykład słowo (ang. *Word* – 32 bity) czy nawet 1024 bity.

HTRANS[1:0]	Master	2-bitowa szyna informująca o statusie obecnego transferu. Może przyjmować stan IDLE – stan przy braku transferu, BUSY – stan chwilowego wstrzymania transferu przez mastera pomiędzy danymi z transferu ciągłego, NOSEQ – transfer pojedynczy lub pierwsza dana z transferu ciągłego, SEQ – stan przy transferze kolejnych danych z transferu ciągłego.
HWDATA[31:0]	Master	32-bitowa szyna danych przepływających w kierunku od urządzenia master do urządzenia slave. Szerokość 32-bitów nie jest tu narzucona na sztywno i producenci mogą ją dostosować do swoich potrzeb (od 8-bitów do 1024-bitów), przy czym ARM zaleca szerokość minimum 32 bity i nieprzekraczającą 256 bitów. W omawianym mikrokontrolerze magistrala danych ma szerokość 32 bitów.
HWRITE	Master	Linia sygnalizuje kierunek transferu danych. Jeżeli jest w stanie wysokim, to następuje zapis danych do urządzenia slave, a jeżeli stan jest niski, to następuje odczyt danych z urządzenia slave.
HRDATA[31:0]	Slave	32-bitowa szyna danych przepływających w kierunku od urządzenia slave do urządzenia master, poprzez ewentualny element Multiplexor. Podobnie jak w szynie HWDATA[31:0] szerokość 32-bitów nie jest tu narzucona.
HREADYOUT	Slave	Stan wysoki na tej linii sygnalizuje zakończenie transferu na magistrali.
HRESP	Slave	Linia informująca o stanie transferu od strony urządzenia slave. Stan niski niesie informację OK, stan wysoki niesie informację ERROR.
HSEL_x	Dekoder adresów	Indywidualna linia kontrolna każdego urządzenia slave doprowadzona od dekodera adresów. Stan na tej linii zależy od ustawienia dekodera adresów i stanu szyny adresowej. Dane urządzenie slave jest aktywne (odbiera lub nadaje dane) tylko wówczas, gdy otrzymuje stan wysoki na tej linii.

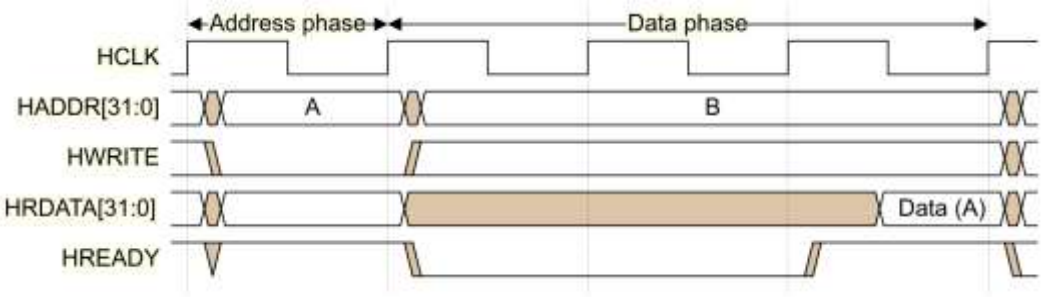
Transfery danych przeprowadzanych na magistrali odbywają się w cyklach synchronicznie z sygnałem zegara HCLK. Cykle podzielone są na fazę adresu i fazę danych. Domyślnie, jeżeli urządzenia master i slave są odpowiednio szybkie to każda z faz trwa jeden okres sygnału zegarowego HCLK. Stan poszczególnych linii i szyn podczas odczytu danej z urządzenia slave został przedstawiony na rys. 4.22.



Rys. 3.11. Stan linii i szyn podczas odczytu danych z urządzenia slave, opracowano wg [11]

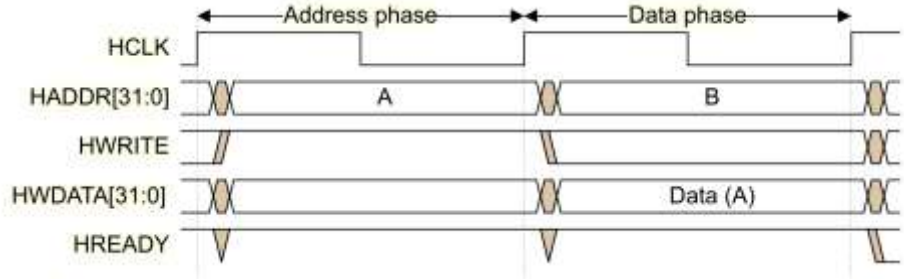
Na rysunku tym można zauważyć, że po pojawieniu się zbocza narastającego na sygnale HCLK urządzenie master wystawia na szynę adresową adres A oraz ustawia stan niski na linii HWRITE, co odpowiada odczytowi danych z urządzenia slave. Pojawienie się kolejnego zbocza narastającego HCLK powoduje zatrzaśnięcie adresu i stanu linii HWRITE w urządzeniu slave, co wymusza początek wystawiania określonej adresem danej na szynę danych HRDATA[31:0]. W tym czasie mikrokontroler wystawia już kolejny adres na szynę adresową i steruje linią HWRITE w zależności od kierunku przepływu danej. Pojawienie się kolejnego zbocza narastającego na linii sygnału zegarowego HCLK powoduje zatrzaśnięcie w urządzeniu master danych wystawionych

przez urządzenie slave na szynie HRDATA[31:0] oraz zatrzaśnięcie adresu i stanu linii HRDATA w urządzeniu slave tak żeby przygotowywało kolejną daną do wystawienia. Przedstawiony odczyt danych przebiegał bez opóźnień, to znaczy urządzenie slave jest na tyle szybkie, że zdążyło przygotować i wystawić na szynę HRDATA[31:0] wymaganą przez urządzenie master daną w ciągu jednego okresu sygnału zegarowego. Czasami jednak może się tak zdarzyć, że urządzenie potrzebuje więcej czasu, na przykład w przypadku wolniejszych pamięci Flash. Wówczas urządzenie slave utrzymuje w stanie niskim linię HREADY tak długo, aż dane nie będą gotowe do odczytu dla urządzenia master. Przykład takiego zdarzenia został przedstawiony na rys. 4.23.



Rys. 3.12. Stan linii i szyn podczas odczytu danych z urządzenia slave z opóźnieniem wystawienia danych, opracowano wg [11]

Widać, że w tym przypadku urządzenie slave potrzebowało dodatkowo dwóch okresów sygnału HCLK w celu wystawienia danych na szynę HREADY[31:0]. W tym czasie urządzenie master cały czas na szynie adresowej wystawia adres kolejnej danej, którą chce odczytać z urządzenia slave. Analogicznie do przedstawionych przebiegów przy odczycie następuje zapis danych do urządzenia slave. Przebiegi podczas takiego procesu zostały przedstawione na rys. 4.24.

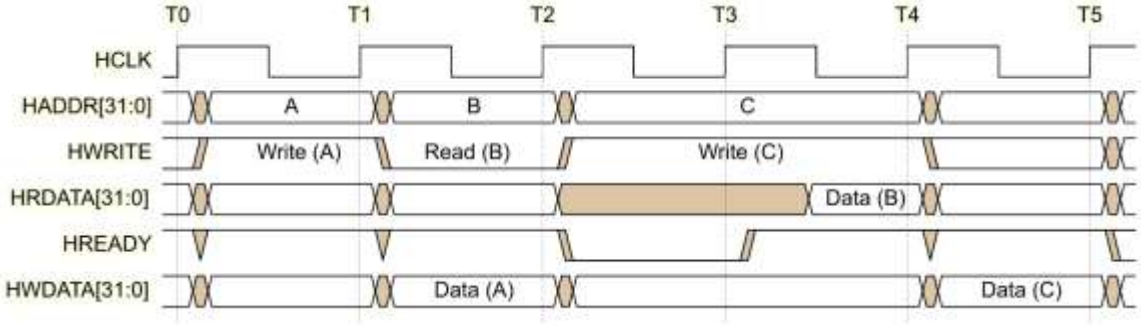


Rys. 3.13. Stan linii i szyn podczas zapisu do urządzenia slave, opracowano wg [11]

Podobnie jak w przypadku odczytu, urządzenie master wystawia adres na szynie adresowej HADDR[31:0] oraz inaczej niż poprzednio ustawia stan wysoki na linii HWRITE, co oznacza zapis do urządzenia slave oraz wystawia daną do zapisu na szynie HWDATA[31:0]. Jest to inna szyna niż w przypadku odczytu. Wraz z pojawianiem się kolejnego zbocza narastającego na sygnale zegarowym urządzenie slave zapisuje stan tych linii i przygotowuje się do zapisu danej z szyny HWDATA[31:0] pod odpowiedni adres. Koniec zapisu sygnalizuje stanem wysokim na linii HREADY. Urządzenie slave w tym przypadku również może wymagać dłuższego czasu na zapis niż jeden okres zegara. W takim przypadku utrzymuje stan niski na linii HREADY tak długo, aż nie zakończy poprawnie zapisu danej. Urządzenie master w tym czasie zobowiązane jest do ciągłego utrzymywania zapisywanej danej na szynie HWDATA[31:0].

Operacje zapisu i odczytu mogą przebiegać naprzemiennie z lub bez opóźnień tworząc ciąg operacji odczyt/zapis. Przykład takiego transferu został zobrazowany na rys. 4.25. Na rysunku tym widać, że zapis danej A odbył się bez opóźnień (ang. *zero wait states*), odczyt

danej B również bez opóźnień natomiast zapis danej C z opóźnieniem o jeden okres sygnału zegarowego (ang. *one wait states*).

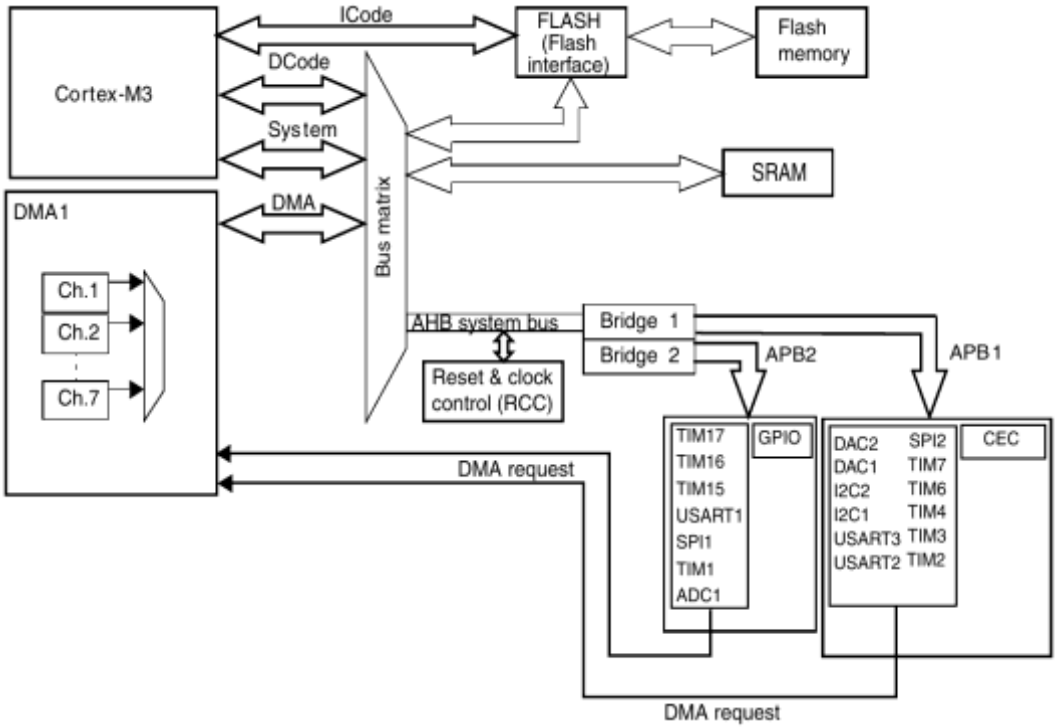


Rys. 3.14. Stan linii i szyn podczas pracy z zapisami i odczytami z urządzenia slave, opracowano wg [11]

Dzięki rozdzieleniu szyny danych do zapisu i szyny danych do odczytu oraz w przypadku szybkich urządzeń wykonywaniu operacji w ciągu jednego okresu sygnału zegarowego, możliwe jest utrzymanie ciągłego strumienia danych pomiędzy urządzeniami master i slave. W mikrokontrolerze STM32F100RBT6B maksymalne taktowanie magistrali AHB wynosi 24 MHz, biorąc pod uwagę, iż szerokość szyny danych wynosi 32 bity, otrzymujemy maksymalną przepustowość magistrali na poziomie 768 Mb/s. Potwierdza to zapewnienia twórcy standardu o wysokiej wydajności tego typu rozwiązania.

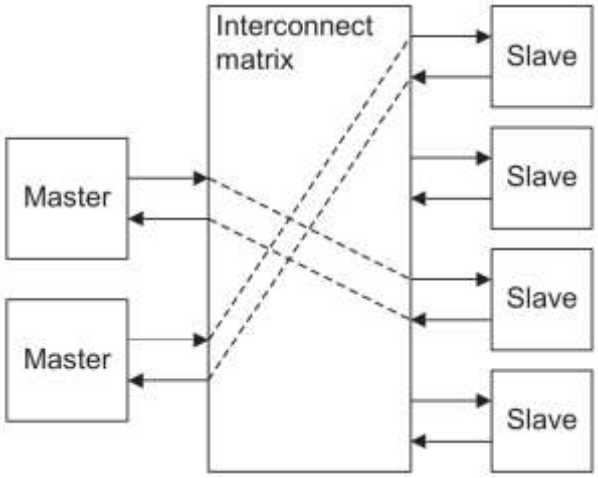
3.3.2. Wielopoziomowy przełącznik magistral AHB-Lite – Bus Matrix

W mikrokontrolerze STM32F100RBT6B elementem łączącym kilka magistral AHB-Lite ze sobą jest Bus Matrix. Jest to rodzaj przełącznika pomiędzy kilkoma urządzeniami typu master i kilkoma typu slave w celu umożliwiania komunikacji każdy-z-każdym przy jednoczesnym zapewnieniu bezkolizyjności transmisji i równemu dostępowi każdemu urządzeniu typu master. Schemat magistral wewnętrznych w omawianym mikrokontrolerze został przedstawiony na rys. 4.26.



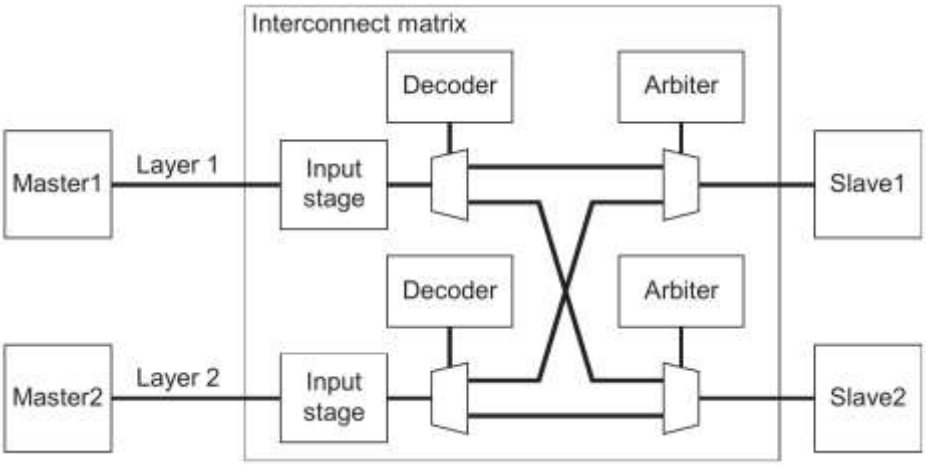
Rys. 3.15. Schemat magistral wewnętrznych w mikrokontrolerze STM32F100RB6T, opracowano wg [12]

Do przełącznika Bus Matrix przyłączone są trzy magistrale AHB-Lite, dla których Bus Matrix jest urządzeniem slave, są to: DCode, System oraz DMA, oraz trzy magistrale, dla których Bus Matrix jest urządzeniem typu master, są to: magistrala pamięci Flash, magistrala pamięci SRAM oraz AHB system bus. Koncepcja działania takiego przełącznika została przedstawiona na rys. 4.27.



Rys. 3.16. Koncepcja działania przełącznika magistral AHB-Lite Bus Matrix, opracowano wg [13]

Przełącznik Bus Matrix ma zadanie umożliwiać urządzeniom master przyłączonych do różnych magistrali AHB-Lite komunikację z urządzeniami slave przyłączonymi również do kilku rożnych magistral. W przypadku, gdy urządzenia master prowadzą wymianę danych z różnymi urządzeniami slave, to przełącznik umożliwia przeprowadzania wszystkich tych transmisji równolegle. W przypadku, gdy oba urządzenia master chcą wymieniać dane z tym samym urządzeniem slave, to przełącznik przeprowadza arbitraż, który z masterów ma mieć dostęp do docelowego urządzenia. Algorytm rozstrzygający nie uwzględnia priorytetów. Na rys. 4.28. przedstawiono sposób implementacji prostego urządzenia Bus Matrix z dwiema magistralami master i dwiema magistralami slave.



Rys. 3.17. Implementacja przełącznika Bus Matrix dla dwóch magistral master i dwóch magistral slave AHB-Lite, opracowano wg [13]

Każda z magistral od urządzenia typu master posiada na wejściu do Bus Matrix element o nazwie Input Stage. Jest on urządzeniem zatrzaskującym adres, dane i stan linii kontrolnych przychodzących z urządzenia master podczas oczekiwana na możliwość wymiany danych z danym urządzeniem slave. Podobnie, każda z magistral wejściowych ma też swój własny dekoder adresów, służący do wyboru danej magistrali wyjściowej w zależności od adresu na szynie adresowej. Dekoder steruje demultiplekserem, który

posiada tyle wyjść ile przyłączonych jest magistral z urządzeniami typu slave. Każde z wyjść magistral typu slave jest zakończone multiplekserem, który łączy wszystkie magistrale od demultiplekserów i wystawia tylko jedną już przyłączoną do magistrali slave. Pracą multipleksera steruje właśnie arbiter, który na podstawie algorytmu podejmuje decyzję, które z urządzeń master powinno aktualnie prowadzić transmisję na magistrali slave.

Przełączniki Bus Matrix to nie tylko domena mikrokontrolerów z rdzeniami ARM. Występują także w mikrokontrolerach opartych na innych architekturach i z innymi systemami magistral danych jednak zasada ich działania jest analogiczna.

3.3.3. Magistrala APB

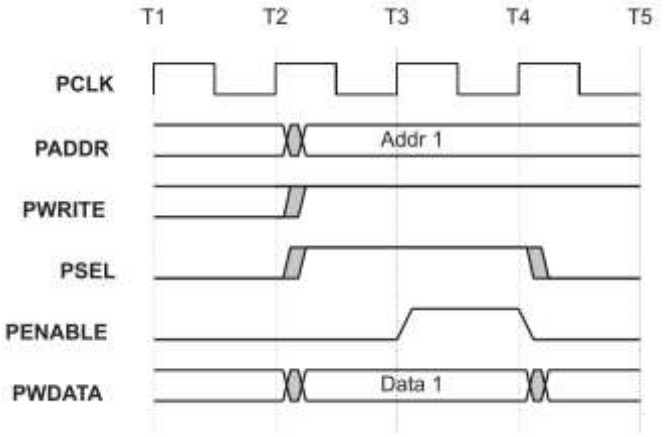
Magistrala APB również jest synchroniczną równoległa magistralą danych, z 32-bitową szyną adresową, 32-bitową szyną danych do zapisu i 32-bitową szyną danych do odczytu. Nie posiada jednak tak wielu trybów komunikacji i zarazem linii sterujących jak w przypadku magistrali AHB. Od magistrali AHB ma mniejszą przepustowość i przez to jest bardziej energooszczędna. W mikrokontrolerze przyłączana jest do urządzeń slave takich jak: interfejsy komunikacyjne: UART, SPI, timery czy przetworniki A/C. Urządzenia te nie wymagają wysokoprzepustowych łącz, w związku z czym zastosowanie w tym miejscu wolniejszej i energooszczędnej magistrali jest uzasadnione. W tab. 4.8. przedstawiono opis poszczególnych linii.

Tablica 3.5. Linie i szyny magistrali AMBA APB

Nazwa	Źródło sygnału	Opis
PCLK	Układ zegarowy	Linia z sygnałem zegarowym dla operacji przeprowadzanych na magistrali. Wszystkie sygnały próbkowane są przy narastającym zboczu sygnału.
PRESETn	Układ resetu	Sygnał resetu dla wszystkich elementów podłączonych do magistrali APB. Jedyny sygnał w magistrali, którego stanem aktywnym jest stan niski.
PADDR[31:0]	Mostek AHB/APB	32-bitowa szyna adresowa.
PSEL_x	Mostek AHB/APB	Indywidualna linia kontrolna każdego urządzenia slave doprowadzona od dekodera adresów zawartego w mostku AHB/APB. Stan na tej linii zależy od ustawienia dekodera adresów i stanu szyny adresowej. Dane urządzenie slave jest aktywne (odbiera lub nadaje dane) tylko wówczas, gdy otrzymuje stan wysoki na tej linii.
PENABLE	Mostek AHB/APB	Sygnał strobujący używany w każdej komunikacji po magistrali APB. Stan wysoki informuje o drugim cyklu transmisji danych na magistrali APB.
PWRITE	Mostek AHB/APB	Linia sygnalizuje kierunek transferu danych. Jeżeli jest w stanie wysokim, to następuje zapis danych do urządzenia slave, a jeżeli stan jest niski, to następuje odczyt danych z urządzenia slave.
PWDATA[31:0]	Mostek AHB/APB	32-bitowa szyna danych przepływających w kierunku od mostka AHB/APB do urządzenia slave.
PRDATA[31:0]	Slave	32-bitowa szyna danych przepływających w kierunku od urządzenia slave do mostka AHB/APB.

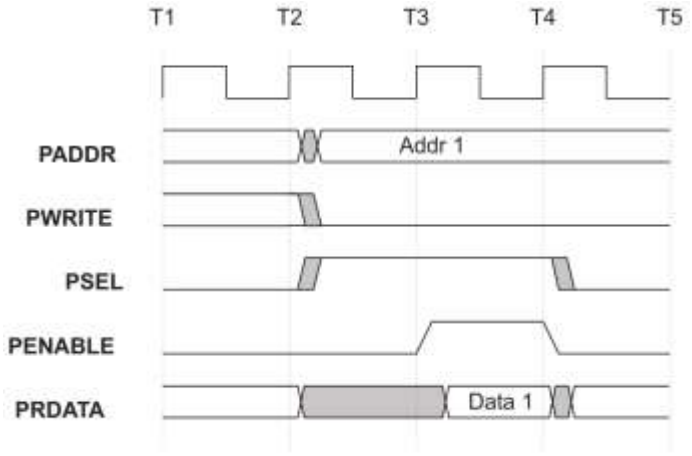
Magistrala może znajdować się w jednym z trzech stanów: IDLE, SETUP oraz ENABLE. Stanem domyślnym jest IDLE – brak przesyłu danych, wszystkie linie PSELx oraz linia PENABLE są w stanie niskim. Kiedy wymagany jest przesyła danych magistrala przechodzi ze stanu IDLE do stanu SETUP. Wówczas wystawiany jest adres na szynie adresowej i generowany jest odpowiedni sygnał PSELx, a także jeżeli następuje zapis danych, to wystawiane są dane na szynie PWDATA[31:0]. Stan taki trwa przez jeden okres sygnału zegarowego PCLK. Po stanie SETUP zawsze kolejnym stanem jest stan

ENABLE. Pojawia się wówczas stan wysoki na linii PENABLE, w którym to momencie wybrane sygnałem PSELx urządzenie slave zatrzaskuje adres na szynie adresowej i ewentualną daną na szynie danych. Po kolejnym okresie sygnału zegarowego, magistrala przechodzi albo w stan IDLE, jeżeli żaden transfer danych nie jest wymagany albo stan SETUP, jeżeli będzie występowała kolejna komunikacja na magistrali. Transfery danych na magistrali AMBA APB odbywają się synchronicznie do przebiegu sygnału zegarowego PCLK. Proces zapisu danej do urządzenia slave został zaprezentowany na rys. 4.29.



Rys. 3.18. Stan linii i szyn podczas zapisu do urządzenia slave, opracowano wg [15]

Na rysunku widać, że przejście ze stanu IDLE do stanu SETUP (pojawienie się stanu wysokiego na linii PSEL) następuje po wystąpieniu zbocza narastającego sygnału zegarowego PCLK. W tej samej chwili czasu ustawiany jest również stan wysoki na linii PWRITE sygnalizujący zapis danej do urządzenie slave oraz adres na szynie adresowej i dana na szynie danych do zapisu PWDATA. Kolejne zbocze narastające na linii PCLK powoduje pojawienie się stanu wysokiego na linii PENABLE i przejście do stanu ENABLE. Urządzenie slave ma czas do następnego zbocza narastającego na poprawne zapisanie danych. Po wystąpieniu kolejnego zobaczą narastającego mostek wymusza stan niski na liniach PSEL i PENABLE, a magistrala przechodzi w stan IDLE. W wersji 2.0 AMBA APB urządzenie slave musi zrealizować proces w określonym czasie. Nie ma możliwości wydłużenia procesu tak jak to było w przypadku magistrali AHB. Natomiast nowszy standard AMBA APB w wersji 3.0 przewiduje dodatkową linię sygnałową PREADY, która pełni analogiczną funkcję do linii HREADY w magistrali AHB-Lite, czyli informuje mostek, że urządzenie slave nie wykonało jeszcze operacji i należy poczekać z kolejnymi instrukcjami.

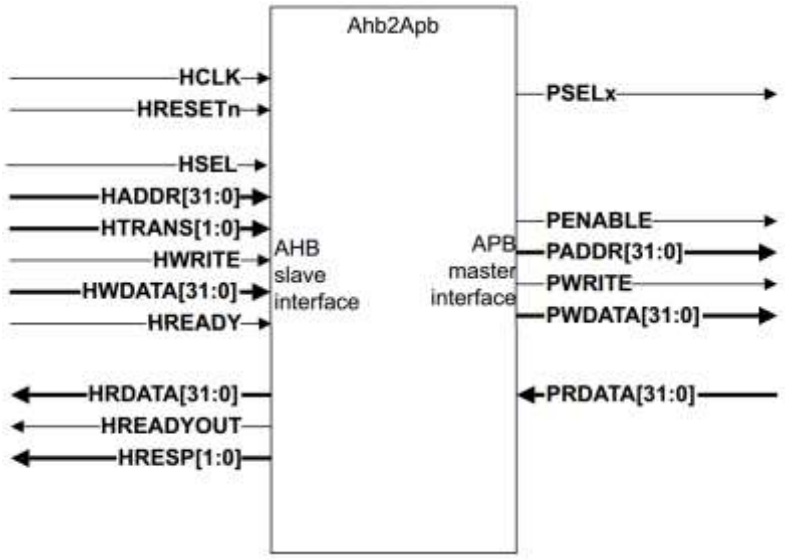


Rys. 3.19. Stan linii i szyn podczas odczytu danej z urządzenia slave, opracowano wg [15]

W dokumentacjach mikrokontrolera STM32F100RBT6B nie jest jednak ujęte, z którą wersją specyfikacji są zgodne zastosowane w mikrokontrolerze magistrale danych.
Proces odczytu danej z urządzenia slave został przedstawiony na rys. 4.30. Podobnie jak w przypadku poprzednim, mostek wystawia adres na szynę adresową przy równoczesnym wymuszeniu stanu wysokiego na linii PSEL i stanu niskiego na linii PWRITE po zboczu narastającym zegara. Stan magistrali zmienia się zatem z IDLE na SETUP, a urządzenie slave czeka na pojawienie się sygnału strobującego zatrzaskującego u niego adres na szynie adresowej. Sygnał ten pojawia się wraz z kolejnym zboczem narastającym na linii PCLK. Magistrala przechodzi w stan ENABLE, a urządzenie slave wystawia dane na szynę PRDATA. Kolejne zbocze narastające powoduje zatrzaśnięcie danych w mostku i przejście magistrali w stan IDLE. Tak samo jak poprzednio, urządzenie nie ma możliwości wydłużenia cyklu. Jest to możliwe dopiero od wersji 3.0 AMBA APB.

3.3.1. Mostek AHB/APB

Jak można zaobserwować na rys. 4.26. za przełącznikiem Bus Matrix dwie z magistral łączą się bezpośrednio z urządzeniami AHB-Lite slave, jakimi są: pamięć SRAM oraz interfejs pamięci Flash. Do trzeciej magistrali (AHB system bus) przyłączone są trzy urządzenia slave: Kontroler sygnału zegarowego i resetu oraz dwa mostki pomiędzy magistralami AHB i APB o nazwach Bridge1 i Bridge2.
Widok mostka z przyłączonymi sygnałami został zaprezentowany na rys. 4.31.



Rys. 3.20. Mostek AHB/APB z widocznymi sygnałami wejściowymi i wyjściowymi, opracowano wg [14]

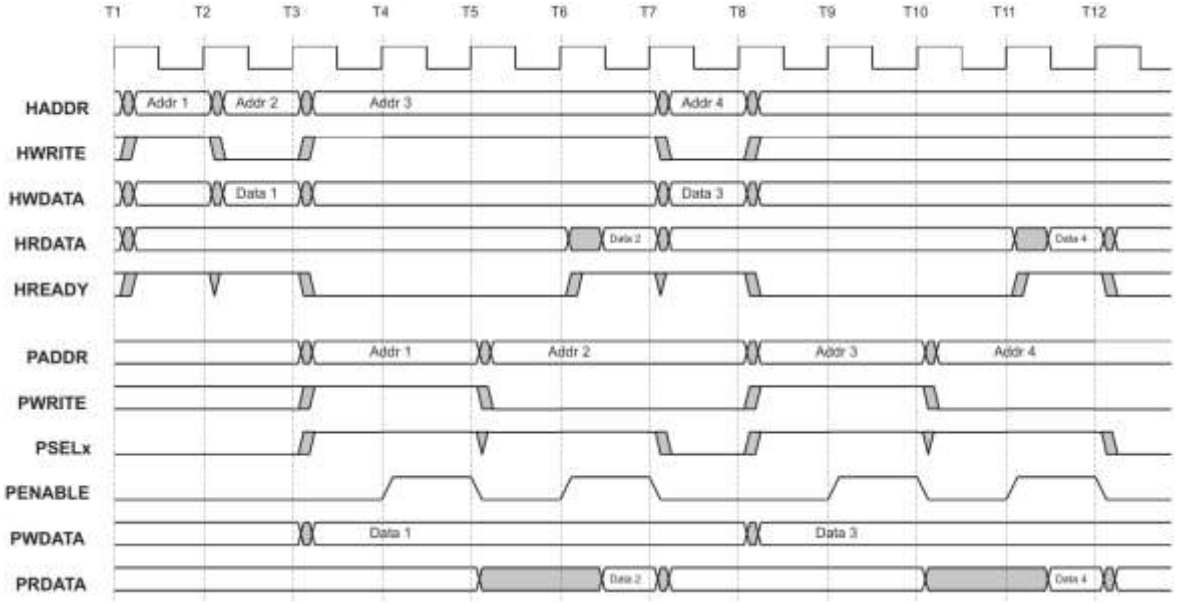
Na przedstawionym rysunku z lewej strony znajdują się sygnały pochodzące od kontrolera sygnału zegarowego i resetu (sygnały HCLK i HRESETn) oraz linie i szyny przyłączone do urządzenia AHB-Lite master (wszystkie pozostałe). Z prawej strony mostka widoczne są linie i szyny tworzące magistralę APB. Mostek wykonuje następujące zadania:

- zatrzaskuje adres na szynie adresowej i utrzymuje tak długo jak trwa transmisja z urządzeniem slave,
- dekoduje adres na szynie adresowej i na tej podstawie generuje sygnały PSELx – indywidualny sygnał wyboru każdego układu typu slave. W czasie wymiany danych master-slave tylko jeden z sygnałów PSEL jest w stanie wysokim (aktywnym),
- kieruje dane z magistrali AHB do urządzeń przyłączonych do magistrali APB podczas operacji zapisu,

- kieruje dane z urządzeń przyłączonych do magistrali APB do urządzenia master magistrali AHB podczas operacji odczytu,
- generuje sygnał strobujący PENABLE.

Można powiedzieć, że dekoder adresów zawarty we strukturach mostka pełni podobną funkcję jak układ U7 typu 74HC138 w systemie DSM-51. W systemie DSM-51 układ U7 otrzymuje na wejście jeden sygnał wyboru urządzeń peryferyjnych CSIO (ang. *Chip Select Input/Output*) pochodzący z dekodera adresów i na podstawie niektórych linii szyny adresowej generuje sygnały wyboru poszczególnych układów peryferyjnych na przykład CSAD (ang. *Chip Select Analog to Digital converter*) czy CSDA (ang. *Chip Select Digital to Analog converter*), czyli sygnały wyboru jednego spośród kilku układów peryferyjnych, z którym będzie prowadzona transmisja danych. W przypadku dekodera adresów zawartego w mostku AHB/APB sytuacja jest podobna, to znaczy dekoder otrzymuje jeden sygnał wyboru HSEL pochodzący z dekodera adresów zawartego w przełączniku Bus Matrix i na podstawie adresu pojawiającego się na szynie danych ustawia w stan wysoki jeden z wielu sygnałów wyboru układów peryferyjnych PSELx, gdzie x zależy od układu, który został wybrany.

Na rys. 4.32. przedstawiono stany poszczególnych linii i szyn magistral AHB i APB po jednej i drugiej stronie mostka podczas wymiany danych pomiędzy urządzeniem nadrzędnym, na przykład rdzeniem procesora, a urządzeniem slave takim jak na przykład przetwornik A/C.



Rys. 3.21. Stany linii i szyn magistral AHB-Lite i APB podczas operacji zapisu i odczytu do urządzenia slave, opracowano wg [15]

Na rysunku tym widać, że cała wymiana danych składa się z następujących po sobie rodzajów operacji:

1. Zapis danej Data 1 do urządzenia slave pod adres Addr 1.
2. Odczyt danej Data 2 z urządzenia slave spod adresu Addr 2.
3. Zapis danej Data 3 do urządzenia slave pod adres Addr 3.
4. Odczyt danej Data 4 z urządzenia slave spod adresu Addr 4.

Kolejne zbocza narastające sygnału HCLK i zarazem PCLK oznaczono jako T1, T2, aż do T12. Na rynku widać, że po wystąpieniu zbocza T1 urządzenie master AHB-Lite wystawia adres Addr 1 oraz stan wysoki na linii HWRITE, co sygnalizuje, że będzie następował zapis do urządzenia slave. Po wystąpieniu zbocza T2, master wystawia na szynę HWDATA danę Data 1 do zapisu pod adres Addr 1 i od razu wystawia adres Addr 2 i linię

HWRITE w celu zasygnalizowania, że będzie następował odczyt danej spod adresu Addr 2. Do czasu wystąpienia zbocza T3 mostek nie zmieniał jeszcze stanu żadnej linii ani po stronie AHB ani APB. Magistrala APB do momentu wystąpienia zbocza T3 była w stanie IDLE. Po wystąpieniu zbocza T3 magistrala APB przechodzi w stan SETUP, to znaczy wystawia adres Addr 1 na szynie PADDR, daną Data 1 na szynie PWDATA oraz ustawia linię PWRITE w stan wysoki, co sygnalizuje urządzeniu slave, że będzie następował zapis danej. Mostek od strony magistrali AHB-Lite ustawia linię HREADY w stan niski, w celu zasygnalizowania, że operacja zapisu do urządzenia jeszcze nie została zakończona i mostek nie jest gotowy do przyjęcia kolejnego zadania. Po wystąpieniu tego samego zbocza T3 urządzenie AHB master wystawia na szynie adresowej adres Addr 3 i linię HWRITE w stan wysoki, co oznacza, że będzie zapisywał pod adres Addr 3. Ta faza adresu w cyklu AHB-Lite będzie jednak trwała dłużej niż jeden okres zegara HCLK ze względu na to, że linia HREADY jest w stanie niskim, co nie pozwala wystawić urządzeniu master kolejnego adresu. Kolejne zbocze narastające T4 na linie zegarowej powoduje przejście magistrali APB ze stanu SETUP do stanu ENABLE, co równoznaczne jest z podaniem stanu wysokiego przez mostek na linię PENABLE. Stan magistrali AHB nie zmienia się. Urządzenie slave ma teraz czas do wystąpienia kolejnego zbocza T5 na zapisanie danej z szyny PWDATA, po wystąpieniu którego mostek wystawia adres Addr 2 na szynę PADDR oraz wymusza stan niski na magistrali PENABLE. Jednocześnie zmienia stan linii PWRITE na niski, co sygnalizuje, że z urządzenia slave będzie następował odczyt danych. Magistrala APB przechodzi zatem w stan SETUP, stan magistrali AHB w dalszym ciągu nie ulega zmianie. Stan ten trwa do kolejnego zbocza narastającego HCLK i PCLK to znaczy zbocza T6, po wystąpieniu którego na magistrali APB mostek wymusza stan wysoki na linii PENABLE, co powoduje przejście tej magistrali w stan ENABLE oraz jednocześnie po stronie magistrali AHB przełącza linię HREADY w stan wysoki co informuje urządzenie master, że przy kolejnym zboczu narastającym zegara może zatrzasnąć prawidłowe dane na szynie HRDATA co zakończy proces odczytu danej z urządzenia slave. Urządzenia slave APB ma zatem czas do zbocza T7 na wystawienie prawidłowych danych na szynę PRDATA. W mostku szyna danych PRDATA i HRDATA są ze sobą bezpośrednio połączone, co sprawia, że urządzenie APB slave wystawiając dane na szynę PRDATA powoduje bezpośrednie wystawienie po szynie HRDATA dla urządzenia master. Po zboczu T7 następują analogiczne do poprzednich zapis i odczyt danych Data 3 i Data 4 z urządzenia slave

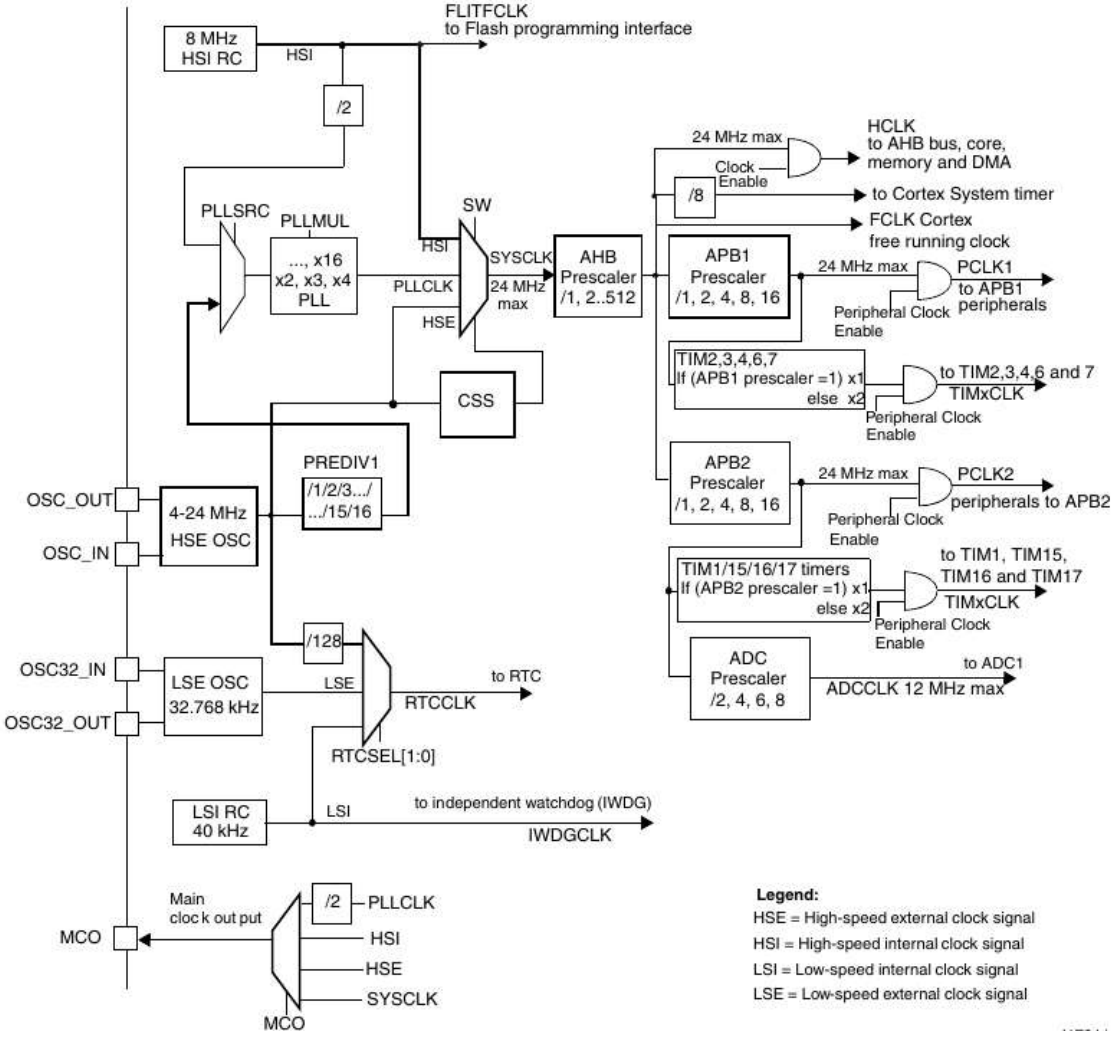
Warto w tym miejscu zwrócić uwagę na sygnał zegara. Domyślnie, sygnały HCLK i PCLK są tym samym sygnałem zegarowym. W omawianym mikrokontrolerze jest jednak możliwość uzyskania sygnałów PCLK poprzez podział sygnału HCLK przez 2, 4, 8 lub 16 w preskalerze osobno dla magistrali APB1 i APB2. Należy zwrócić uwagę, że operacje w dalszym ciągu będą przeprowadzane synchronicznie, ale po stronie magistrali APB będą trwały odpowiednio: 2, 4, 8 lub 16 razy dłużej niż przy równej częstotliwości sygnałów HCLK i PCLK.

3.4. Dystrybucja sygnałów zegarowych

Układ generowania i dystrybucji sygnałów zegarowych w omawianym mikrokontrolerze jest o wiele bardziej rozbudowany niż w przypadku mikroprocesora 80C51. Spowodowane jest to głównie tym, że mikrokontrolery zaprojektowane w oparciu o architekturę ARM charakteryzują się możliwością niewielkiego poboru mocy przy wysokiej wydajności. W tym celu układ kontroli musi mieć możliwość zmian parametrów sygnałów zegarowych (źródeł i częstotliwości) podczas pracy mikrokontrolera w celu zmniejszenia poboru energii, gdyż należy pamiętać, że wzrost częstotliwości pracy układów cyfrowych powoduje zwiększenie strat, a tym samym zapotrzebowania na energię

elektryczną. W omawianym mikrokontrolerze istnieje możliwość indywidualnego wyboru, do których urządzeń peryferyjnych ma być doprowadzony sygnał zegarowy w celu odłączenia nieużywanych urządzeń i kolejnej oszczędności energii. Oczywiście w przypadku urządzenia stacjonarnego zasilanego z zasilacza wtyczkowego, jakim jest DSM-51 nie ma to wielkiego znaczenia, nie mniej jednak w urządzeniach przenośnych jest to nieoceniona funkcjonalność.

Schemat sieci dystrybucji sygnałów zegarowych (ang. *Clock tree*) został przedstawiony na rys. 4.33.



Rys. 3.22. Sieć dystrybucji sygnałów zegarowych w mikrokontrolerze STM32F100RBT6B, opracowano wg [12]

Omawiany mikrokontroler posiada cztery źródła sygnałów zegarowych, są to generatory:

- HSI (ang. *High Speed Internal* – wewnętrzny, wysokiej częstotliwości),
- LSI (ang. *Low Speed Internal* – wewnętrzny, niskiej częstotliwości),
- HSE (ang. *High Speed External* – zewnętrzny, wysokiej częstotliwości),
- LSE (ang. *Low Speed External* – zewnętrzny, niskiej częstotliwości).

Sygnały zegarowe LSI i HSI są generowane wewnątrz mikrokontrolera poprzez generatory typu RC. Częstotliwości ich pracy są określone na stałe, dla HSI jest to 8 MHz, dla LSI jest to 40 kHz. Generatory HSE i LSE są przyłączone do zewnętrznych wyprowadzeń mikrokontrolera, do których można przyłączyć rezonator kwarcowy lub ceramiczny albo sygnał zegarowy (sinusoidalny, trójkątny lub prostokątny o wypełnieniu około 50%) pochodzący z zewnętrznego generatora. Częstotliwość pracy generatora LSE jest również

stała i wynosi 32,768 kHz, natomiast częstotliwość pracy HSE zależy od częstotliwości rezonansowej przyłączonego rezonatora lub częstotliwości zewnętrznego generatora i może się zmieniać w zakresie od 4 do 24 MHz.
Wśród układów potrzebujących sygnału zegarowego znajdują się takie, do który można doprowadzić sygnał z różnych źródeł, ale także, które źródło sygnału mają przypisane na stałe bez możliwości wyboru. Urządzeniem takim jest interfejs programowania pamięci Flash zabudowanej w mikrokontrolerze, gdyż doprowadzony do niego sygnał zegarowy o nazwie FLITFCLK jest sygnałem branym prosto z linii zegarowej HSI bez możliwości wyboru. Drugim urządzeniem jest niezależny Watchdog (ang. *IWDG*), który zawsze jest taktowany z zegara LSI. Pozostałe elementy systemu mają możliwość wyboru źródła sygnału. Zegar czasu rzeczywistego (ang. *RTC*) może być taktowany poprzez jedno z trzech źródeł: LSI, LSE albo HSE podzielone przez 128. Wyboru dokonuje się poprzez ustawianie flag RTCSEL[1:0] w rejestrze RCC_BDCR. Głównym sygnałem zegarowym w systemie jest sygnał zegara systemowego SYSCLK. Z niego biorą początek sygnały zegarowe na wszystkie wewnętrzne magistrale oraz urządzenia peryferyjne oprócz Watchdoga IWDG i zegara RTC. Sygnał SYSCLK może mieć wybrane jedno z trzech źródeł: bezpośredni HSE, bezpośredni HSI oraz sygnał PLLCLK pochodzący z powielacza częstotliwości opartego na pętli synchronizacji fazy PLL. Pętla synchronizacji fazy PLL pozwala w tym przypadku na wygenerowanie stabilnego sygnału zegarowego o częstotliwości będącej wielokrotnością sygnału otrzymywanego na wejście. Mnożenie może odbywać się: x16, x4, x3, x2, a jako sygnał wejściowy PLL może przyjąć jeden z dwóch dostępnych: HSI/2 lub HSE/1, /2, /3, …. /15, /16. Dzięki takiemu zabiegowi możliwe staje się wygenerowanie sygnału zegarowego o częstotliwości 24 MHz przy użyciu na przykład zewnętrznego rezonatora kwarcowego o częstotliwości rezonansowej 8 MHz. Wybór sygnału dla linii SYSCLK odbywa się programowo poprzez ustawienie flag SW w rejestrze konfiguracyjnym RCC_CFGR. Oprócz powyższego na rys. 4.33 widoczny jest układ CSS, który także może dokonywać wyboru źródła sygnału SYSCLK. Jeżeli element ten zostanie aktywowany, to czuwa nad prawidłowym przebiegiem sygnału zegarowego HSE. W chwili wykrycia błędu, moduł nadzorczy CSS przełącza wybór sygnału SYSCLK na zegar HSI oraz zgłasza odpowiednie przerwanie do systemu. Sygnał SYSCLK przechodzi przez główny preskaler, gdzie częstotliwość jego może zostać podzielona przez liczbę z zakresu od 1 do 512. Za preskalerem zachodzi dopiero podział sygnału na różne elementy systemu. Bezpośrednio brany jest sygnał FCLK Cortex, który służy do próbkowania sygnałów przerwań i taktowania elementów debugera. Podobnie, również bezpośrednio jest brany sygnał HCLK dla magistral AHB-Lite, oraz rdzenia, pamięci i modułu DMA, z tą różnicą, że sygnał HCLK może zostać wyłączony w celu ograniczania poboru energii. Z sygnału za głównym preskalerem po podzieleniu przez /8 powstaje sygnał zegara systemowego (ang. *System timer*). Jest to timer niezależny od pozostałych timerów umieszczonych w mikrokontrolerze, który należy do rdzenia Cortex-M3. Według założeń może zostać on wykorzystany do generowania przerwań, które później można wykorzystać na przykład w celu przełączania procesów czy zadań w systemie wielozadaniowym. Natomiast równie dobrze można wykorzystać pochodzące od niego przerwania w każdy inny sposób, bądź nie obsługiwać ich w ogóle. Sygnał pochodzący z głównego preskalera rozdziela się jeszcze na dwa elementy: preskaler dla magistrali APB1 i peryferii do niej przyłączonych oraz preskaler dla magistrali APB2 i elementów do niej przyłączonych. W obu preskalerach można nastawić podział częstotliwości na /1, /2, /4, /8 lub /16. Bezpośrednio z preskalera magistrali APB1 powstaje sygnał PCLK1 magistrali APB1, który dodatkowo można wyłączyć, analogicznie sytuacja się ma z sygnałem PCLK2 magistrali APB2 pochodzącym z drugiego preskalera. Za preskalerem dla magistrali APB1 znajdują się jeszcze powielacze częstotliwości, które

umożliwia ewentualne jej zdwojenie, służące do wygenerowania sygnałów zegarowych dla timerów: TIM2, 3, 4, 6 i 7. Za preskalerem dla magistrali APB2 znajdują się również powielacze dla timerów: TIM1, 15, 16 i 17, a także preskaler o podziale /2, /4, /6 i /8 do generowania sygnału zegarowego ADCLK dla przetwornika A/C. Większość przełączeń preskalerów i źródeł sygnałów można dokonać w dowolnym momencie pracy mikrokontrolera oprócz niektórych, jak na przykład zmiana źródła sygnału dla linii SYSCLK może nastąpić tylko wówczas, gdy sygnał, na który jest zmieniany jest dostępny i jest stabilny, w przeciwnym wypadku przełączenie nie nastąpi. Analogicznie, mikrokontroler nie zezwoli na wyłączenie danego oscylatora (np. HSE), jeżeli jest on wykorzystywany bezpośrednio czy pośrednio przez PLL do generowania sygnału SYSCLK. Mikrokontroler posiada również możliwość wysłania wybranego sygnału zegarowego na wyprowadzenie mikrokontrolera MCO. Użytkownik ma do wyboru możliwość wyprowadzenia sygnałów: HSE, HSI, SYSCLK oraz PLLCLK/2.

3.5. Układ resetu mikrokontrolera

Układ resetu, inaczej zerowania mikrokontrolera znajduje się fizycznie w tej samej jednostce, co układ kontroli nad sygnałami zegarowymi, stąd jego nazwa RCC (ang. *Reset and Clock Control*). W urządzeniu występują trzy typu sygnału zerowania mikrokontrolera: reset systemowy (ang. *system Reset*), reset od układów kontroli zasilania (ang. *power Reset*) oraz reset kontrolera kopii zapasowej (ang. *backup domain Reset*).

Reset systemowy powoduje ustawienie rejestrów do wartości domyślnych, czyli wartości po resecie (ang. *reset value*) oprócz flag zerowania w kontrolerze sygnałów zegarowych w rejestrze CSR oraz rejestrów umieszczonych w kontrolerze kopii zapasowej. Sygnał ten zostaje wygenerowany w przypadku jednego z następujących wydarzeń:

- niski stan na linii NRST (reset zewnętrzny mikrokontrolera),
- zakończenie zliczania watchdoga okienkowego (WWDG reset),
- zakończenie zliczania watchdoga niezależnego (IWDG reset),
- reset programowy (ang. *software rest – SW*),
- reset od kontrolera niskiego poboru mocy (ang. *Low-power reset*).

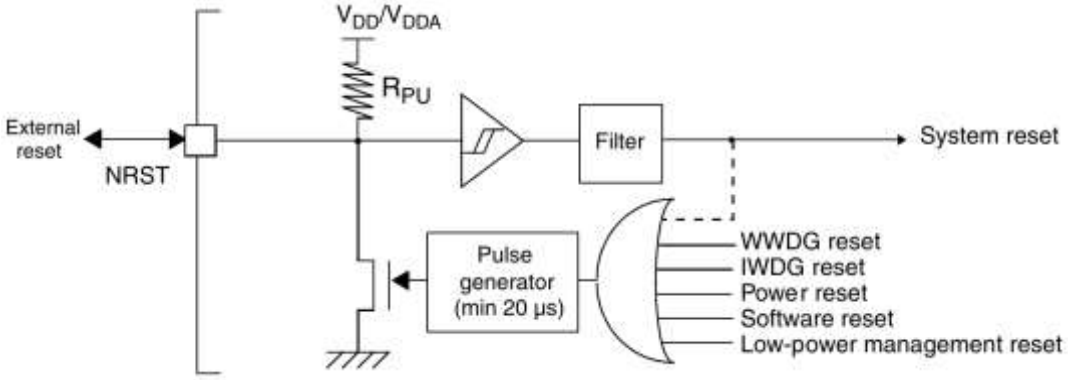
Źródło resetu jest odnotowywane poprzez ustawienie odpowiednich flag w rejestrze RCC_CSR. Flagi te nie są kasowane podczas występowania sygnału zerującego, w związku z czym po powrocie mikrokontrolera do poprawniej pracy można sprawdzić przyczynę wystąpienia resetu. Reset od kontrolera niskiego poboru mocy może nastąpić, jeżeli ustawione są odpowiednie flagi w rejestrach opcji zapisanych w pamięci Flash. Wówczas reset systemowy jest generowany przy przechodzeniu do stanu Standby albo do stanu Stop mikrokontrolera.

Reset od układów kontroli zasilania na następuje w przypadku zaistnienia jednego ze zdarzeń:

- włączenie lub wyłączenie zasilania,
- wyjście z trybu pracy Standby.

Reset ten powoduje przywrócenie wartości domyślnych we wszystkich rejestrach łącznie z rejestrem RCC_CSR zawierającym flagi źródeł sygnału resetującego za wyjątkiem rejestrów kopii zapasowej. Schemat układu wewnętrznego resetu mikrokontrolera przedstawiono na rys. 4.34. Na rysunku widać, że wyprowadzenie mikrokontrolera NRST jest wewnątrz mikrokontrolera spolaryzowane napięciem V_{DD}/V_{DDA} poprzez rezystor podciągający R_{PU}. Linia ta poprzez bramkę z przerzutnikiem Schmitta tworzy linię sygnału resetu systemowego. Wygenerowanie sygnału System reset może zostać wykonane poprzez sprowadzenie linii wyprowadzenia NRST do potencjału masy. Może odbyć się to poprzez zewnętrzny układ lub wewnętrzny tranzystor typu MOSFET. Tranzystor ten

sterowany jest generatorem zapewniającym trwanie impulsu zerującego przynajmniej przez 20 µs, natomiast sam generator wyzwalany jest poprzez pojawienie się przynajmniej jednego z sygnałów zerowania: od Watchdoga niezależnego, od Watchdoga okienkowego, od kontrolera napięcia zasilania, od kontrolera niskiego poboru mocy od resetu programowego.



Rys. 3.23. Uproszczony schemat obwodów resetujących mikrokontrolera, opracowano wg [12]

Po wystąpieniu sygnału system reset mikrokontroler zaczyna wykonywać zapisane w programie instrukcje. W pierwszej kolejności odczytuje wartość spod adresu 0x0000 0000 i interpretuje ją, jako adres szczytu stosu, następnie odczytywana jest wartość spod adresu 0x000 0004, którą interpretuje jako adres, pod który powinien odbyć się skok w celu rozpoczęcia wykonywania programu (ang. *reset vector*).

Reset kontrolera kopii zapasowej oddziałuje tylko na kontroler rejestru kopii zapasowej. Możliwe są dwa źródła tego sygnału:

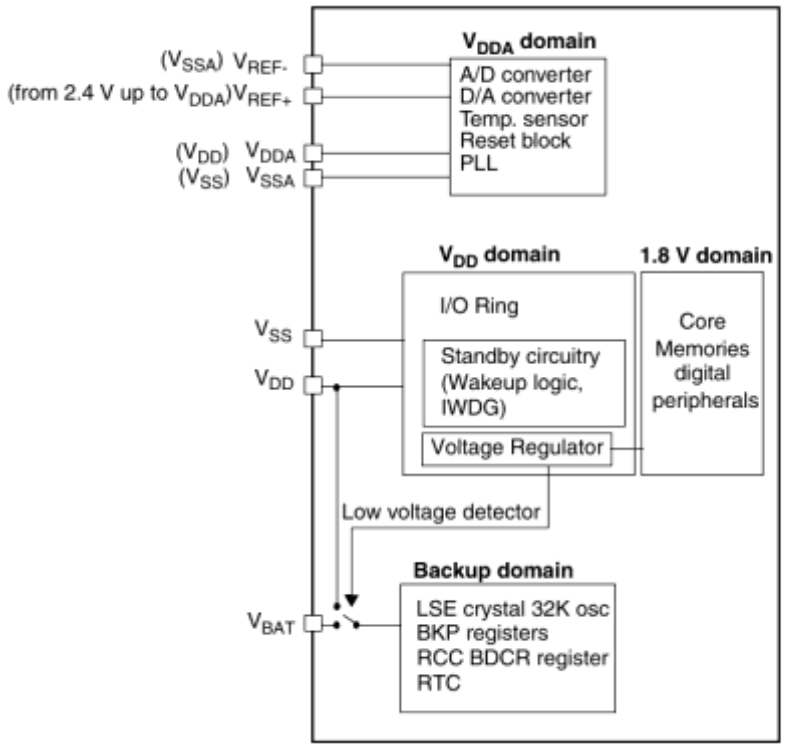
- reset programowy,
- reset od powrotu napięcia zasilania kontrolera kopii zapasowej.

Reset programowy wywoływany jest poprzez ustawienie flagi BDRST w rejestrze kontrolnym RCC_BDCR. Reset od pojawienia się napięcia zasilania generowany jest przy pojawieniu się napięcia V_{BAT} lub V_{DD} w przypadku, gdy oba te zasilania nie były wcześniej obecne.

3.6. Wewnętrzne układy zasilania mikrokontrolera

Mikrokontroler może być zasilany napięciem stałym z zakresu od 2,0 do 3,6 V przyłączonym do wyprowadzeń V_{DD} oraz V_{SS}. Dodatkowo, możliwe jest przyłączenie baterii o napięciu z tego samego zakresu podtrzymującej pracę niektórych elementów systemu. Osobno zasilana jest także część analogowa przetworników A/C i C/A. Schemat przedstawiający strukturę zasilania mikrokontrolera został przedstawiony na rys. 4.35. Na rysunku przedstawione są również wyprowadzenia V_{REF-} oraz V_{REF+} służące do podania napięcia referencyjnego dla przetworników A/C i C/A jednak w mikrokontrolerze STM32F100RBT6B z uwagi na 64-wyprowadzeniową obudowę nie są one dostępne i wewnętrznie są połączone z wyprowadzeniami zasilania części analogowej V_{DDA} oraz V_{SSA}. Mikrokontroler oferuje możliwość podtrzymania pracy niektórych elementów przy zaniku napięcia V_{DD} poprzez podtrzymanie bateryjne. Bateria przyłączona jest do wyprowadzenia V_{BAT} i w czasie, gdy mikrokontroler zasilany jest wyprowadzeniami V_{DD} nie jest ona używana. Odpowiedzialny za to jest przełącznik sterowany przez detektor niskiego napięcia zasilania i w czasie, gdy napięcie to spadnie poniżej 2,0 V, to przełącza zasilanie wybranych układów na pochodzące z wyprowadzenia V_{BAT}. Urządzeniami tymi są: oscylator LSE, zegar czasu rzeczywistego RTC, rejestry przechowywane w kontrolerze kopii zapasowej oraz rejestr kontrolny kontrolera kopii zapasowej. Mikrokontroler posiada wewnętrzny regulator napięcia (ang. *Voltage regulator*), konwertujący napięcie zasilania

do poziomu 1,8 V potrzebnego do zasilania rdzenia, pamięci oraz urządzeń peryferyjnych. Po wystąpieniu sygnału reset, regulator napięcia jest włączony, natomiast w trakcie pracy mikrokontrolera stan jego może ulec zmianie w zależności od trybu pracy całego mikrokontrolera.



Rys. 3.24. Schemat blokowy układów zasilania mikrokontrolera STM32F100RBT6B, opracowano wg [12]

W trybie pracy z pełną wydajnością, regulator dostarcza pełnej mocy do elementów potrzebujących napięcia 1,8 V (rdzenia, pamięci i urządzeń peryferyjnych). W trybie stop mikrokontrolera, regulator dostarcza ograniczoną moc do elementów zasilanych z 1,8 V, zachowywane są zawartości rejestrów i dane zapisane w pamięci SRAM. Natomiast w przypadku, gdy mikrokontroler znajduje się w stanie Standby, regulator napięcia jest wyłączony, zawartości rejestrów i pamięci SRAM są tracone oprócz danych znajdujących się w kopii bezpieczeństwa i obwodów trybu Standby.

3.7. Tryby oszczędzania energii

Domyślnie, po resecie systemowym, czyli na przykład po podaniu napięcia zasilania, mikrokontroler pracuje w trybie RUN, to znaczy z pełną wydajnością. W celu oszczędności energii istnieje jednak kilka trybów pracy mikrokontrolera z mniejszym poborem mocy, są to: uśpienie (ang. *Sleep*), wstrzymanie (ang. *Stop*) i tryb pozostania w stanie gotowości (ang. *Standby*). Oprócz powyższych, możliwe jest także zmniejszanie częstotliwości zegara systemowego czy taktowania magistral APB. Zestawienie możliwych trybów pracy z mniejszym poborem mocy przedstawiono w tab. 4.9. Pierwszym z trybów redukcji mocy jest tryb uśpienia. W trybie tym sygnał zegara odłączony jest jedynie od samej jednostki centralnej procesora, natomiast pozostałe urządzenia peryferyjne włączając w także elementy peryferyjne rdzenia Cortex-M3 takie jak sterownik przerwań NVIC czy timer SysTick, pozostają w niezmienionym stanie. Przejście do trybu uśpienia odbywa się poprzez wykonanie instrukcji WFI (ang. *Wait for interrupt*) lub WFE (ang. *Wait for event*). W zależności od stanu flagi SLEEPONEXIT zawartej w rejestrze CSB_SCR różne są mechanizmy przejścia w tryb uśpienia. Jeżeli

flaga jest wyzerowana, uśpienie następuje od razu po wywołaniu instrukcji WFI lub WFE, w przypadku ustawienia omawianej flagi na wartość 1, to uśpienie następuje, po opuszczeniu przerwania, w którym instrukcja WFI lub WFE została wykonana.

Tablica 3.6. Zestawienie trybów oszczędzania energii w mikrokontrolerze STM32F100RB6TB

Nazwa trybu	Przejście	Wybudzenie	Działanie na zegar elementów zasilanych z 1,8 V	Działanie na zegar elementów zasilanych z V_{DD}	Regulator napięcia
Uśpienie (Sleep)	Instrukcja WFI	Dowolne przerwanie	Wyłączenie zegara jednostki centralnej CPU. Brak działania na pozostałe elementy	Brak działania	Załączony w trybie pracy z pełną wydajnością
	Instrukcja WFE	Zdarzenie Wakeup			
Wstrzymanie (Stop)	PDDS = 0 LPDS = 0 lub 1 SLEEPDEEP = 1 + wywołanie instrukcji WFI lub WFE	Wybudzenie sygnałem na linii EXTI (skonfigurowaną w rejestrach EXTI)	Sygnał zegarowy wszystkich urządzeń zasilanych z 1,8 V wyłączony	Wyłączenie oscylatorów HSI oraz HSE	Załączony jak wyżej lub w trybie ograniczonej wydajności
Gotowość (Standby)	PDDS = 1 SLEEPDEEP = 1 + wywołanie instrukcji WFI lub WFE	Zbocze narastające na wyprowadzeniu WKUP lub alarm od układu RTC, zerowanie sygnałem zewnętrznym lub zadziałanie watchdoga niezależnego			Wyłączony

W celu powrotu mikrokontrolera do pracy normalnej po wywołaniu rozkazu WFI, wystarczy wystąpienie dowolnego przerwania, natomiast w przypadku rozkazu WFE, powrót do pracy normalnej może nastąpić tylko poprzez wystąpienie zdarzenia Wybudź (ang. *Wakeup*). Tryb uśpienia charakteryzuje się najwyższym poborem energii spośród wszystkich trybów redukcji mocy, ale również najszybszym powrotem to trybu pracy normalnej.

Drugi z trybów oszczędzania energii powoduje odłączenie sygnałów zegarowych od wszystkich urządzeń zasilanych z napięcia 1,8 V, ale z zachowaniem stanów rejestrów oraz zawartości pamięci SRAM. Wyłączone są oscylatory HSI oraz HSE oraz pętla PLL. Regulator napięcia może pracować albo z pełną mocą (flaga LPDS w rejestrze PWR_CR wyzerowana) lub z ograniczoną wydajnością (flaga LPDS ustawiona). Przejście do trybu stop odbywa się również poprzez wywołanie jednej z instrukcji WFI lub WFE, ale przy ustawionej wcześniej fladze SLEEPDEEP w rejestrze SCB_SCR i wyzerowanej fladze PDDS w rejestrze PWR_CR. Wybudzenie następuje poprzez pojawienie się sygnału na jakimkolwiek wyprowadzeniu skonfigurowanym jako zewnętrzne przerwanie w przypadku wcześniejszego użycia rozkazu WFI, lub poprzez pojawienie się sygnału na jakimkolwiek wyprowadzeniu skonfigurowanym jako zdarzenie zewnętrzne w przypadku wcześniejszego użycia instrukcji WFE. Po wyjściu z trybu stop jako źródło zegara systemowego wybrany jest oscylator HSI bez względu na wybór przed przejściem do trybu stop. Tryb charakteryzuje się mniejszym zużyciem energii niż sleep, jednak czas powrotu do pracy normalnej mikrokontrolera jest dłuższy, ze względu na konieczność powtórnego

uruchomienia generatora HSI oraz ewentualnego przywrócenia regulatora napięcia do pracy z pełną wydajnością.

Ostatni z trybów pracy z ograniczonym poborem mocy – Standby, jest najbardziej energooszczędny ze wszystkich trybów. Przejście to tego trybu odbywa się podobnie jak w przypadku trybu stop, z tą różnicą, że flaga PDDS w rejestrze PWR_CR musi być ustawiona (w przeciwnym wypadku mikrokontroler przejdzie do trybu stop), a flaga LPDS sterująca pracą regulatora napięcia nie ma już znaczenia, z tego względu, że regulator napięcia w trybie gotowości jest zawsze wyłączany. W zawiązku z powyższym, tracona jest zawartość pamięci SRAM oraz rejestrów oprócz pamięci kopii zapasowej Wybudzenie z trybu może spowodować pojawienie się zbocza narastającego na wyprowadzeniu WKUP, zdarzenie „alarm" od zegara czasu rzeczywistego RTC lub reset od zbocza na zewnętrznym wyprowadzeniu NRST lub niezależnego watchdoga IWDG. Czas potrzebny na powrót do normalnej pracy to czas potrzebny na przejście procedury zerowania mikrokontrolera, czyli m.in. uruchomienia regulatora napięcia, generatorów i wykonania instrukcji potrzebnych do skonfigurowania mikrokontrolera.

Oprócz powyższych użytkownik ma możliwość na wpływ poboru mocy mikrokontrolera poprzez oddziaływanie na częstotliwość pracy zegara systemowego oraz magistrali APB. Odpowiednie rejestry można konfigurować w trakcie pracy programu i w razie zapotrzebowania na mniejszą moc obliczeniową ustawiać większe podziały na preskalerach sygnałów zegarowych. Warto również pamiętać, że nieużywane urządzenia peryferyjne nie powinny być podłączone do źródła sygnałów zegarowych, ale domyślnie po resecie mikrokontrolera tak właśnie są skonfigurowane.

3.8. Kontroler kopii zapasowej

Omawiany mikrokontroler posiada możliwość przechowywania zawartości wybranych rejestrów w kontrolerze kopii zapasowej (ang. *Backup Register*) podczas braku napięcia zasilania V_{DD}. W celu podtrzymania wartości wykorzystywane jest napięcie baterii przyłączonej do wyprowadzenia V_{BAT}. Kontroler posiada pojemność dziesięciu 16-bitowych rejestrów, czyli ma możliwość zapisania dwudziestu bajtów danych. Zawartość rejestrów jest zachowywana także w trakcie wystąpienia resetu systemowego mikrokontrolera. Korzystanie z kopii zapasowej, odbywa się w ten sposób, że w programie wpisuje się wartości (na przykład nastawienia wprowadzone przez użytkownika) do rejestrów od BKP_DR1 do BKP_DR10, które zachowywane są przez kontroler kopi zapasowej. Po powrocie napięcia zasilania, zamiast przywracać ustawienia domyślne nastawień, można przywrócić nastawienia zapisane w rejestrach BKP_DRx.
Bateria podtrzymująca rejestry w kopii zapasowej podtrzymuje również pracę zegara czasu rzeczywistego RTC (ang. *Real Time Clock*) wraz z oscylatorem LSE, który dostarcza do niego sygnał o częstotliwości 32,768 kHz.

3.9. Sterownik przerwań NVIC

Przerwania w systemach mikroprocesorowych są niezwykle istotnym elementem pracy całego układu. Praktycznie jedynie tylko najprostsze aplikacje są w stanie poprawnie pracować bez wykorzystywania funkcji przerwań. W pozostałych przypadkach przerwania znaczenie upraszczają działanie programu oraz przy wykorzystaniu trybów zmniejszonego poboru mocy także powodują oszczędności energii i wydłużenie czasu pracy na zasilaniu bateryjnym. Poprzez przerwanie należy rozumieć wstrzymanie wykonywania kolejnych instrukcji programu głównego w celu zapoczątkowania wykonywania rozkazów z innego miejsca w kodzie programu (wykonywania instrukcji przerwania). Po wykonaniu rozkazów zawartych w przerwaniu, jednostka centralna powraca do wykonywania programu głównego w miejscu, w którym został on wstrzymany. Źródłem zgłoszeń żądań

przerwań mogą być różne urządzenia w systemie. Praktycznie każde urządzenie peryferyjne, w tym: przetworniki A/C i C/A, timery, timer SysTick, kontrolery portów szeregowych, kontroler DMA, zegar RTC oraz inne mogą zgłosić sygnał żądania obsługi przerwania i to najczęściej kilka różnych. Na przykład kontroler portu UART może zgłosić przerwanie po odebraniu ramki danych, a także po jej nadaniu. Takie rozwiązanie powoduje, że mikrokontroler szybko reaguje na asynchronicznie pojawiające się sygnały od urządzeń peryferyjnych i wykonuje wówczas odpowiednie fragmenty instrukcji zawartych w kodzie programu. W przypadku nie korzystania z przerwań, mikrokontroler musiałby ciągle sprawdzać stany poszczególnych flag w rejestrach urządzeń peryferyjnych w celu stwierdzenia czy dane urządzenie nie zgłasza zapotrzebowania na obsłużenie. Wszystkie bowiem urządzenia peryferyjne są urządzeniami typu slave, nie mogą one więc zapoczątkować komunikacji. Mogą co najwyżej zapisać odpowiednie flagi w swoich rejestrach konfiguracyjnych i zgłosić odpowiednią linią informację do kontrolera przerwań, że potrzebują obsłużenia. Wykorzystany w niniejszej pracy mikrokontroler wyposażony jest w zaawansowany kontroler przerwań NVIC (ang. *Nested Vectored Interrupt Controller* – Zagnieżdżony, Wektorowy Kontroler Przerwań).

Tablica 3.7. Tablica wektorów przerwań mikrokontrolera STM32F100RB6TB z wybranymi przerwaniami

Pozycja	Numer IRQ	Typ priorytetu	Skrót	Opis	Adres
-	-	-	-	Zarezerwowane	0x0000 0000
-	-3	Na stałe	Reset	Zerowanie mikrokontrolera	0x0000 0004
-	-2	Na stałe	NMI_Handler	Przerwania niemaskowalne	0x0000 0008
-	-1	Na stałe	HardFault_Handler	Wszystkie rodzaje błędów	0x0000 000C
…	…	…	…	…	…
0	7	Ustawialny	WWDG	Przerwanie od watchdoga okienkowego	0x0000 0040
…	…	…	…	…	…
6	13	Ustawialny	EXTI0	Przerwanie zewnętrzne, linia 0	0x0000 0058
7	14	Ustawialny	EXTI1	Przerwanie zewnętrzne, linia 1	0x0000 005C
8	15	Ustawialny	EXTI2	Przerwanie zewnętrzne, linia 2	0x0000 0060
9	16	Ustawialny	EXTI3	Przerwanie zewnętrzne, linia 3	0x0000 0064
10	17	Ustawialny	EXTI4	Przerwanie zewnętrzne, linia 4	0x0000 0068
…	…	…	…	…	…
18	25	Ustawialny	ADC1	Globalne przerwanie od przetwornika A/C	0x0000 0088
…	…	…	…	…	…
28	35	Ustawialny	TIM2	Globalne przerwanie od timera 2	0x0000 00B0
29	36	Ustawialny	TIM3	Globalne przerwanie od timera 3	0x0000 00B4
…	…	…	…	…	…
37	44	Ustawialny	USART1	Globalne przerwanie od kontrolera portu szeregowego USART1	0x0000 00D4
…	…	…	…	…	…

Kontroler jest układem nadzorczym kontrolującym stan linii ze stanami zgłoszeń od poszczególnych urządzeń peryferyjnych oraz samego rdzenia Cortex-M3, gdyż rdzeń w omawianym mikrokontrolerze również jest w stanie generować żądania obsłużenia przerwań. Kontroler NVIC obsługuje 57 różnych źródeł przerwań w tym 16 od rdzenia. Po stwierdzeniu, że dane urządzenie zgłasza zapotrzebowanie na obsłużenie przerwania,

kontroler wysyła do jednostki CPU informację, że należy przerwać wykonywany obecnie kod i rozpocząć wykonywać instrukcje spod wskazanego adresu. Adresy umieszczone są w tak zwanej tablicy wektorów przerwań (stąd nazwa wektorowy kontroler przerwań). Tablica taka umieszczona jest na początku przestrzeni adresowej programu poczynając od adresu 0x0000 0004. Pod tymi adresami znajdują się adresy z przestrzeni programu z umieszczonymi programami poszczególnych przerwań. Dzięki takiemu zabiegowi przy wystąpieniu przerwania, kontroler od razu jest wstanie podać jednostce CPU adres spod którego należy rozpocząć wykonywanie programu. Pod adresem 0x0000 0004 umieszczony jest adres, który zostanie wykonywany po resecie systemowym mikrokontrolera. Mikrokontroler jest urządzeniem 32-bitowym, w związku z czym kolejne adresy w wektorze będą przesunięte o 32 bity, czyli 4 bajty. Częściowa tablica wektorów przerwań omawianego mikrokontrolera została przedstawiona w tab. 4.10, natomiast pełna tablica wektorów przerwań znajduje się w dokumentacji producenta [12]. Domyślnie większość przerwań jest wyłączona w sterowniku NVIC, użytkownik powinien świadomie uruchomić funkcjonalność przerwań dla potrzebnych urządzeń. Istnieją natomiast przerwania, które są zawsze włączone i nie ma możliwości ich wyłączenia (zamaskowania). Są to tak zwane przerwania niemaskowalne (ang. *NMI – Non Maskable Interrupt*). Przerwaniem niemaskowalnym w omawianym mikrokontrolerze jest obsługa zgłoszenia od układu CSS kontrolującego sygnał zegarowy HSE. W przypadku uaktywnienia tego układu i wystąpienia błędu w sygnale HSE zostaje wygenerowane zgłoszenie obsłużenia przerwania, które na pewno zostanie wykonane. Uruchamiając więc moduł kontroli CSS należy również umieścić odpowiedni adres do programu przerwania w wektorze NMI_Handler.

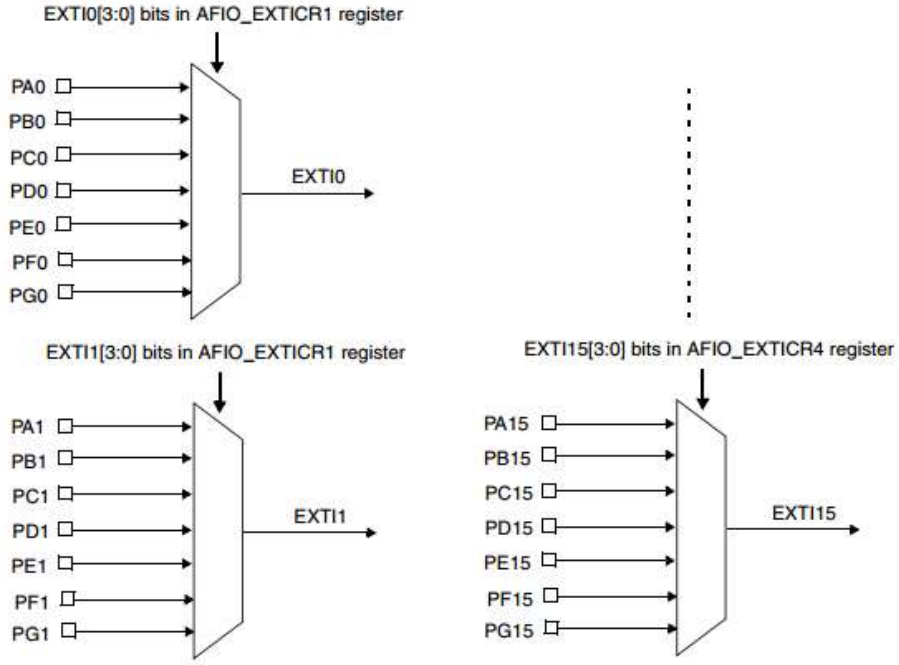
3.9.1. Priorytety przerwań

W tab. 4.10 wyjaśnienia wymagają jeszcze trzy kolumny: Pozycja, Numer IRQ (ang. *Interrupt Request Number – numer żądania przerwania*) i typ priorytetu. W przypadku, gdy w systemie skonfigurowane jest jedno przerwanie, to ustawienia priorytetów nie mają znaczenia, gdyż zgłoszenia żądania przerwania, powoduje wstrzymanie obecnie wykonywanego programu głównego (ewentualnie wybudzenie procesora), wykonanie instrukcji przerwania i powrót do programu głównego, czy uśpienia. Więcej komplikacji może się pojawić, w przypadku gdy w skonfigurowane zostanie więcej źródeł inicjujących przerwania. Jako, że mogą pojawić się w dowolnym momencie pracy mikrokontrolera (na przykład zgłoszenie od klawiatury), może się także zdarzyć, że kolejne żądanie przerwania przyjdzie w trakcie poprzedniego przerwania. W takim przypadku w zależności od stosunku priorytetu przerwania obecnie wykonywanego, do przychodzącego mogą nastąpić różne schematy zachowania. W przypadku, gdy przychodzące przerwanie ma priorytet wyższy niż obecnie wykonywane następuje tak zwane wywłaszczenie i obsłużenie przerwania o wyższym priorytecie, później powrót w celu zakończenia przerwania o niższym priorytecie i później dopiero powrót do programu głównego. Stąd słowo „Zagnieżdżony” (ang. *Nested*) w nazwie kontrolera przerwań. W przypadku natomiast, gdy przychodzące przerwanie ma priorytet niższy, to oczekuje ono w kolejce, aż przerwanie obecnie wykonywane zostanie ukończone. Należy przy tym mieć na uwadze, że niższe numery mają wyższy priorytet, to znaczy na przykład priorytet 5 jest wyższy od priorytetu 10. Domyślnie, po restarcie mikrokontrolera, wszystkie przerwania mają ustawiony priorytet na 0. Priorytety ustawia się w rejestrach od NVIC_IPR0 do NVIC_IPR20. Jako, że priorytety są 4-bitowe, każdy z rejestrów przechowuje priorytety dla czterech wektorów. Pierwszy rejestr zawiera ustawienia przerwań o pozycjach od 0 do 3 (kolumna pozycja z tab. 4.10), drugi od 4 do 7 i tak dalej. Widać również stąd, że przerwania systemowe takie jak reset, czy Hard_fault

nie posiadają pozycji i możliwości zmiany priorytetu. Sterownik NVIC oferuje jednak jeszcze bardziej zawansowaną kontrolę nad priorytetami, to znaczy podział przerwań pod kątem priorytetów na grupy i podgrupy. Jak wspomniano wcześniej, przerwaniom można ustawić priorytet przerwań zapisany na 4-bitach, to znaczy w skali od 0 do 15. Sterownik oferuje podział tych czterech bitów na numer grupy i podgrupy. Domyślnie, wszystkie bity ustawiają numer grupy, natomiast poprzez wpisanie innej wartości w polu oznaczonym PRIGROUP w rejestrze SCB_AIRCR można otrzymać inne podziały, takie jak na przykład 3 bity oznaczające numer grupy i 1 bit oznaczający numer podgrupy aż do wszystkich 4 bitów oznaczających numer podgrupy. Wywłaszczenie przerwania następuje jedynie w przypadku przychodzącego przerwania o wyższym numerze grupy. Jeżeli przychodzące przerwanie ma ten sam numer grupy albo niższy to pozostaje w kolejce niezależnie od numeru podgrupy. Oczekujące przerwania na wykonanie kolejkowane są najpierw po numerze grupy, w ramach konkretnych grup po numerze podgrupy, a w przypadku tych samych numerów podgrupy to numerze żądania przerwania IRQ.

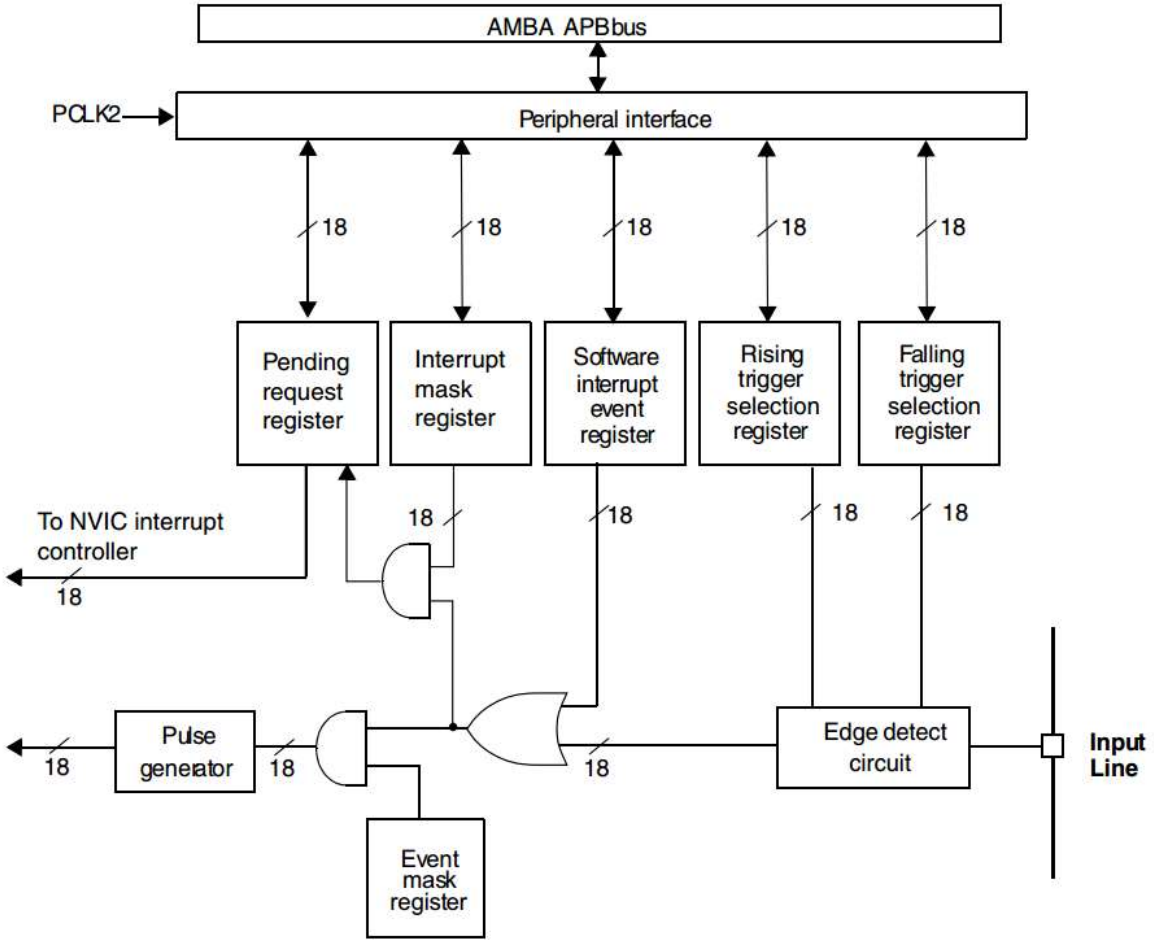
3.9.2. Przerwania zewnętrzne

W mikrokontrolerze źródła przerwań oprócz wewnętrznych urządzeń peryferyjnych mogą także pochodzić z poza samego mikrokontrolera, nazywane są wówczas przerwaniami zewnętrznymi (ang. *EXTI – External Interrupt*). Źródłem takiego przerwania może być na przykład przyłączona klawiatura, zewnętrzny przetwornik analogowo-cyfrowy czy kontroler pamięci EEPROM. Zazwyczaj do tego celu wykorzystane są od dwóch do ośmiu wybranych wyprowadzeń mikrokontrolera, którym można załączyć funkcjonalność żądania przerwania. W omawianym mikrokontrolerze wszystkie porty, które są wyprowadzeniami ogólnego przeznaczenia (ang. *GPIO – General Purpose Input Output*) mogą zostać skonfigurowane jako źródło przerwania. Wyprowadzenia zewnętrzne zgrupowane są w 16 źródeł przerwań w ten sposób, że linie o numerze 0 wszystkich portów (na przykład linia 0 portu A, linia 0 portu B i tak dalej) mogą być źródłem przerwania EXTI0, linie o numerze 1 wszystkich portów mogą być źródłem przerwania EXTI1 i tak analogicznie aż do linii o numerze 15, które mogą być źródłem przerwania EXTI15. Omawiany schemat grupowania przerwań zewnętrznych został przedstawiony na rys. 4.36.



Rys. 3.25. Grupowanie wyprowadzeń w źródła przerwań zewnętrznych, opracowano wg [12]

Oprócz przedstawionych, znajdują się także linie EXTI16 i EXTI17. Linia EXTI16 podłączona jest do wyjścia programowalnego detektora napięcia (ang. *PVD – Programmable voltage detector*), którego można użyć do kontroli napięcia V_{DD} i V_{DDA}. Linia EXTI17 jest natomiast przyłączona do wyjścia alarmowego zegara czasu rzeczywistego RTC. Wyboru czy dana linia ma być źródłem przerwania dokonuje się w rejestrach konfiguracyjnych od AFIO_EXTICR1 do AFIO_EXTICR4. Dla każdej linii od EXTI1 do EXTI17 można skonfigurować dodatkowo tryb wyzwalania przerwania. Poglądowy schemat konfiguracji przerwań zewnętrznych został przedstawiony na rys. 4.37.



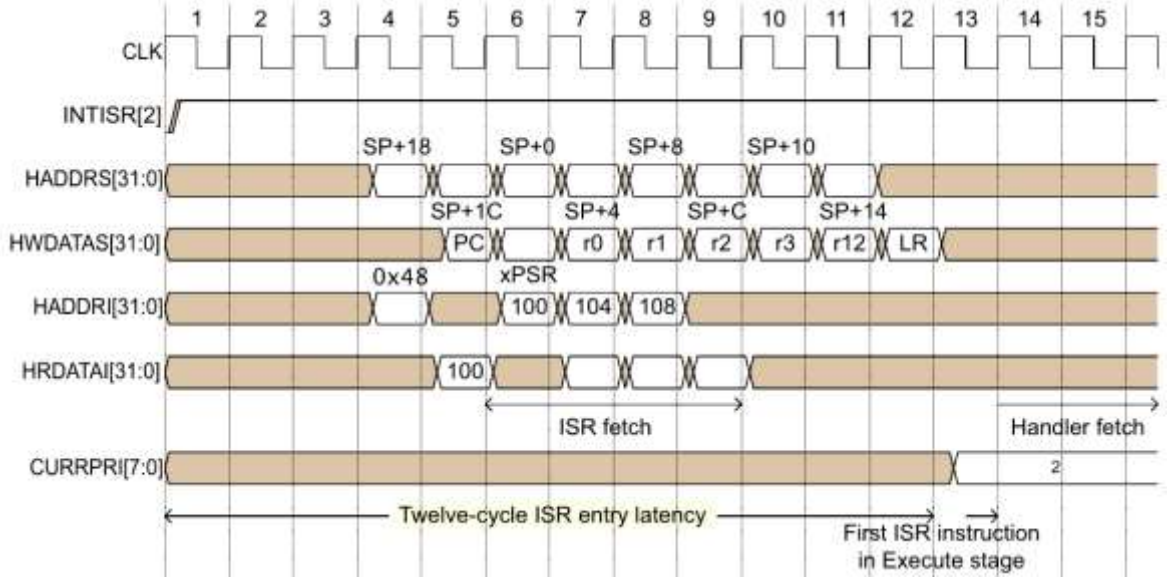
Rys. 3.26. Wpływ rejestrów konfiguracyjnych na poszczególny elementy układu przerwań zewnętrznych, opracowano wg [12]

Na rysunku można zaobserwować, że wyzwolenie przerwania można uzależnić od zbocza narastającego lub opadającego na danej linii, a także zamaskować dane przerwanie poprzez ustawienie odpowiednich flag w rejestrach maskujących. Domyślnie wszystkie przerwania zewnętrzne są zamaskowane, to znaczy programista musi w programie świadomie skonfigurować odpowiednie rejestry w celu uaktywnienia danego przerwania.

3.9.3. Procedura obsługi przerwania

Mikrokontroler po otrzymaniu zgłoszenia przerwania od danego urządzenia, przerywa obecnie realizowany kod programu głównego w celu zrealizowania rozkazów danego przerwania, a następnie powraca do programu głównego w miejscu, w którym został przerwany. Żeby było to możliwe, mikrokontroler pomiędzy wstrzymaniem wykonywania programu głównego, a obsługą przerwania musi wykonać pewne czynności umożliwiające później powrót do programu głównego. Operacjami tymi są: odczytanie

adresu spod odpowiedniego wektora oraz odłożenie na stosie wartości rejestru licznika programu PC (ang. *Program* Counter) procesora przechowującego adres aktualnie wykonywanej instrukcji, rejestrów ogólnego przeznaczenia R0, R1, R2, R3 oraz R12, rejestru stanu procesora PSR (ang. *Processor Status Register*) a także rejestru powrotu LR (ang. *Link Register*) przechowującego ewentualny adres instrukcji powrotu po wykonaniu podprogramu. Procesor w trakcie przerwania wykorzystania jedynie pięć lub mniej rejestrów ogólnego przeznaczenia, stąd tylko niektóre odkładane są na stos. Proces odkładania na stos, a po zakończenie procedury przerwania również przywracania wartości tych rejestrów ze stosu przeprowadzany jest automatycznie przez mikrokontroler. Użytkownik nie musi wywoływać w tym przypadku rozkazów PUSH i POP. Proces wejścia do obsługi przykładowego przerwania został przedstawiony na rys. 4.38.



Rys. 3.27. Zależności czasowe podczas wejścia w przerwanie, opracowano wg [16]

Po pojawieniu się sygnału żądania przerwania INTISR[2], mikrokontroler rozpoczyna sekwencję wejścia w przerwanie. INTISR[2] oznacza przerwanie o pozycji 2 (tab. 4.10). Mikrokontroler równocześnie zaczyna przeprowadzać dwie operacje – odczytanie wektora dla przerwania o pozycji 2 oraz umieszczanie na stosie kolejnych rejestrów specjalnych. Możliwe jest to dzięki temu, że operacja odczytu wektora przerwania oraz rozkazów znajduje się w przestrzeni pamięci programu, którą obsługuje szyba DCode bus (tu oznaczona jako HADDR oraz HWDATA), a stos umieszczony jest w pamięci danych, do której dostęp możliwy poprzez szynę System bus (HADDRS oraz HRDATAS). W czwartym cyklu zegara pod adres SP+18, co należy rozumieć, jako aktualny adres szczytu stosu plus 0x18 bajtów dalej zapisywana jest aktualna wartość rejestru PC.

Stara wartość wskaźnika SP →	poprzednia ostatnia wartość
	xPSR
	PC
	LR
	R12
	R3
	R2
	R1
Nowa wartość wskaźnika SP →	R0

Rys. 3.28. Odłożone na stosie rejestry specjalne i ogólnego przeznaczenia, przy przechodzeniu do przerwania

Później pod adres SP+1C zapisywana jest wartość rejestru xPSR i analogicznie, aż do zapisania w adresie SP+14 wartości rejestru LR co ma miejsce po dwunastu okresach zegara CLK od momentu pojawienia się zgłoszenia żądania przerwania. Widok stosu po przeprowadzeniu operacji odkładania rejestrów został przedstawiony na rys. 4.39. W tym czasie na szynie DCode bus następuje również w czwartym cyklu zegara odczytanie adresu spod wektora danego przerwania, w tym wypadku umieszczonego pod adresem 0x48. W przykładzie, wektor ten zawierał adres 0x100, co oznacza, że kod przerwania rozpoczyna się spod tego adresu. W związku z czym, pierwszy z etapów przetwarzania potokowego (jednostka wstępnego pobierania instrukcji) pobiera trzy kolejne rozkazy z pamięci programu poczynając od adresu 0x100. Kolejne potoki zaczynają przetwarzanie pobranych instrukcji, tak że po dwunastym okresie zegara, czyli w tej chwili co zakończenie wrzucania rejestrów specjalnych na stos, pierwsza z instrukcji przerwania jest już w fazie wykonywania. Szyna CURRPRI[7:0] reprezentuje priorytet obecnie realizowanego przerwania.
Na podstawie rys. 4.38 należy wywnioskować, że podczas wykonywania przerwania, jednostka centralna nie musi wykonywać dodatkowych rozkazów zapisanych w procedurze przerwania (ang. *Overhead*). W porównaniu z procesorem 80C51 jest to znaczna różnica, gdyż w 80C51 w procedurze przerwania, jednostka centralna wykonując kod przerwania dopiero zachowuje na stosie wartości niektórych rejestrów oraz ewentualnie przełącza bank rejestrów, a także wykonuje rozkaz skoku do odpowiedniego fragmentu w kodzie programu. W przypadku mikrokontrolerów z rdzeniem ARM jednostka centralna nie wykonuje tych zadań i od razu przystępuje realizacji zdań związanych z przerwaniem.

Oprócz przerwań, w mikrokontrolerze występują również zdarzenia (ang. *events*). Zdarzenia są obsługiwane całkowicie sprzętowo, a ich pojawienie się nie powoduje wykonania fragmentu kodu programu. Wpływają na pracę mikrokontrolera, a ich obsługa jest wielokrotnie szybsza niż przerwań, ze względu właśnie, na brak wykonywanego programu. Zdarzenie używane jest na przykład w przypadku wybudzenia mikrokontrolera (opuszczenia trybu pracy sleep).
Domyślnie przerwania i zdarzenia oprócz resetu i przerwań niemaskowalnych są zdezaktywowane w systemie, programista musi dokonać konfiguracji odpowiednich rejestrów w celu uaktywnienia wybranych przerwań. Warto również wspomnieć, że wszystkie przerwania dostępne w systemie można wywołać programowo, poprzez ustawienie odpowiednich flag w rejestrach konfiguracyjnych, ma to na przykład wykorzystanie w przypadku potrzeby przetestowania fragmentu programu bez potrzeby oczekiwania na przerwanie od urządzenia peryferyjnego.

3.10. Kontroler bezpośredniego dostępu do pamięci

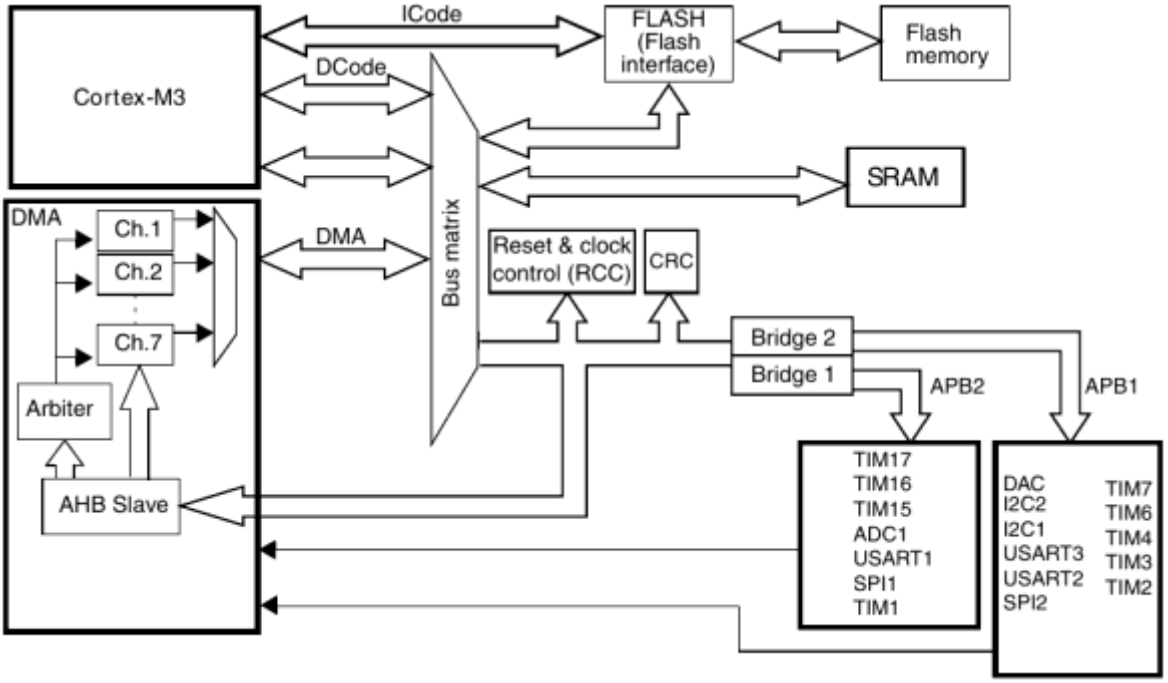
W mikrokontrolerach bez mechanizmu bezpośredniego dostępu do pamięci DMA (ang. *Direct Memory Access*) jednostka centralna zajmuje się każdą operacją przesyłu danych, zarówno pomiędzy urządzeniami peryferyjnymi, a pamięcią RAM jak również w drugą stronę. Przykładem na to może być przetwornik analogowo-cyfrowy, który zgłasza przerwanie w momencie zakończenia przetwarzania, jednostka centralna w ramach obsługi tego przerwania realizuje kod, który pobiera dane z przetwornika A/C i umieszcza je w pamięci RAM pod danym adresem, w celu późniejszej obróbki (na przykład po otrzymaniu przerwania od timera). Operacja przesyłu danej nie jest jedna czynnością wymagającą dużych zdolności obliczeniowych, należy po prostu odczytać wartość z urządzenia jakim jest przetwornik A/C i zapisać w odpowiednie miejsce w pamięci RAM. Można powiedzieć, że szkoda marnotrawić czas jednostki centralnej na tak trywialne operacje.

W celu odciążenia CPU od tego rodzaju zadań powstały kontrolery bezpośredniego dostępu do pamięci DMA. Taki kontroler w pierwszym kroku musi zostać skonfigurowany poprzez ustawienie odpowiednich wartości w jego rejestrach konfiguracyjnych, skonfigurowane muszą zostać również urządzenia peryferyjne chcące korzystać z mechanizmu DMA. Później, w trakcie pracy mikrokontrolera dodatkowe czynności ze strony jednostki centralnej nie są już wymagane. Kontroler DMA nasłuchuje na swoich liniach, czy któreś z urządzeń nie zgłasza zapotrzebowania na bezpośredni dostęp do pamięci. Może nim być wspomniany przetwornik analogowo-cyfrowy. Kontroler DMA po otrzymaniu takiego zgłoszenia odpytuje przetwornik o dane i zapisuje je we wskazane w rejestrach konfiguracyjnych miejsce w pamięci RAM. Co więcej, DMA umożliwia także przesyłanie danych w drugą stronę, to znaczy urządzenie jakim jest na przykład port UART może zgłosić zapotrzebowanie na dostęp do danej w pamięci RAM spod wskazanego adresu. Kontroler DMA w tym przypadku odczytuje daną z podanego miejsca i zapisuje do urządzenia UART. Może to być ta sama dana, która została odczytana od przetwornika A/C. Dzięki takiemu rozwiązaniu, jednostka centralna CPU nie przerywa swojej pracy i dalej realizuje kod programu, podczas gdy mikrokontroler za sprawą DMA odczytał daną z przetwornika A/C i wystawił ją na port UART. Ma to szczególnie duże znaczenia w mikrokontrolerach o architekturze LDR-STR czyli wykonywania rozkazów przez jednostkę centralną tylko na danych znajdujących się w rejestrach podręcznych, gdyż w celu zapisania danej z jednego miejsca w pamięci do drugiego, najpierw trzeba daną odczytać i zapisać do rejestru podręcznego (np. R2), a później z tego rejestru zapisać pod wskazany adres. W przypadku bez wykorzystywania DMA, jednostce centralnej zajęłoby to około kilku - kilkunastu cykli zegara dłużej niż z wykorzystaniem DMA.
Główne cechy zastosowanego kontrolera DMA są następujące:

- 7 niezależnie konfigurowanych kanałów,
- każdy z 7 kanałów jest sprzętowo skojarzony z wybranym urządzeniem peryferyjnym w celu wyzwolenie transferu z wykorzystaniem DMA, możliwe jest także programowe wyzwolenie takiego transferu,
- możliwość programowego ustawienia priorytetów dla każdego kanału: bardzo wysoki, wysoki, średni i niski, bądź zastosowania priorytetów sprzętowych, gdzie wyższość obsługi mają kanały o niższym numerze (np. CH2 ma wyższy priorytet niż CH5),
- niezależnie konfigurowana szerokość doczytywanego/zapisywanego słowa (8, 16 lub 32 bity),
- wsparcie dla danych zapisywanych w buforze cyklicznym,
- możliwość wygenerowania przerwań w zależności od przebiegu transferu danych,
- różne rodzaje transferów: z peryferii do pamięci, z pamięci do peryferii, z pamięci do pamięci, z peryferii do peryferii,
- dostęp do pamięci Flash, SRAM, urządzeń peryferyjnych przyłączonych do magistral APB1, APB2 oraz AHB,
- programowalna ilość danych do przetransferowania, aż do 65536 bajtów.

Wyjaśnienia wymaga tu zapis do bufora cyklicznego. W tym trybie pracy kanału, możliwe jest określenie początku i końca danych w pamięci SRAM dla danych odczytywanych urządzenia peryferyjnego. Każda dana z urządzenia peryferyjnego odczytywana przy kolejnych zgłoszeniach DMA jest zapisywana pod kolejne adresy w pamięci RAM, aż do momentu, gdy kontroler napotka ostatni adres bufora cyklicznego, wówczas kolejna dana z urządzenia peryferyjnego będzie zapisana pod pierwszym adresem bufora.
Fragment schematu wewnętrznego mikrokontrolera zawierający kontroler DMA przedstawiono na rys. 4.40. Producent nie podaje jednak do wiadomości użytkowników jakiego rodzaju jest to kontroler, czy zaimplementowany na podstawie jednego

z projektów opracowanego przez firmę ARM, czy jest to konstrukcja własna. Kontroler posiada dwa interfejsy AHB-Lite – master oraz slave. Interfejs slave służy do konfigurowania rejestrów kontrolera DMA poprzez zapisanie w nich odpowiednich wartości.



Rys. 3.29. Fragment schematu blokowego mikrokontrolera STM32F100RBT6B z widocznym kontrolerem DMA, opracowano wg [12]

Interfejs ten widoczny jest w przestrzeni adresowej jako urządzenie peryferyjne jak każde inne, w zawiązku z czym zapisem wartości pod ten adres zajmuje się przy konfiguracji rejestrów jednostka centralna procesora. Z drugiej strony kontrolera znajduje się interfejs master magistrali AHB-Lite, który łączy się z przełącznikiem Bus Matrix. Jak wspomniano w poprzednich rozdziałach, przełącznik ten pozwala wielu magistralom master dostęp do poszczególnych urządzeń slave. W tym przypadku ma za zadanie udostępnianie pamięci SRAM oraz urządzeń peryferyjnych w zależności od potrzeb albo jednostce CPU albo kontrolerowi DMA, a w przypadku jednoczesnego żądania dostępu do urządzenia przeprowadza arbitraż, który z masterów powinien mieć obecnie dostęp, a który powinien poczekać.

Zastosowany w omawianym mikrokontrolerze kontroler DMA jest siedmiokanałowy. Oznacza to, że może zostać skonfigurowany do maksymalnie siedmiu różnych zadań przesyłu danych. Wspominany wcześniej przykład z odczytem danej ze sterownika A/C mógłby zostać skonfigurowany na przykład na kanale nr 1 (ang. *CH1 – Chanel 1*), a zapis danej do kontrolera portu szeregowego UART już na kanale nr 2. Każdy z kanałów ma osobną konfigurację adresów, spod/do których następuje zapis/odczyt danych. Kanały między sobą różnią się priorytetami, to znaczy w przypadku kilku zgłoszeń żądań bezpośredniego dostępu do pamięci najpierw obsługiwane są żądania dla kanałów z wyższym priorytetem.

Możliwe do wyboru źródła żądań dla poszczególnych kanałów zostały przedstawione w tab. 4.11. Zestawienie należy rozumieć w ten sposób, że wyzwolenie transferu dla kanału nr 1 kontrolera DMA może nastąpić przez zgłoszenie od przetwornika A/C lub kanału 3 timera TIM2 lub kanału 1 timera TIM4, po wcześniejszym skonfigurowaniu danego kanału DMA oraz skonfigurowania urządzenia peryferyjnego będącego źródłem sygnału żądania bezpośredniego dostępu do pamięci. Należy przy tym

zwrócić uwagę, że wyzwolenie od pewnego urządzenia, nie musi wiązać się z późniejszym transferem danych z/do tego urządzenia.

Tablica 3.8. Zestawienie źródeł żądań dla każdego kanału DMA

Urządzenie peryferyjne	Kanał 1	Kanał 2	Kanał 3	Kanał 4	Kanał 5	Kanał 6	Kanał 7
A/C	A/C						
SPI		SPI1_RX	SPI1_TX	SPI2_RX	SPI2_TX		
USART		USART3_TX	USART3_RX	USART1_TX	USART1_RX	USART2_TX	USART2_RX
I^2C				I^2C2_TX	I^2C2_RX	I^2C1_TX	I^2C1_RX
TIM1		TIM1_CH1	TIM1_CH2	TIM1_CH4 TIM1_TRIG TIM1_COM	TIM1_UP	TIM1_CH3 TIM1_CH2 TIM1_CH1	
TIM2	TIM2_CH3	TIM2_UP			TIM2_CH1		TIM2_CH2 TIM2_CH4
TIM3		TIM3_CH3	TIM3_CH4 TIM3_UP			TIM3_UP TIM3_TRIG	
TIM4	TIM4_CH1			TIM4_CH2	TIM4_CH3		TIM4_UP
TIM6 lub C/A Kanał1			TIM6_UP / DAC_Chanel1				
TIM7 lub C/A Kanał2				TIM7_UP / DAC_Chanel2			
TIM15					TIM15_CH1 TIM15_UP TIM15_TRIG TIM15_COM		
TIM16						TIM16_CH1 TIM16_UP	
TIM17							TIM17_CH1 TIM17_UP

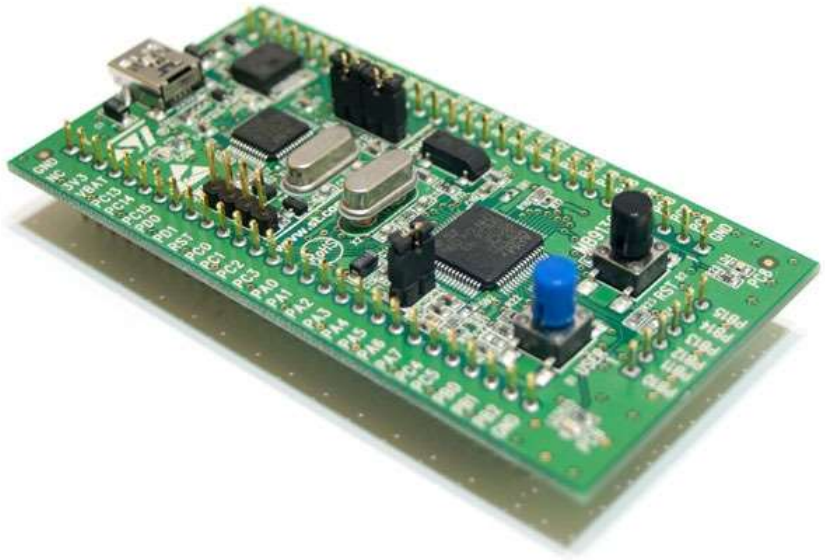
Na przykład kanał 2 można skonfigurować w ten sposób, że gdy przyjdzie zgłoszenie od kanału 3 timera 3 (TIM3_CH3) to wyzwalamy transfer z pamięci SRAM spod danego adresu do sterownika I^2C, którego zgłoszenie żądania transferu DMA nie jest podłączone do kanału 2 DMA.
Domyślnie, po resecie mikrokontrolera, żaden kanał w sterowniku DMA nie jest uaktywniony. Użytkownik powinien świadomie skonfigurować i uruchomić potrzebne mu kanały w kontrolerze DMA.

4. Przykładowe otoczenie procesora ARM Cortex

4.1. Budowa zestawu STM32VLDISCOVERY

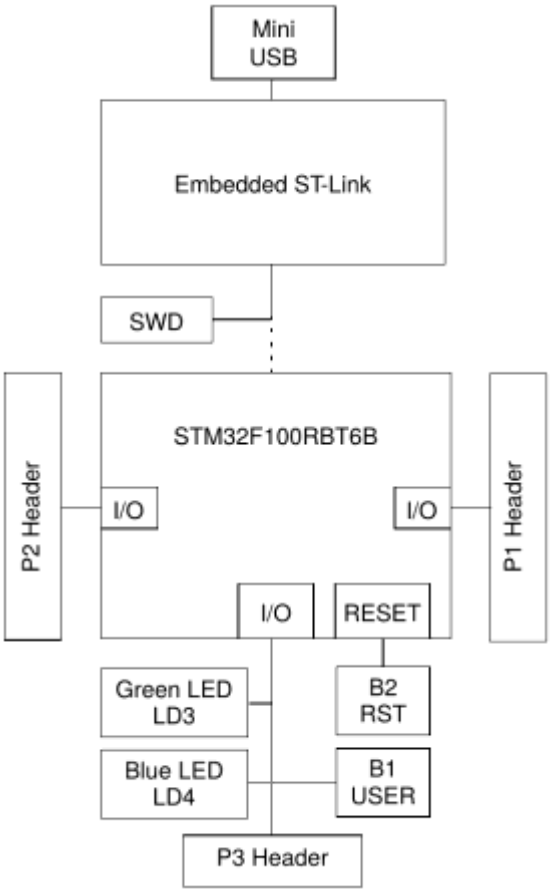
Wygląd zestawu STM32VLDISCOVERY został przedstawiony na rys. 4.5. Układ składa się z dwóch zasadniczych części połączonych ze sobą na stałe – samego zestawu uruchomieniowego z mikrokontrolerem STM32F100RBT6B oraz z programatora-debugera ST-Link z portem miniUSB. Połączenie pomiędzy obydwoma częściami następuje poprzez zworki. Producent przewidział w ten sposób możliwość zastosowania programatora ST-Link do współpracy z innymi procesorami niż ten wlutowany w zestaw ewaluacyjny. Oprócz powyższych na płytce zabudowany jest także stabilizator, dzięki czemu zestaw można zasilać napięciem 5V poprzez piny bądź z gniazda miniUSB. Schematy blokowy oraz ideowy zestawu zostały przedstawione na rys. 4.6. oraz rys. 4.7. Głównym elementem całego zestawu jest oczywiście mikrokontroler STM32F100RBT6B wraz z szeregiem otaczających go elementów. W śród elementów można wyróżnić przede wszystkim dwa 28-pinowe oraz jedno 6-pinowe jednorzędowe złącza w postaci goldpinów

o rastrze 2,54 mm służące do podłączeń wyprowadzeń mikrokontrolera do zewnętrznych układów bez specjalistycznych narzędzi.

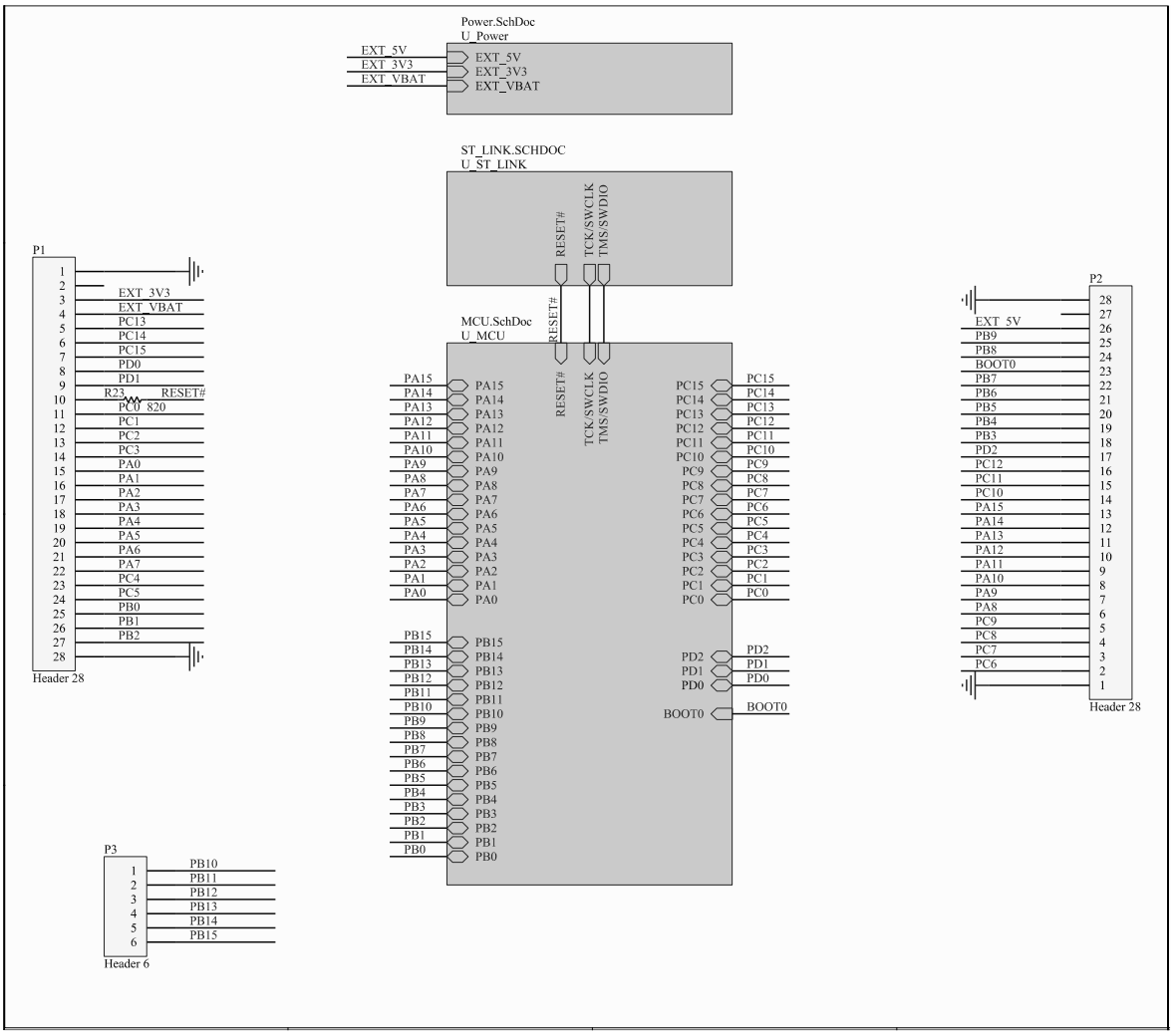


Rys. 4.1. Zestaw ewaluacyjny STM32VLDISCOVERY

Na złączach tych oznaczonych jako P1, P2, P3 wyprowadzona jest większość dostępnych w mikrokontrolerze pinów oraz zasilanie +5 V oraz +3,3 V. Cztery wyprowadzenia z listw zaciskowych podłączone są do potencjału masy (ang. *GND*).



Rys. 4.2. Schemat blokowy zestawu ewaluacyjnego STM32VLDISCOVERY, opracowano wg [7]



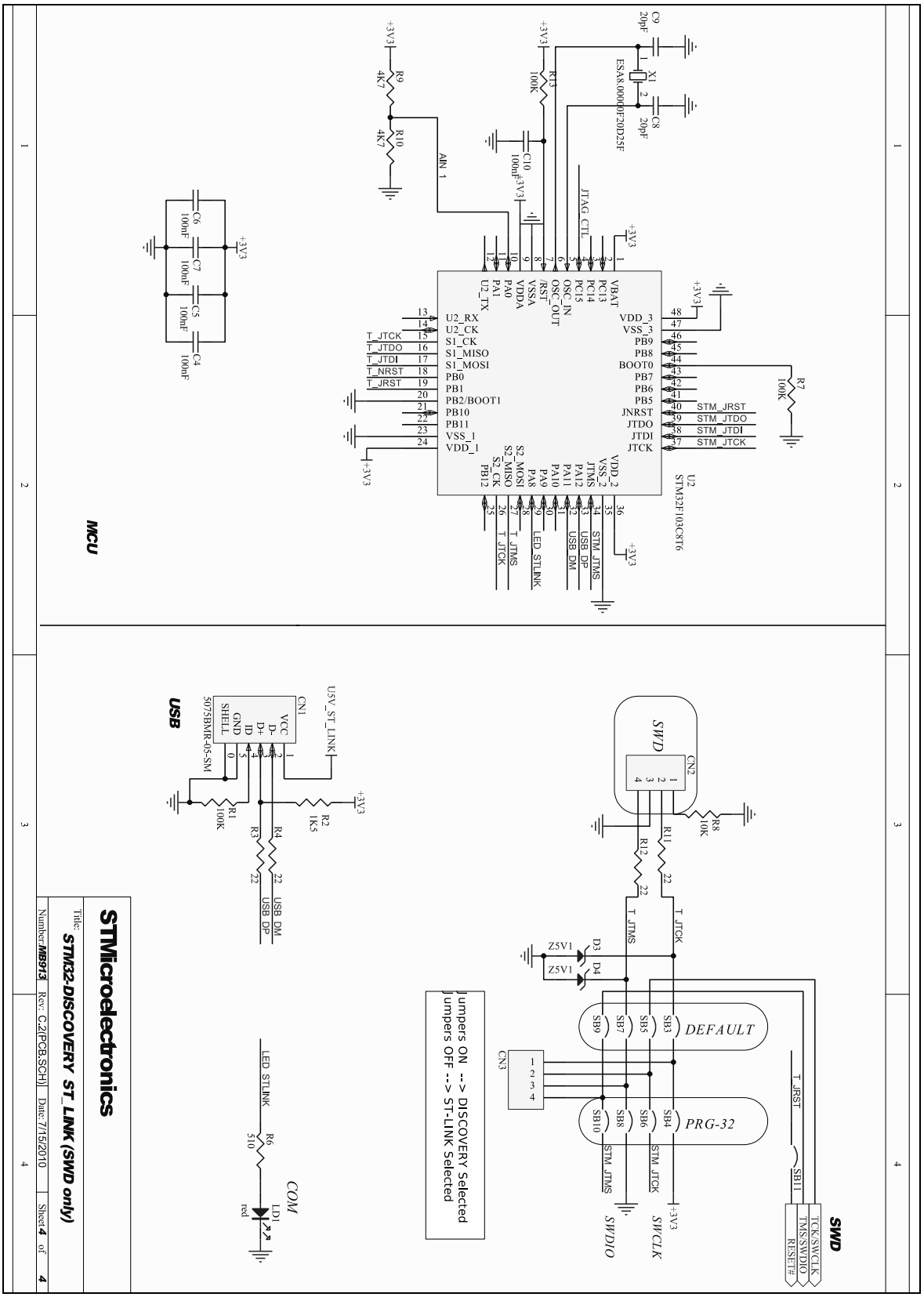
Rys. 4.3. Schemat ideowy zestawu ewaluacyjnego STM32VLDISCOVERY, opracowano wg [7]

4.1.1. Programator ST-Link

Zintegrowany z zestawem ewaluacyjnym programator/debuger ST-Link można wykorzystywać na dwa sposoby:

- do programowania/debugowania głównego mikrokontrolera STM32F100RBT6B,
- do programowania/debugowania innych płyt z mikrokontrolerami poprzez podłączenie przewodami do gniazda SWD CN2 programatora.

Fragment schematu STM32VLDISCOVERY przedstawiającego część z programatorem został przedstawiony na rys. 4.8. Centralnym elementem programatora jest mikrokontroler U2 typu STM32F103C8T6. Jest to również 32-bitowy układ z rdzeniem ARM Cortex-M3 zamknięty w 48-wyprowadzeniowej obudowie. Mikrokontroler wyposażony jest m.in. w interfejs USB, 2xSPI, 2xI^2C, 3xUSART, posiada 64 kB pamięci Flash oraz 20 kB pamięci SRAM. Mikrokontroler jest fabrycznie zaprogramowany na pełnienie funkcji debugera/programatora mikrokontrolerów wykorzystując interfejs SWD (ang. *Serial Wire Debug*). Należy tutaj zaznaczyć, że w mikrokontrolerach rodziny ARM7/ARM9 standardowym interfejsem do programowania i debugowania był JTAG (ang. *Joint Test Action Group*). W rdzeniach rodziny Cortex, ARM Holdings dodał dodatkowo interfejs SWD. Nowy interfejs został stworzony w celu minimalizacji potrzebnych linii mikrokontrolera do obsługi debugera - JTAG potrzebował 5 linii, a SWD tylko 2 linie. Dodatkowo SWD wykorzystuje te same linie, co JTAG, w związku z czym przy pracy w trybie SWD pozostałe linie JTAG użytkownik może wykorzystać do swoich celów.



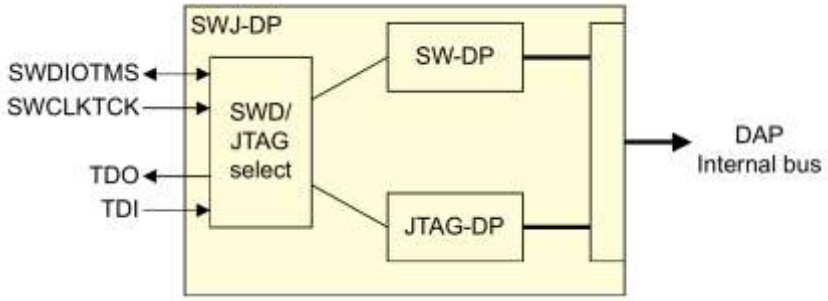
Rys. 4.4. Schemat wbudowanego programatora-debugera ST-Link, opracowano wg [7]

W tab. 4.2. zestawiono linie interfejsów JTAG i SWD wraz z opisem funkcji. Wspólne linie TCK i SWCLK oraz TMS i SWDIO są w dokumentacji [8] określane jako SWCLKTCK (wspólna linia SWCLK i TCK) oraz SWDIOTMS (wspólna linia SWDIO i TMS).

Tablica 4.1. Linie interfejsów debugera JTAG i SWD

Linia JTAG	Funkcja JTAG	Linia SWD	Funkcja SWD
TCK	Test Clock	SWCLK	Clock
TMS	Test Mode Select	SWDIO	Data In/Out
TDI	Test Data In	-	-
TDO	Test Data Out	-	-
TRST	Test Reset	-	-

Po restarcie mikrokontrolera debuger jest ustawiony w tryb JTAG, przełączenie w tryb SWD może nastąpić poprzez wysłanie odpowiedniej sekwencji 16-bitowej przez programator na linię SWDIOTMS. Powoduje to przełączenie interfejsu debugera mikrokontrolera w tryb SWD. Uproszczony schemat wewnętrzny interfejsu debugera z przełączaniem trybów JTAG – SWD został przedstawiony na rys. 4.9.



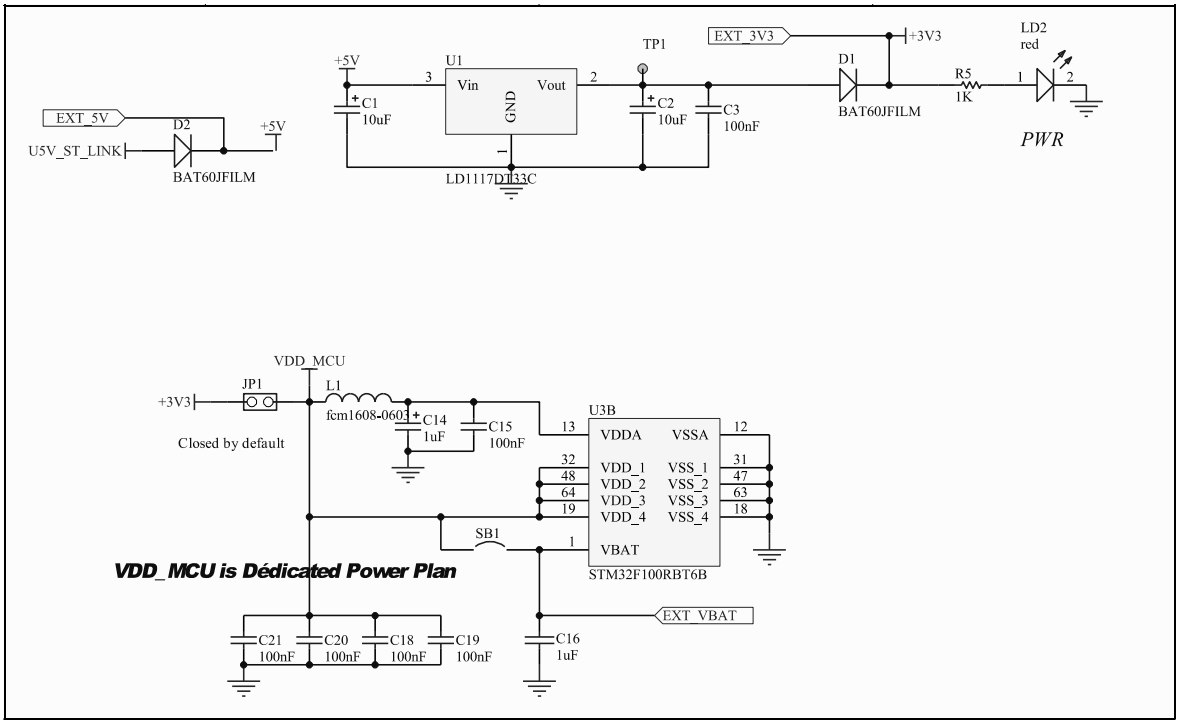
Rys. 4.5. Schemat blokowy interfejsów debugera SWD i JTAG, SWD/JTAG select – moduł przełącznika trybów pracy interfejsu, DAP Internal bus – (ang. *Debug Access Port*) wewnętrzny interfejs debugera, opracowano wg [8]

Od strony sprzętowej wbudowany programator ST-Link wykorzystuje w swoim mikrokontrolerze port SPI do wysyłania i odbierania danych na linii SWDIO oraz sterowania sygnałem zegarowym na linii SWCLK. Z drugiej strony linie mikrokontrolera USB_DP oraz USB_DM podłączone są do gniazda miniUSB do linii D- i D+. Po podłączeniu programatora łączem USB do komputera klasy PC z zainstalowanym systemem operacyjnym Windows i oprogramowaniem Keil ARM SDK urządzenie wykrywane jest jako debuger ST-Link.

Źródłem sygnału zegarowego układu U2 jest wewnętrzny generator z podłączonym zewnętrznym rezonatorem kwarcowym X1 o częstotliwości 8 MHz. Oprócz powyższego na płytce programatora ST-Link znajdują się układy generowania sygnału RESET po pojawieniu się napięcia zasilania (elementy R13 oraz C10) oraz dzielnik napięcia zasilania (R9, R10) podłączony do wejścia PA0 mikrokontrolera. Producent nie wyjaśnia jednak, w jakim celu zastosowano pomiar napięcia 3,3 V. Na płytce znajduje się także czerwona dioda LED SMD, podłączona poprzez rezystor R6 do wyprowadzenia PA8 mikrokontrolera, która sygnalizuję pracę debugera/programatora. Połączenie programatora ST-Link z dalszą częścią płytki zawierającą między innymi mikrokontroler STM32F100RBT6B odbywa się poprzez założenie zworek na złączu CN3. Domyślnie zworki te są założone.

4.1.2. Stabilizacja napięcia zasilania

Zestaw ewaluacyjny wyposażony jest również we własny stabilizator napięcia 3,3 V, dzięki czemu może być zasilany napięciem 5V poprzez wyprowadzenie na złączu P2 lub napięcie z portu miniUSB. Fizycznie elementy stabilizatora znajdują się w części płytki należącej do programatora ST-Link. Schemat zasilania został przedstawiony na rys. 4.10.



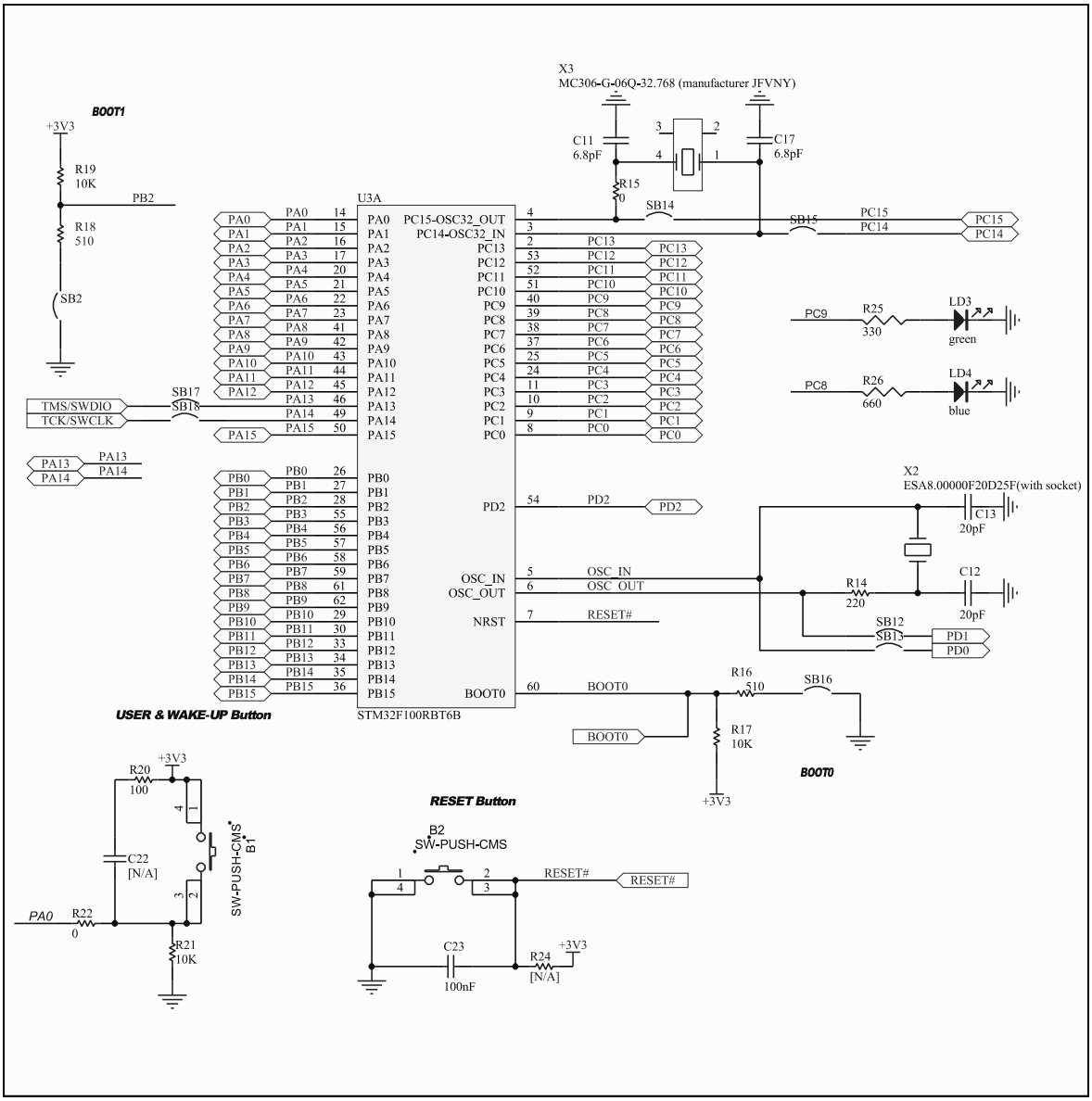
Rys. 4.6. Schemat części zasilającej zestawu STM32VLDISCOVERY, opracowano wg [7]

Głównym elementem części zasilającej jest stabilizator liniowy U1 typu LD1117DT33C. Jest to stabilizator typu LDO (ang. *Low-dropout regulator*), którego wyróżnia od pozostałych stabilizatorów liniowych możliwość poprawnej pracy przy niewielkiej różnicy napięcia wejściowego i wyjściowego. W przypadku LD1117DT33C jest to 1 V. Za stabilizatorem znajduje się dioda prostownicza D1 typu BAT20JFILM. Jest to dioda typu Schottky'ego, która charakteryzuje się mniejszym spadkiem napięcia niż zwykłe prostownicze diody krzemowe. W przypadku diody BAT20JFILM spadek napięcia wynosi średnio 0,35 V przy przepływie prądu 100 mA. Dioda D1 zapobiega przepływowi wstecznego prądu przez stabilizator U1 w przypadku gdyby od strony dalszej części obwodu pojawiło się napięcie wyższe niż od strony stabilizatora (np. przy zasilaniu zestawu bezpośrednio napięciem 3,3 V). Przed stabilizatorem znajduje się również dioda prostownicza D2 tego samego typu, w celu zabezpieczenia portu miniUSB przed pojawieniem się napięcia wyższego od strony zestawu ewaluacyjnego niż występującego na porcie miniUSB. Część zasilająca wyposażona jest również w czerwoną diodę LED SMD LD2 podłączoną poprzez rezystor R5 do zasilania 3,3 V. Dioda sygnalizuje obecność napięcia 3,3 V. Sam mikroprocesor STM32F100RBT6B nie jest zasilany bezpośrednio ze stabilizatora napięciem 3,3 V. W torze zasilania znajduje się jeszcze złącze JP1, na które domyślnie założona jest zworka typu Jumper, gdzie zamiast niej można podłączyć miliamperomierz w celu pomiaru prądu pobieranego przez mikrokontroler. Za złączem JP1 znajdują się kondensatory ceramiczne filtrujące napięcie zasilania oraz filtr dolnoprzepustowy napięcia zasilania zbudowany z dławika L1 oraz kondensatorów C14 i C15 do zasilania części analogowej mikrokontrolera.

4.1.3. Otoczenie mikrokontrolera STM32F100RBT6B

Druga część zestawu ewaluacyjnego, która zajmuje większą część obwodu drukowanego to sam mikrokontroler STM32F100RBT6B wraz z szeregiem elementów towarzyszących. Schemat tej części został przedstawiony na rys. 4.11. Wśród elementów znajdują się dwa rezonatory kwarcowe, pierwszy o częstotliwości 8 MHz przyłączony do

wyprowadzeń 5, 6 mikrokontrolera, drugi o częstotliwości 32,768 kHz przyłączony do wyprowadzeń 3, 4.



Rys. 4.7. Schemat części zestawu ewaluacyjnego zawierającego mikrokontroler STM32F100RBT6B

Rezonator kwarcowy 8 MHz zamontowany jest w płytce przy pomocy specjalnego gniazda, co pozwala na jego łatwą wymianę na inny bez użycia lutownicy. Wyprowadzenia mikrokontrolera, do których podłączony jest rezonator mogą też być używane jako standardowe porty WE/WY, wyprowadzone są więc także na rzędy goldpinów. Oprócz rezonatorów na płytce znajdują się dwie diody LED: LD3 (zielona) oraz LD4 (niebieska) podłączone poprzez rezystory R25 i R26 do wyprowadzeń mikrokontrolera odpowiednio: PC9 i PC8. Płytka zawiera także dwa przyciski w postaci mikrołączników. Przycisk B1 podłączony do linii PA0 mikrokontrolera, może pełnić dowolną funkcję zaprogramowaną przez użytkownika, przycisk B2 podłączony do linii RESET mikrokontrolera, powoduje reset układu. Na linii PA0 w stanie normalnym panuje potencjał masy wymuszony rezystorem R21, naciśnięcie przycisku B1 powoduje podanie napięcia 3,3 V na linię PA0. Odwrotnie jest w przypadku linii RESET, gdzie w normalnym stanie panuje wysoki potencjał 3,3 V dzięki rezystorowi R24, natomiast naciśnięcie przycisku B2 sprowadza linię RESET do potencjału masy. Oprócz powyższego płytka

poprzez rezystory R18, R19 oraz R16 i R17 polaryzuje wyprowadzenia BOOT0 i BOOT1 (PB2) mikrokontrolera. W domyślnej konfiguracji linie te spolaryzowane są w ten sposób, że linia BOOT0 jest na potencjale masy, a linia BOOT1 na wysokim potencjale 3,3 V. Mikrokontroler STM32F100RBT6B w zależności o stanu tych linii w różnych miejscach rozpoczyna czytanie instrukcji. W tab. 4.3. zestawiono możliwe konfiguracje.

Tablica 4.2. Tryby rozruchu

BOOT1	BOOT0	Miejsce czytania instrukcji programu
x	0	Główna pamięć Flash
0	1	Pamięć systemowa
1	1	Wbudowany SRAM

Stan pinów BOOT0 oraz BOOT1 jest zatrzaskiwany w układzie przy pojawieniu się czwartego narastającego zbocza sygnału SYSCLK po wystąpieniu resetu. Więcej na temat przestrzeni adresowej dostępnych pamięci znajduje się w rozdziale 5.3 Organizacja pamięci.

5. Model programowy mikrokontrolera STM32F100RBT6B

Model programowy omawianego mikrokontrolera (ang. *Instruction Set* Architecture) to inaczej mówiąc informacje na temat sposobu jego użycia przez użytkownika, czyli programistę. Dla programisty bowiem, nie jest aż tak istotne w jaki sposób zbudowana jest magistrala AHB-Lite czy informacje w jaki sposób mostek AHB/APB przesyła dane z wybranym urządzeniem peryferyjny mikrokontrolera. Z punktu widzenia użytkownika danego mikrokontrolera najważniejsze są informację na temat listy rozkazów procesora, zestawie dostępnych dla niego rejestrów czy zasadach obsługiwania wyjątków i przerwań, gdyż wszystkie te elementy zależą od architektury danego mikrokontrolera. Celem programisty jest napisanie programu realizującego określone zadania w sposób najbardziej optymalny pod kątem zastosowania.

5.1. Lista instrukcji – informacje ogólne

Mikrokontrolery z rdzeniami ARM umożliwiają pracę w kilkoma różnymi listami instrukcji w zależności od wersji architektury. Pierwotnie mikrokontrolery obsługiwały listę instrukcji ARM. W liście tej każdy rozkaz ma 32-bity i może być wykonywany warunkowo, a programy cechuje duża wydajność. Wadą tej listy jest duża zajętość miejsca w pamięci programu. W celu rozwiązania mankamentu dużej ilości miejsca powstała lista instrukcji Thumb. Można powiedzieć, iż jest to podzbiór listy ARM. Rozkazy w tej liście są 16-bitowe, lecz działania w dalszym ciągu są przeprowadzane na danych 32-bitowych. Programy cechuje duża tak zwana gęstość kodu, to znaczy polecenia zajmują mało miejsca w pamięci. Lista Thumb posiada jednak pewne ograniczenia w stosunku do listy ARM – tylko niektóre operacje mogą być wykonane warunkowo oraz dostępne są tylko pierwsze 8 rejestrów ogólnego przeznaczenia (R0…R7). Rdzenie implementowały obie listy instrukcji, ale należało je przełączyć z jednego trybu do drugiego w celu wykonania instrukcji z drugiej listy, a przełączenie nie następowało natychmiastowo. W związku z tym, jako rozwinięcie listy Thumb powstała lista Thumb-2. Cechuje ją duża gęstość kodu przy wysokiej wydajności. Rozkazy z tej listy są zarówno 16- jak i 32-bitowe. Nie ma jednak potrzeby przełączania trybu pracy rdzenia. W celu umożliwienia pisania optymalnych ze względu na listę Thumb-2 programów stworzono specjalną odmianę języka asemblera – UAL (ang. *Unifield Assembler Language* – zunifikowany język asemblera). Kod programu napisany w języku UAL może zostać zasemblowany zarówno do postaci wykorzystującej listę instrukcji Thumb-2 jak i ARM. Na początku listingu programu definiuje się odpowiednią dyrektywą, jaka lista rozkazów ma zostać użyta. Oprócz powyższych, różne architektury implementują również inne specyficzne listy instrukcji, na przykład: Jazelle, SIMD, TrustZone czy NEON advanced SIMD. Rdzeń Cortex-M3 wykorzystuje jednak tylko instrukcje z listy Thumb-2, więc tej listy dotyczyć będzie dalszy opis.

Wśród rozkazów z listy Thumb-2 są instrukcje 16- i 32- bitowe. Niektóre operacje mogą być wykonywane, jako jedne i drugie. Przykładem może być instrukcja dodawania wartości liczbowej do rejestru przedstawiona na listingu 5.1.

Listing 5.1. Instrukcja dodania liczby do rejestru

```
ADDS R0, #5 ; dodanie liczby 5 do rejestru R0
```

W takim przypadku asembler wybierze domyślny rodzaj instrukcji, najczęściej będzie to 16-bitowa instrukcja z listy Thumb-2. W przykładzie widocznym na listingu 5.2 również zostanie zastosowana instrukcja 16-bitowa, gdyż użycie sufiksu .N (ang. *Narrow*) po nazwie rozkazu powoduje wymuszenie zastosowanie właśnie instrukcji 16-bitowej.

Listing 5.2. Wymuszenie rozkazu 16-bitowego

```
      ADDS.N R0, #5 ; dodanie liczby 5 do rejestru R0
```

Natomiast wymuszenie instrukcji 32-bitowej odbywa się poprzez zastosowanie sufiksu .W (ang. *Wide*), której przykład znajduje się na listingu 5.3.

Listing 5.3. Wymuszenie rozkazu 32-bitowego

```
      ADDS.W R0, #5 ; dodanie liczby 5 do rejestru R0
```

Próba użycia sufiksu .N dla instrukcji, która występuje tylko w wersji 32-bitowej kończy się błędem przy próbie asemblacji. Często w literaturze można spotkać błędne utożsamianie instrukcji 32-bitowych z listy Thumb-2 z 32-bitowymi instrukcjami listy ARM. Kod maszynowy jednych i drugich jest jednak znacząco inny. Instrukcja w języku asembler przedstawiona na listingu 5.1 podczas asemblacji z użyciem listy ARM otrzymuje postać w kodzie maszynowym:

```
      0xE2900005
```

Natomiast ta sama instrukcja przy asemblacji z użyciem listy Thumb-2 i wymuszeniem trybu 32-bitowego otrzymuje postać:

```
      0xF1100005
```

Jak widać nie jest to ta sama postać, co w liście rozkazów ARM. W trybie 16-bitowej instrukcji Thumb ten sam rozkaz maszynowo ma następującą postać:

```
      0x3005
```

Znaczenie poszczególnych bitów w instrukcjach jest różne w zależności od typu instrukcji (np. arytmetyczne, logiczne, porównania). Nie jest celowe więc rozpisywanie znaczenia poszczególnych bitów w kodzie maszynowym z uwagi na fakt, iż w każdej instrukcji będzie on inny.

Większość dostępnych operacji można wykonać warunkowo (ang. *conditional*), to znaczy wykonanie danego rozkazu może zależeć od stanu flag Z, C, N i V w rejestrze specjalnym APSR. Tak też było w 32-bitowych liście rozkazów ARM, natomiast w liście 16-bitowych instrukcji Thumb taką możliwość dawał tylko rozkaz skoku warunkowego (instrukcja B). Przykład porównania rejestru R3 z wartością 0 i wykonania w zależności od wyniku porównania odpowiedniej instrukcji w wersji Thumb zamieszczono na listingu 5.4.

Listing 5.4. Warunkowe wykonanie instrukcji ADD w zależności od wyniku porównania

```
      CMP R3, #0  ; porównaj wartość w rejestrze R3 z wartością 0, jeżeli
                  ; są równe, to ustawiana jest flaga Z
      BEQ skok    ; jeżeli Z=1 (fraza EQ) to skocz (instrukcja B) pod
                  ; etykietę „skok”
      ADD R0, R1, R2 ; R0 = R1 + R2
skok
      ADD R5, R6, R7 ; R5 = R6 + R7
```

W przypadku korzystania z Thumb-2 można zapisać powyższy kod inaczej, to znaczy, frazę warunkową można użyć razem z rozkazem ADD otrzymując bardziej zwięzły i szybciej wykonujący się kod (brak rozgałęzienia). W tym przypadku postać programu będzie taka jak przedstawiono na listingu 5.5.

Listing 5.5. Warunkowe wykonanie instrukcji ADD możliwe do zapisania w liście Thumb-2

```
      CMP R3, #0  ; porównaj wartość w rejestrze R3 z wartością 0, jeżeli
                  ; są równe, to ustawiana jest flaga Z
      ADDNE R0, R1, R2  ; jeżeli Z≠1 to R0 = R1 + R2
      ADD R5, R6, R7    ; R5 = R1 + R2
```

Takich instrukcji warunkowych można umieścić kilka po sobie, lecz należy mieć na uwadze, że jeżeli liczba ich przekracza 4 to bardziej optymalnie jest użycie instrukcji skoku warunkowego (B) z podaniem etykiety w celu ominięcia całego bloku operacji. Należy również uwzględnić, iż niektóre z rozkazów zmieniają stan flag Z, C, N i V. Operacje porównania zawsze zmieniają stan powyższych flag, natomiast dla większości operacji arytmetycznych jest to opcjonalne, na przykład rozkaz ADD nie zmienia stanu flag, ale rozkaz ADDS (mnemonik S – ang. *set flags*) już tak.
Lista dostępnych fraz warunkowych została przedstawiona w tab. 5.1.

Tablica 5.1. Wyrażenia warunkowe dla rozkazów z listy Thumb-2

Skrót	Rozwinięcie (ang)	Warunek	Stan flag
EQ	Equal	Równy	Z = 1
NE	Not Equal	Nierówny	Z = 0
CS	Carry set	Ustawiona flaga przeniesienia carry; większy lub równy (liczby bez znaku)	C = 1
CC	Carry clear	Wyzerowana flaga przeniesienia carry; mniejszy (liczby bez znaku)	C = 0
MI	Negative	Ujemny	N = 1
PL	Positive or Zero	Dodatni lub zerowy	N = 0
VS	Overflow set	Ustawiona flaga przepłnienia	V = 1
VC	Overflow clear	Wyzerowana flaga przepełnienia	V = 0
HI	Unsigned higher	Większy (liczby bez znaku)	C = 1 oraz Z = 0
LS	Unsigned lower or same	Mniejszy lub równy (liczby bez znaku)	C = 0 lub Z = 0
GE	Greater or equal	Większy lub równy	N = V
LT	Less then	Mniejszy	N ≠ V
GT	Greater then	Większy	Z = 0 oraz N = V
LE	Less then or equal	Mniejszy lub równy	Z = 1 lub N ≠ V
AL	Always	Zawsze (skrót można pominąć)	bez znaczenia

Powyżej omawiany przykład można zapisać również z użyciem porównania CBZ (porównaj rejestr z zerem i skocz pod etykietę, jeżeli zmienna w rejestrze jest równa 0). Wówczas omawiany fragment posiadałby postać przedstawioną na listingu 5.6.

Listing 5.6. Skok warunkowy z wykorzystaniem instrukcji CBZ

```
      CBZ R3, skok      ; jeżeli R3 = 0 to skocz pod etykietę „skok”
      ADD R0, R1, R2    ; R0 = R1 + R2
skok
      ADD R5, R6, R7    ; R5 = R6 + R7
```

Należy tu zwrócić uwagę na jeszcze jedną osobliwość języka UAL. Mnemoniki wpisywane w kodzie programu nie mają pełnego odzwierciedlenia później w kodzie maszynowym. Przykładem właśnie takiego zdarzenia są instrukcje z dodanym wyrażeniem warunkowym, gdyż zapis:

```
      ADDNE R0, R1, R2
```

jest w rzeczywistości tłumaczony w kodzie maszynowym na następujące:

```
      IT NE
      ADDS r0,r1,r2
```

Natomiast taki rozkaz:

```
      ADDEQ R0, R1, R2
```

na następujące:

```
      IT EQ
      ADDS r0,r1,r2
```

Widać, że wyrażenie warunkowe zostaje wydzielone z nazwy rozkazu jako operand dla instrukcji IT. Jest to nowość wprowadzona w liście Thumb-2 równoważna wyrażeniu „Jeżeli – To – W przeciwnym wypadku" (ang. *If-Then-Else*). Wyrażenie IT rozpoczyna blok instrukcji, którego wykonanie zależy od spełnienia lub nie podanego warunku jako operand wyrażenia IT. Omawiany przykład można rozumieć następująco: jeżeli EQ (czyli równy, czyli Z = 1) to wykonaj polecenie niżej (czyli ADDS), a jeżeli nie to je pomiń. W tym przypadku blok instrukcji wykonywanych warunkowo składał się tylko z jednej instrukcji. Maksymalnie instrukcji w bloku IT może znajdować się cztery, a każdej z nich nadaje się określenie czy ma być wykonana przy spełnieniu warunku (*Then*) czy nie (*Else*). W tym celu stosuje się dodatkowe litery T lub E następujące po instrukcji IT. Sama instrukcja IT określa długość bloku instrukcji wykonywanych warunkowo na 1. Instrukcja ITx (gdzie x może być literą T lub E) określa długość bloku instrukcji wykonywanych warunkowo na 2, aż do instrukcji ITxyz (gdzie x, y, z mogą być literą T lub E) określa długość bloku instrukcji wykonywanych warunkowo na 4. Pierwsza instrukcja jest zawsze wykonywana przy spełnieniu warunku (*Then*) natomiast wykonanie pozostałych zależy od kolejności liter x, y, z po rozkazie IT. Zostało to zobrazowane na przykładzie przedstawionym na listingu 5.7.

Listing 5.7. Blok czterech instrukcji wykonywanych warunkowo z użyciem rozkazu IT

```
ITTET EQ
MOV R1, #4
MOV R2, #5
MOV R3, #6
MOV R4, #7
```

W tym przypadku, jeżeli Z=1 (instrukcja warunkowa EQ) to zostanie wykonana instrukcja pierwsza z bloku instrukcji (zawsze T – *Then*), instrukcja druga (IT**T**ET) oraz instrukcja czwarta (ITTE**T**). Instrukcja trzecia, czyli MOV R3, #6 nie zostanie wykonana, gdyż posiada znacznik E – *Else* (ITT**E**T). W przypadku gdyby Z≠1, to zostanie wykonana tylko instrukcja trzecia. Taki zapis można zastosować bezpośrednio kodzie programu lub też w postaci formy przedstawionej na listingu 5.8.

Listing 5.8. Blok czterech instrukcji wykonywanych warunkowo z użyciem mnemoników NE i EQ

```
MOVEQ R1, #4
MOVEQ R2, #5
MOVNE R3, #6
MOVEQ R4, #7
```

W jednym i drugim przypadku zostanie wygenerowany ten sam kod maszynowy, z tym że w przypadku pierwszym programista ma większą kontrolę nad blokiem instrukcji IT.

Listing 5.9. Różne warianty instrukcji ADD

```
ADD{S}<c><q> {<Rd>,} <Rn>, #<const>
ADD{S}<c><q> {<Rd>,} <Rn>, <Rm> {,<shift>}

; gdzie:
; {}        - opcjonalny argument
; <#const>  - stała z przedziału 0…4095
; <Rd>      - rejestr przeznaczenia, jeżeli pominięty, to ten sam
;             co <Rn>
; <Rn>      - rejestr zawierający wartość pierwszego argumentu
; <Rm>      - rejestr zawierający wartość drugiego argumentu
; {S}       - jeżeli występuje ten znak, to wykonanie instrukcji
;             aktualizuje flagi w rejestrze APSR
; <c>       - wyrażenie warunkowe (z tab. 4.7)
; <q>       - sufiks .W lub .N
; <shift>   - opcjonalne przesunięcie bitowe drugiego argumentu
```

Niektóre z operacji mogą zostać wykonane ze zmienną liczbą argumentów. Przykładem może być instrukcja dodawania ADD. W zależności od potrzeby programisty instrukcja może przyjmować jedną z kilku postaci zaprezentowanych na listingu 5.9. Uwzględniając wszystkie możliwe kombinacje parametrów uzyskuje się aż 360 wariantów rozkazu ADD. Wyjaśnienia wymaga tu jeszcze opcjonalny parametr <shift>. Dzięki zastosowaniu szybkiego 32-bitowego rejestru przesuwnego (Barrl Shifter) widocznego na rys. 4.16. możliwe jest opcjonalne przesunięcie bitowe lub arytmetyczne drugiego argumentu. Możliwe przesunięcia zestawiono w tab. 5.2.

Tablica 5.2. Wyrażenia warunkowe dla rozkazów z listy Thumb-2

Skrót	Rozwinięcie (ang)	Opis	n
LSL #n	Logical Shift Left	Przesunięcie bitowe w lewo, z zapełnianiem zerami najmniej znaczących bitów	$0 \le n \le 31$
LSR #n	Logical Shift Right	Przesunięcie bitowe w prawo, z zapełnianiem zerami najbardziej znaczących bitów	$0 \le n \le 32$
ASR #n	Arithmetic Sfigt Right	Przesunięcie bitowe w prawo, z zapełnianiem najbardziej znaczących bitów wartością 31-bitu	$0 \le n \le 32$
ROR #n	Rotate Right	Przesunięcie bitowe w prawo, z zapełnianiem z przenoszeniem najmniej znaczącego bitu na pozycję najbardziej znaczącego	$0 \le n \le 31$
RRX	Rotate right one bit	Przesunięcie bitowe w prawo o jeden bit z zapełnieniem najbardziej znaczącego bitu wartością flagi C (*carry*)	-

Dzięki powyższym możliwościom przykład poniższej funkcji

```
R1 = R2 + (R3 * 4)
```

można zapisać w postaci jednej instrukcji

```
ADD R1, R2, R3, LSL #2
```

która w dodatku wykonuje się w ciągu jednego cyklu zegara. Powoduje to wspominaną dużą wydajność mikrokontrolera przy równie wysokiej gęstości kodu.
Z zaprezentowanych relatywnie prostych przykładów widać, że wykorzystując listę instrukcji Thumb-2 programiści mają duże możliwości optymalizacji kodu pod kątem czy to czytelności, czy szybkości działania czy ilości zajmowanego miejsca w pamięci programu.

5.2. Lista instrukcji rdzenia Cortex-M3

Listę rozkazów rdzenia Cortex-M3 można podzielić na sześć działów tematycznych:

- instrukcje dostępu do pamięci,
- instrukcje ogólnego przetwarzania danych,
- instrukcje mnożenia i dzielenia,
- instrukcje nasycania,
- instrukcje rozgałęzień,
- instrukcje pozostałe.

Szczegółowy opis wszystkich dostępnych rozkazów znajduje się w instrukcji programowania mikrokontrolera [9]. Poniżej, ze względu na ograniczenia ilościowe pracy, znajdują się jedynie najważniejsze rozkazy mikrokontrolera wraz ze skrótowym ich opisem.

5.2.1. Instrukcje dostępu do pamięci

Jednostka wykonawcza mikrokontrolera może przeprowadzać działania jedynie na danych zawartych w rejestrach ogólnego przeznaczenia. Oznacza to, że w przypadku potrzeby przeprowadzenia operacji na danych znajdujących się w pamięci lub

w urządzeniach peryferyjnych należy najpierw te dane zapisać w rejestrach ogólnego przeznaczenia, później przeprowadzić działanie i na koniec ewentualnie zapisać w odpowiednie miejsce. Taki model z jednej strony pozwala jednostce CPU szybciej przeprowadzać działania (operowanie tylko na rejestrach ogólnego przeznaczenia), z drugiej jednak strony wymusza wprowadzanie dodatkowych rozkazów dostępu do pamięci. Sumarycznie jednak, korzystne jest takie rozwiązanie.

Zestawienie instrukcji dostępu do pamięci przedstawione jest w tab. 5.3.

Tablica 5.3. Instrukcje dostępu do pamięci

Skrót	Opis
ADR	Ładowanie adresu do rejestru względem adresu w PC
CLREX	Anulowanie trybu wyłączności
LDM{mode}	Załadowanie wielu rejestrów danymi
LDR{type}	Załadowanie rejestru daną
LDREX{type}	Załadowanie rejestru daną w trybie wyłączności
POP	Odłożenie danej na stos
PUSH	Ściągnięcie danej ze stosu
STM{mode}	Zapisanie danych z wielu rejestrów
STR{type}	Zapisanie danej z rejestru
STREX{type}	Zapisanie danej z rejestru w trybie wyłączności

Z podanej listy, rozkazy PUSH i POP oznaczają odłożenie podanych rejestrów na stosie oraz ich pobranie. Adres stosu jest pobierany spod adresu 0x0000 0000 pamięci programu po resecie mikrokontrolera i wpisywany do rejestru SP, który przechowuje adres szczytu stosu. Programista używając rozkazów PUSH i POP nie musi wiedzieć w którym obszarze pamięci leży stos, ważne jest tylko, żeby pamiętał o kolejności odkładania danych rozkazem PUSH, gdyż później rozkaz POP ściąga dane ze stosu w odwrotnej kolejności, to znaczy ostatnia odłożona dana, będzie pobrana jako pierwsza (stąd nazwa stos). Przykład zastosowania omawianych instrukcji został zaprezentowany na listingu 5.10.

Listing 5.10. Przykład użycia rozkazów PUSH i POP

```
MOV R4, #23       ; zapisanie wartości 23 w rejestrze R4,
                  ; od tej chwili wartość rejestru R4 jest równa 23
PUSH {R4}         ; odłożenie zawartości rejestru R4 na stosie,
                  ; wartość rejestru R4 nie zmieniła się i dalej = 23
MOV R4, #3        ; zapisanie wartości 3 w rejestrze R4,
                  ; od tej chwili wartość rejestru R4 równa się 3
POP {R4}          ; pobranie zawartości ze stosu i zapisanie w R4,
                  ; od tej chwili wartość R4 znowu jest równa 23
PUSH {R1, R2, R3} ; odłożenie na stosie w kolejności: R3, R2 i R1
POP {R1, R2, R3}  ; pobranie ze stosu i zapisanie w kol.: R1, R2, R3
PUSH {R1-R3}      ; to samo co PUSH {R1, R2, R3}
```

Jak widać, podając rejestry dla funkcji POP i PUSH należy jest umieścić w nawiasach klamrowych {}. Rozkazy umożliwiają wpisanie kilku rejestrów do odłożenia w jednej instrukcji. Zapis takiej instrukcji polega na podaniu rejestrów do odłożenia oddzielonych przecinkami (,) lub zakresu rejestrów poprzez oddzielenie pierwszego i ostatniego myślnikiem (-). Dla instrukcji PUSH rejestry wówczas odkładane są w kolejności odwrotnej od podanej, to znaczy dla omawianego przykładu najpierw będzie odłożony R3, potem R2 i na końcu R1. Funkcja POP natomiast pobiera dane ze stosu i zapisuje do rejestrów w takiej kolejności jak są podane, to znaczy najpierw R1, potem R2 i na końcu

R3. Zapis taki jak w przykładzie powoduje więc, że dane trafiają z powrotem do rejestrów z których pochodziły.

W przypadku potrzeby odczytania/zapisania pojedynczych danych spod/do konkretnego miejsca w przestrzeni adresowej wykorzystywane są instrukcje LDR i STR. Instrukcje LDR służą do pobrania danej spod podanego adresu i zapisanie jej do rejestrów ogólnego przeznaczenia, natomiast instrukcje STR działają w odwrotną stronę, to znaczy zapisują dane zawarte w rejestrach ogólnego przeznaczenia pod wskazany adres w przestrzeni adresowej. Różne warianty wykorzystania instrukcji LDR i STR zostały przedstawione na listingu 5.11.

Listing 5.11. Warianty użycia rozkazów LDR i STR

```
1: op{type}{cond} Rt, [Rn {, #offset}]
2: op{type}{cond} Rt, [Rn, #offset]!
3: op{type}{cond} Rt, [Rn], #offset
4: opD{cond} Rt, Rt2, [Rn {, #offset}]
5: opD{cond} Rt, Rt2, [Rn, #offset]!
6: opD{cond} Rt, Rt2, [Rn], #offset
7: op{type}{cond} Rt, [Rn, Rm {, LSL #n}]
8: LDR{type}{cond} Rt, label
```

Na listingu zastosowano następujące oznaczenia:

- {} – parametr opcjonalny,
- op – mnemonik LDR lub STR,
- type – opcjonalne parametr określający typ danej:
  - B – bajt bez znaku,
  - SB – bajt ze znakiem,
  - H – półsłowo (16-bitów) bez znaku,
  - SH – półsłowo (16-bitów) ze znakiem,
  - brak – słowo (32-bity),
- cond – opcjonalny skrót od wyrażenia warunkowego (tab. 5.1),
- Rt – rejestr do którego będzie zapisana dana/z którego będzie pobrana dana,
- Rn – rejestr przechowujący adres, do którego/z którego nastąpi zapis/odczyt danej,
- offset – wartość stała służąca do wyliczenia adresu odniesienia poprzez przesunięcie adresu znajdującego się w rejestrze Rn o tą wartość,
- Rt2 – dodatkowy rejestr danych wykorzystywany przy operacjach zapisu/odczytu dwóch danych z pamięci,
- Rm – rejestr z wartością służącą do wyliczenia adresu odniesienia poprzez przesunięcie adresu znajdującego się w rejestrze Rn o wartość znajdującą się w rejestrze Rm,
- LSL #n – opcjonalne przesunięcie bitowe w lewo o ‘n’ pozycji wartości w rejestrze Rm,
- label – etykieta adresu w przestrzeni adresowej względem adresu w PC.

Z powyższego widać, że odczyt i zapis danych z/do przestrzeni adresowej można przeprowadzić wielowariantowo. Zapis pod pozycją 1: jest adresowaniem pośrednim rejestrowym z przesunięciem w postaci natychmiastowej, co oznacza, że w przypadku wykonania rozkazu LDR R0, [R2, #0x04] do rejestru R0 zostanie wpisana 32-bitowa wartość znajdująca się pod adresem przechowywanym w rejestrze R2 powiększonym o 4 bajty. Należy w tym miejscu zaznaczyć, że po wykonaniu rozkazu, wartość w rejestrze R2 (czyli przechowywany w nim adres) nie ulega zmianie.

Wariant zapisany pod pozycją 2: oznacza adresowanie pośrednie rejestrowe preindeksowane wartością w postaci natychmiastowej. Należy rozumieć to w ten sposób, że w przypadku rozkazu LDR R0, [R2, #0x04]! najpierw obliczany jest adres, jako suma

adresu znajdującego się w rejestrze R2 oraz bezpośredniego przesunięcia, czyli 4 bajty, następnie nowy adres jest zapisywany do rejestru R2 a na końcu do rejestru R0 zapisywana jest dana znajdująca się pod nowym adresem zapisanym w rejestrze R2. Wariant z linii 3: jest bardzo podobny do poprzedniego, z tą różnicą, iż w tym przypadku jest to adresowanie postindeksowane, przez co należy rozumieć, że wartość w rejestrze przechowującym adres zmieniana jest dopiero na sam koniec operacji, a odczyt/zapis danej odbywał się przy wykorzystaniu adresu sprzed modyfikacji.
Warianty zapisane w liniach 4:, 5:, 6: listingu 5.11 przedstawiają analogiczną funkcjonalność do wariantów z linii 1:, 2:, 3: z tą różnicą, iż w tym przypadku odczytywane są po dwa słowa i wpisywane do dwóch podanych rejestrów: Rt i Rt2.
Kolejny z wariantów, przedstawiony w linii 7: nazywany jest adresowaniem pośrednim rejestrowym z przesunięciem w postaci rejestru. Podobne jest do adresowania z linii 1:, z tą różnicą, że przesunięcia nie podaje się w postaci stałej wartości, lecz wartości znajdującej się w rejestrze Rm. Inaczej mówią, wynikowy adres z/do którego nastąpi odczyt/zapis jest sumą wartości znajdujących się w rejestrach Rn i Rm.
Linia 8: omawianego listingu przedstawia zapis adresowania pośredniego wykorzystującego etykiety (ang. *label*). Do rejestru Rt zostanie zapisana wartość wskazywana w przestrzeni adresowej etykietą.
Istnieje jeszcze jedna możliwość adresowania w postaci tak zwanych pseudoinstrukcji. Przykłady pseudoinstrukcji zostały przedstawione na listingu 5.12.

Listing 5.12. Przykłady adresowania z wykorzystaniem pseudonistrukcji

```
      LDR R2, =0xF0000000
      LDR R3, =0xF0000001
```

W tym przypadku do rejestru R2 zostanie wstawiona liczba 0xF0000000, a do rejestru R3 liczba 0x0F0000001, natomiast sposób, w jaki zostanie to przetłumaczone bezpośredni rozkaz dla procesora, na etapie pisania programu jest nieznany. Asembler dopiero na etapie asemblacji programu dobiera najodpowiedniejszy sposób na wstawienie danej liczby do rejestru. Na listingu 5.12 przykłady, pomimo iż bardzo podobne zostaną przetłumaczone na inne rozkazy procesora. Dla rejestru R2 będzie to instrukcja przestawiona na listingu 5.13, natomiast dla rejestru R3 będą to instrukcje z listingu 5.14.

Listing 5.13. Instrukcja LDR R2, =0xF0000000 po procesie asemblacji

```
      MOV R2, #0xF0000000
```

Listing 5.14. Instrukcja LDR R3, =0xF0000001 po procesie asemblacji

```
0x08000026 LDR R3, [PC, #36]; @0x0800004C
[…]; tu dalsza część programu
0x0800004C 0001      DCW      0x0001
0x0800004E F000      DCW      0xF000
```

Można zauważyć, że w przypadku wpisywania liczby 0xF0000000 do rejestru R2 asembler zastosował funkcję MOV, gdyż powyższą liczbę można zapisać używając danej 16-bitowej oraz przesunięcia bitowego. Liczby 0xF0000001 wpisywanej do rejestru R3 nie da się jednak zapisać w postaci 16-bitowej + przesunięcie, w związku z czym kompilator zastosował tu inną metodę wprowadzenia tej liczby do rejestru. W sposób automatyczny umieścił na końcu programu pod adresem 0x0800004C wspomnianą wcześniej liczbę, a w programie wynikowym użył funkcji LDR w celu załadowania wartości spod wspomnianego adresu do rejestru R2. Zastosował w tym przypadku adresowanie względem licznika programu PC. Wpisanie adresu do rejestru R2 zajmuje w tym przypadku jeden cykl zegara, a do rejestru R3 dwa cykle.

Przedstawiony sposób obsługi pamięci powoduje, że niektóre operacje należy niestety przeprowadzać w nieco bardziej skomplikowany sposób niż w procesorach

rodziny 8051. Przykładem niech będzie operacja zinkrementowania (powiększenia wartości o 1) danej znajdującej się pod adresem 0x80. W asemblerze procesora 8051 program realizujący powyższe zadanie miałby postać przedstawioną na listingu 5.15.

Listing 5.15. Operacja inkrementacji danej pod adresem 0x80 w procesorach rodziny 8051

```
      INC 80h
```

Jak widać, jest to jeden rozkaz, gdyż procesor ten potrafi operować bezpośrednio na danych w pamięci RAM. W celu uzyskania tego samego efektu w omawianym mikrokontrolerze, należy napisać fragment programu przedstawiony na listingu 5.16.

Listing 5.16. Operacja inkrementacji danej pod adresem 0x80 w procesorach z rdzeniem ARM

```
      LDR   R0, =0x80   ; wpisanie liczby 0x80 do rejestru R0
      LDR   R1, [R0]    ; zapisanie do R1 danej z adresu 0x80
      ADD   R1, R1, #1  ; zwiększenie o 1 danej w rejestrze R1
      STR   R1, [R0]    ; zapisanie pod adres 0x80 danej z rejestru R1
```

Należy jednak pamiętać, iż procesor wyposażony jest w 13 rejestrów ogólnego przeznaczenia, służących właśnie celom zmniejszenia potrzeby sięgania do danych umieszczonych w pamięci SRAM. Pisząc programy bezpośrednio w asemblerze, sposób używania rejestrów R0..R12 i częstość korzystania z funkcji LDR i STR zależy tylko od programisty, natomiast korzystając z języka wyższego poziomu, jakim jest np. język C, to kompilator zajmuje się optymalizacją kodu wynikowego w celu jak najszybszej realizacji zaplanowanych działań. Algorytmy optymalizacji są najważniejszą cechą współczesnych kompilatorów świadcząca o ich jakości.

5.2.2. Instrukcje ogólnego przetwarzania danych

Można powiedzieć, że głównym zadaniem mikrokontrolera, jest przetwarzanie danych. Rozkazy przedstawione w poprzednim rozdziale, wykorzystywane są w celu pobierania i zapisywania danych z przestrzeni adresowe, natomiast do wykonywania operacji arytmetycznych, logicznych, porównań służą rozkazy zestawione w tab. 5.4. Z przedstawionych rozkazów na uwagę zasługują najczęściej wykorzystywane w programach to znaczy instrukcje ADD, SUB, MOV oraz logiczne i bitowe AND, ORR, EOR, natomiast wszystkie rozkazy przedstawione w tab. 5.4 są opisane w instrukcji programowania dostarczanej przez producenta [9].
Rozkaz dodawania ADD jak przedstawiono na listingu 5.9. podobnie jak inne rozkazy może występować w wielu wariantach, uwzględniając mnemoniki od warunkowego wykonywania oraz ilości argumentów. Przykłady wykorzystania instrukcji ADD przedstawiono na listingu 5.17.

Listing 5.17. Przykład użycia rozkazu ADD

```
ADD R0, #3        ; dodanie liczby 3 do rejestru R0 i zapisanie w R0
ADD R0, R0, #3    ; dodanie liczby 3 do rejestru R0 i zapisanie w R0
ADD R5, R4, #0x50 ; dodanie liczby 0x50 do rejestru R4 i zapisanie w R5
ADD R1, R2, R3    ; dodanie wartości rejestrów R2 i R3 i zapisanie w R1
ADD R1, R2        ; dodanie wartości rejestru R2 do R1 i zapisanie w R1
```

W zależności od potrzeb, rozkaz ADD może być dwu lub trójargumentowy, ostatnim argumentem może być rejestr lub wartość stała. Wartość stała nie może jednak przekraczać wartości od 0 do 4095. Należy przy tym pamiętać, że drugi operand zawsze można przed użyciem przesunąć bitowo w lewo lub prawo (tab. 5.2) dzięki czemu w operacji dodawania można korzystać z liczb stałych większych niż 12-bitowych. Analogicznie do rozkazu ADD można potraktować instrukcję SUB, czyli odejmowania. Jej składnia jest identyczna

do składni rozkazu ADD. Przykłady zastosowania instrukcji SUB zostały przedstawione na listingu 5.18.

Listing 5.18. Przykład użycia rozkazu SUB

```
SUB R0, #3        ; odjęcie liczby 3 od rejestru R0 i zapisanie w R0
SUB R5, R4, #0x50 ; odjęcie liczby 0x50 od rejestru R4 i zapisanie w R5
SUB R1, R2, R3    ; odjęcie wartości rejestru R3 od R2 i zapisanie w R1
```

Jak widać, z przedstawionych przykładów, instrukcję SUB wykorzystuje się analogicznie do instrukcji ADD.

Tablica 5.4. Instrukcje ogólnego przetwarzania danych

Skrót	Opis
ADC	Dodawanie z przeniesieniem
ADD	Dodawanie
AND	Logiczne AND
ASR	Przesunięcie bitowe w prawo, z zapełnianiem najbardziej znaczących bitów wartością 31-bitu
BIC	Zerowanie bitu
CLZ	Zliczanie przewodnich zer w rejestrze drugim i wpisywanie wyniku do rejestru pierwszego
CMN	Porównanie poprzez dodawanie dwóch wartości do siebie
CMP	Zwykłe porównanie (poprzez odejmowanie dwóch wartości od siebie)
EOR	Logiczny XOR
LSL	Przesunięcie bitowe w lewo, z zapełnianiem zerami najmniej znaczących bitów
LSR	Przesunięcie bitowe w prawo, z zapełnianiem zerami najbardziej znaczących bitów
MOV	Kopiowanie wartości
MOVT	Kopiowanie wartości stałej 16-bitowej do ważniejszej [31:16] części rejestru
MVN	Kopiowanie wartości po binarnym NOT
ORN	Logiczne nie OR (NOR)
ORR	Logiczne OR
RBIT	Bitowe NOT
REV	Zamiana kolejności bajtów w słowie
ROR	Przesunięcie bitowe w prawo, z zapełnianiem z przenoszeniem najmniej znaczącego bitu na pozycję najbardziej znaczącego
RRX	Przesunięcie bitowe w prawo o jeden bit z zapełnieniem najbardziej znaczącego bitu wartością flagi C (*carry*)
RSB	Odwrotne odejmowanie (operand 2 – operand 1)
SBC	Odejmowanie z przeniesieniem
SUB	Odejmowanie
TEQ	Bitowy XOR – to samo co EORS ale bez zmian w rejestrach
TST	Bitowy AND – to samo co ANDS ale bez zmian w rejestrach

Kolejną z ważnych instrukcji procesora, jest rozkaz MOV. Służy on do przekopiowania wartości z jednego rejestru do drugiego, ewentualnie przekopiowania wartości stałej do rejestru. Przykłady użycia rozkazu MOV zostały przedstawione na listingu 5.19. W przypadku wpisywania wartości stałej, podana wartość nie może być większa od 65535, gdyż liczba ta zapisywana jest na 16-bitach. Opcjonalnie można użyć

jak zawsze w przypadku drugiego operandu przesunięcia bitowego w celu uzyskania większej wartości.

Listing 5.19. Przykład użycia rozkazu MOV

```
MOV R0, #0x123 ; wpisanie liczby 0x123 do rejestru R0
MOV R1, R0     ; wpisanie wartości znajdującej się w rejestrze R0 do R1
```

Instrukcje realizujące funkcje logiczne („i”, „lub”, „nie” oraz inne) na poszczególnych bitach wartości rejestrów są bardzo użyteczne chociażby w przypadku maskowania części rejestrów w celu znalezienia różnic na niektórych bitach. Rozkazem implementującym logiczne „i” jest instrukcja AND, logiczne „lub” jest instrukcja ORR, a logiczną różnicę symetryczną (XOR) jest instrukcja EOR Przykłady ich użycia przedstawiono na listingu 5.20.

Listing 5.20. Przykład użycia rozkazów AND, ORR, EOR

```
AND   R1, R0, #0xFF ; bitowe AND pomiędzy R0 i 0xFF i zapis wyniku do R1
ORR R1, R0, #0xFF   ; bitowe OR pomiędzy R0 i 0xFF i zapis wyniku do R1
EOR R0, #0xFF   ; bitowy XOR pomiędzy R0 i 0xFF i zapis wyniku do R0
```

Operacje te mogą również zmieniać stan flag N i Z w rejestrze PSR procesora. W tym celu należy tak jak w przypadku innych rozkazów zakończyć mnemonik instrukcji literą „S” co należy rozumieć jako ustawienie flag (ang. *Set flags*), to znaczy mnemoniki rozkazów będą się przedstawiły wówczas jako ANDS, ORRS oraz EORS.

Nie mniej ważne są operacje porównania pewnej wartości z inną wartością. Operacje takie służą rozgałęzianiu programu, gdyż w wyniku porównania procesor może zacząć realizować alternatywny ciąg instrukcji. Instrukcją porównania jest na przykład rozkaz CMP. Porównuje on wartości podane jako operandy. Pierwszym operandem może być tylko rejestr, drugim rejestr lub wartość stała. Operacja porównania nie zmienia zawartości rejestrów, wpływa tylko na stan flag N, Z, C i V rejestru PSR procesora. Przykład użycia został przedstawiony na listingu 5.21.

Listing 5.21. Przykład użycia instrukcji porównania CMP

```
CMP R0, #40 ; porównaj zawartość rejestru R0 z liczbą 40
            ; jeżeli są równe ustawiane są flagi Z i C,
            ; jeżeli wartość w R0 > 40 ustawiana jest flaga C,
            ; jeżeli wartość w R0 < 40 ustawiana jest flaga N.
```

Stan tych flag można w dalszej części programu wykorzystać do warunkowego wykonania pewnych funkcji lub do skoku warunkowego. Fragment prostego programu wykorzystującego porównanie został przedstawiony na listingu 5.22.

Listing 5.22. Przykład użycia instrukcji porównania CMP we fragmencie programu

```
CMP R0, #40  ; porównaj zawartość rejestru R0 z liczbą 40
ADDLT R0, #5 ; jeżeli R0 < 40 to dodaj do R0 liczbę 5
SUBGT R0, #10; jeżeli R0 > 40 to odejmij od R0 liczbę 10
```

Instrukcja porównania w przedstawionym fragmencie w zależności od wyniku ustawia flagi N, Z, C i V. Na podstawie stanu tych flag wykonywane lub nie są instrukcje ADD oraz SUB, gdyż zaopatrzone są w odpowiednie mnemoniki wykonania warunkowego LT oraz GT (tab. 5.1). W przypadku gdy wartość w rejestrze R0 jest równa 40, żaden z dwóch kolejnych rozkazów nie jest wykonywany.

5.2.3. Instrukcje mnożenia i dzielenia

Instrukcje mnożenia i dzielenia również można zaliczyć do instrukcji przetwarzania danych, jednak ze względu na budowę procesora zostały wydzielone do osobnej sekcji. Mianowicie rdzeń Cortex-M3 w swoich strukturach posiada jednostkę sprzętowego

mnożenia i dzielenia, która przejmuje na siebie obowiązek wykonywania tych rozkazów. Pomimo tego, operacje takie są i tak dość obciążające procesor, gdyż na ich wykonanie potrzeba kilka okresów zegara systemowego. Instrukcje mnożenia i dzielenia zostały przedstawione w tab. 5.5.

Tablica 5.5. Instrukcje mnożenia i dzielenia

Skrót	Opis
MLA	Mnożenie z dodaniem wyniku do określonego rejestru, wynik 32-bitowy
MLS	Mnożenie z odjęciem wyniku od określonego rejestru, wynik 32-bitowy
MUL	Mnożenie, wynik 32-bitowy
SDIV	Dzielenie ze znakiem
SMLAL	Mnożenie ze znakiem z dodaniem wyniku do określonego rejestru, wynik 64-bitowy
SMULL	Mnożenie ze znakiem z odjęciem wyniku od określonego rejestru, wynik 64-bitowy
UDIV	Dzielenie bez znaku
UMLAL	Mnożenie bez znaku z dodaniem wyniku do określonego rejestru, wynik 64-bitowy
UMULL	Mnożenie bez znaku z odjęciem wyniku od określonego rejestru, wynik 64-bitowy

Dokładny opis powyższych rozkazów został przedstawiony w instrukcji [9] natomiast w niniejszej pracy ze względu na ograniczenia ilościowe zostaną przedstawione przykłady wykorzystania instrukcji mnożenia MUL, MLA oraz dzielenia UDIV. Wspomniane rozkazy zostały przedstawione na listingu 5.23.

Listing 5.23. Przykład użycia rozkazów MUL, MLA oraz UDIV

```
MUL   R5, R6, R7        ; wymnożenie wartości z R6 i R7 i zapisanie w R5
MUL   R6, R7            ; wymnożenie wartości z R6 i R7 i zapisanie w R6
MLA   R1, R2, R3, R4    ; wymnożenie wartości z R2 i R3, dodanie wartości
                        ; z R4 i zapisanie w R1
UDIV  R8, R9, R10       ; podzielenie wartości z R9 przez R10 i zapisanie
                        ; wyniku w R8
```

Co ciekawe, w instrukcjach tych argumentami mogą być tylko rejestry, nie można podać do nich wartości stałej tak jak w przypadku instrukcji ADD czy MOV.

5.2.4. Instrukcje nasycania

W liście rozkazów omawianego mikrokontrolera występują dwie instrukcje nasycania: SSAT oraz USAT. Ich zestawienie zostało przedstawione w tab. 5.6.

Tablica 5.6. Instrukcje nasycania

Skrót	Opis
SSAT	Nasycanie ze znakiem
USAT	Nasycanie bez znaku

Przez pojęcie nasycania należy rozumieć ograniczenie wartości poprzez podanie liczby bitów nasycenia, to znaczy ograniczenie wartości poprzez podanie maksymalnej liczby bitów, na których liczba może zostać zapisana. Składnia polecenia została przedstawiona na listingu 5.24.

Listing 5.24. Składnia instrukcji SSAT oraz USAT

```
op{cond} Rd, #n, Rm {, shift #s}
```

W listing zastosowano następujące oznaczenia:

- {} – parametr opcjonalny,
- op – mnemonik SSAT lub USAT,

- cond – opcjonalny skrót od wyrażenia warunkowego (tab. 5.1),
- Rd – rejestr do którego będzie zapisana wartość po saturacji,
- #n – liczba bitów nasycenia ,
- Rm – rejestr z wartością na której będzie przeprowadzane nasycanie,
- shift #s – opcjonalne przesunięcie bitowe rejestru Rm przed nasyceniem.

Zasada działania instrukcji USAT zostanie przedstawiona na przykładzie przedstawionym na listingu 5.25.

Listing 5.25. Przykłady wykorzystania instrukcji USAT

```
MOV R0, #0x012F   ; wpisanie do R0 wartości 0x012F
USAT R1, #16, R0  ; nasycenie R0 do 16 bitów i wpisanie nasyconej
                  ; wartości do rejestru R1. W tym przypadku do R1
                  ; zostanie wpisana niezmieniona wartość znajdująca
                  ; się w R0, to znaczy 0x012F, gdyż zapisana jest na
                  ; dziewięciu bitach, a to nie jest więcej niż 16 bitów
USAT R1, #9, R0   ; nasycenie R0 do 9 bitów i wpisanie nasyconej
                  ; wartości do rejestru R1. W tym przypadku do R1
                  ; zostanie wpisana niezmieniona wartość znajdująca
                  ; się w R0, to znaczy 0x012F, gdyż zapisana jest na
                  ; dziewięciu bitach, a to nie jest więcej niż 9 bitów
USAT R1, #7, R0   ; nasycenie R0 do 7 bitów i wpisanie nasyconej
                  ; wartości do rejestru R1. W tym przypadku do R1
                  ; zostanie wpisana maksymalna wartość jaką można
                  ; zapisać na 7 bitach, to znaczy 0x007F gdyż wartość
                  ; w rejestrze R0 jest zapisana na większej ilości bitów
                  ; niż podane 7 bitów nasycenia.
```

Można zatem powiedzieć, że instrukcje nasycania są pewnego rodzaju ograniczeniem na maksymalną wartość. Oprócz powyższego funkcje zmieniają również stan flagi Q w rejestrze PSR procesora. W przypadku, gdy wartość przed nasycaniem i po nie zmieniła się, flaga Q pozostaje nieustawiona. Natomiast, gdy nasycenie spowoduje zmianę wartości, to flaga Q zostaje ustawiona. W celu wyzerowania flagi Q należy wykorzystać instrukcję MSR.

5.2.5. Instrukcje rozgałęzień

Bardzo ważne w trakcie wykonywania programu są instrukcje rozgałęzień, inaczej mówiąc skoków warunkowych, czy bezwarunkowych.

Tablica 5.7. Instrukcje rozgałęzień

Skrót	Opis
B	Skok
BL	Skok z linkowaniem
BLX	Skok niebezpośredni z linkowaniem
BX	Skok niebezpośredni
CBNZ	Porównanie i skok w przypadku ≠ 0
CBZ	Porównanie i skok w przypadku = 0
IT	Blok instrukcji „Jeżeli – to”

Zestawienie instrukcji rozgałęzień zostało przedstawione w tab. 5.7. Składnia instrukcji skoków B, BL, BLX oraz BX została przedstawiona na listingu 5.26. W trakcie wykonywania programu procesor napotyka na pewne instrukcje skoków warunkowych, od którego to warunku zależy czy realizowany będzie kod kolejnej instrukcji, czy instrukcji spod innego adresu. Takie sytuacje zdarzają się bardzo często, na przykład procesor

sprawdza wartość reprezentującą napięcie na wejściu analogowym i w zależności od przekroczenia lub nie danego progu wykonuje lub nie dany fragment programu (na przykład załączającego sygnalizację alarmową). Domniema się, że w programach będących wynikiem działania kompilatora, instrukcje skoków pojawiają się co około kilkanaście rozkazów.

Listing 5.26. Składnia instrukcji B, BL, BLX oraz BX

```
B{cond} label
BL{cond} label
BX{cond} Rm
BLX{cond} Rm
```

W listing zastosowano następujące oznaczenia:

- {} – parametr opcjonalny,
- cond – opcjonalny skrót od wyrażenia warunkowego (tab. 5.1),
- label – etykieta występująca w kodzie programu, pod którą nastąpi skok,
- Rm – rejestr zawierający adres skoku.

Instrukcje B oraz BL wykonują skok do adresu oznaczonego etykietą label, natomiast rozkazy BX oraz BLX pod adres, który jest przechowywany w podanym rejestrze. W przypadku pierwszym na etapie asemblacji wyznaczany jest adres względem licznika PC pierwszej instrukcji znajdującej się pod etykietą label i taki też jest wpisywany jako parametr wywoływania rozkazu B. W przypadku instrukcji skoków niebezpośrednich BX oraz BLX adres jest pobierany w trakcie procesu wykonywania z podanego rejestru, na etapie kompilacji nie można przewidzieć jaki adres znajdzie się w rejestrze. Mając na uwadze układ predykcji skoku w procesorze Cortex-M3 należy pamiętać, że instrukcja skoku niebezpośredniego BX lub BLX jest najbardziej kosztowną ze względu na czas procesora instrukcją rozgałęzienia. Układ predykcji nie jest w stanie przewidzieć do jakiego adresu nastąpi skok, gdyż dopiero całkowite wykonanie instrukcji pozwala określić ten adres. Przy podawaniu adresów dla instrukcji skoków BX oraz BLX należy również pamiętać o tym, że w adresach tych najmniej znaczący bit powinien być ustawiony, to znaczy jeżeli ma nastąpić skok pod adres 0x08000034, to do rejestru, który będzie wykorzystywany w instrukcji skoku należy wpisać wartość 0x08000035. Ustawienie tego bitu informuje procesor, że instrukcje zaczynające się od tego adresu są z listy instrukcji Thumb.

Instrukcje skoków z linkowaniem BL oraz BLX powodują zapisanie adresu kolejnej instrukcji po BL lub BLX w rejestrze LR. W takim wypadku, program ma możliwość powrotu po wykonaniu pewnego ciągu instrukcji w innym miejscu programu do miejsca z którego nastąpił skok i kontynuowania dalszej części programu. Taka funkcjonalność jest przydatna w przypadku wykorzystywania w programie tak zwanych podprogramów. Pewne czynności, które są wielokrotnie powtarzane w trakcie wykonywania programu, wydziela się w postaci podprogramu, a w głównym kodzie wstawia się tylko w odpowiednich miejscach instrukcje skoku do danego podprogramu. Dzięki takiemu rozwiązaniu zmniejsza się ilość miejsca zajmowanego przez program dzięki wyeliminowaniu powtarzających się fragmentów kodu przy niestety ale nieznacznym spadku wydajności, gdyż instrukcje skoku i później skoku powrotnego do programu głównego są tu elementami, na które procesor przeznacza dodatkowy czas. Przykład prostego fragmentu programu wykorzystującego podprogram został przedstawiony na listingu 5.27. Przedstawiony fragment programu ten nie posiada żadnych walorów z punktu widzenia funkcjonalności, służy tylko w celu zaprezentowania zasady działania instrukcji skoku i wykorzystania podprogramów. Widać, że podprogram, który

realizuje przywrócenie pewnych domyślnych wartości w rejestrach R0, R1 i R2 został wywołany dwa razy w programie głównym poprzez wykorzystanie instrukcji BL.

Listing 5.27. Przykład wykorzystania podprogramu

```
      BL ustaw_rejestry       ; skok do pierwszej instrukcji pod etykietą
                              ; ustaw_rejestry, czyli do instrukcji
                              ; LDR R0, =2016
      CMP R1, R5              ; po wykonaniu wszystkich instrukcji z
                              ; podprogramu, procesor realizuje kolejne
                              ; rozkazy od tego miejsca, czyli porównanie
                              ; rejestru R1 z R5
      ADD R1, R7              ; dodanie do R1 wartości R7 i wpisanie do
                              ; R1
      SUB R1, R2              ; odjęcie od R1 wartości R2 i wpisanie do
                              ; R1
      BL ustaw_rejestry       ; ponownie skok do pierwszej instrukcji pod
                              ; etykietą ustaw_rejestry, czyli do
                              ; instrukcji LDR R0, =2016
      ADD R2, #4              ; po wykonaniu wszystkich instrukcji z
                              ; podprogramu, procesor realizuje kolejne
                              ; rozkazy od tego miejsca, czyli dodanie do
                              ; R2 liczby 4 i zapisanie w R2
      (...)                   ; tu kolejne instrukcje programu głównego

ustaw_rejestry                ; etykieta, którą podaje się w rozkazie BL
      LDR R0, =2016           ; wpisanie do R0 wartości 2016
      LDR R1, =2025           ; wpisanie do R1 wartości 2025
      LDR R2, =2178           ; wpisanie do R2 wartości 2178
      BX LR                   ; skok do miejsca, z którego podprogram
                              ; został wywołany
```

Oprócz powyższych lista instrukcji Thumb-2 zawiera także instrukcje skoków warunkowych CBNZ i CBZ. Pierwszy z nich dokonuje skoku pod wskazaną etykietę, w przypadku gdy podany rejestr jest różny od zera, drugi dokonuje skoku w przypadku, gdy wartość w podanym rejestrze jest równa zero. Przykład wykorzystania instrukcji CBZ został przedstawiony na listingu 5.28.

Listing 5.28. Przykład wykorzystania rozkazu CBNZ

```
      MOV R4, #5
zmniejsz
      CBZ R4, dalej
      SUB R4, #1
      B zmniejsz
dalej
      MOV R6, R5
```

W podanym przykładzie fragment pomiędzy etykietami „zmniejsz", a „dalej" wykona się pięć razy, aż wartość w rejestrze R0 będzie wynosiła 0, wówczas nastąpi skok do etykiety „dalej" i wykonanie kolejnych instrukcji z kodu programu.

Do instrukcji rozgałęzień został także włączony blok instrukcji wykonania warunkowego IT, którego przykłady zostały przedstawione na listingu 5.7.

5.2.6. Instrukcje pozostałe

Lista instrukcji procesora zawiera także rozkazy niezwiązane bezpośrednio z przetwarzaniem danych czy rozgałęzieniami. W tab. 5.8. zestawiono pozostałe instrukcje procesora. Wśród zaprezentowanych różnych instrukcji, najprostszą jest instrukcja NOP, której wykonanie nie wprowadza żadnych zmian. Natomiast jak każda instrukcja, jej

wykonanie można uzależnić od stanu flag w rejestrze PSR, poprzez uzupełnienie skrótu NOP o mnemonik wykonania warunkowego (tab. 5.1).

Tablica 5.8. Instrukcje różne

Skrót	Opis
CPSID	Zmiana stanu procesora, wyłączenie przerwań
CPSIE	Zmiana stanu procesora, załączenie przerwań
MRS	Kopiowanie z rejestru specjalnego do rejestru ogólnego przeznaczenia
MSR	Kopiowanie z rejestru ogólnego przeznaczenia do rejestru specjalnego
NOP	Instrukcja pusta, nic nie rób
WFE	Przejście do trybu ograniczonego poboru mocy, wybudzenie poprzez zdarzenie
WFI	Przejście do trybu ograniczonego poboru mocy, wybudzenie poprzez przerwanie

Instrukcje WFE i WFI służą do wprowadzenia mikrokontrolera w stan ograniczonego poboru mocy, których dokładny opis znajduje się w rozdziale 4.8 „Tryby oszczędzania energii".

Na uwagę zasługują natomiast rozkazy MRS oraz MSR. Do rejestrów specjalnych procesora, to znaczy: APSR, IPSR, EPSR, IEPSR, IAPSR, EAPSR, PSR, MSP, PSP, PRIMASK, BASEPRI, BASEPRI_MAX, FAULTMASK i CONTROL nie można uzyskać dostępu poprzez zwykłe polecenia MOV, czy LDR/STR. W celu odczytania lub zapisania wartości należy korzystać z rozkazów MRS i MSR. Przykłady wykorzystania powyższych instrukcji przedstawiony został na listingu 5.29.

Listing 5.29. Przykład wykorzystania rozkazów MRS i MSR

```
MRS R0, CONTROL
ORR R0, #1
MSR   CONTROL, R0
```

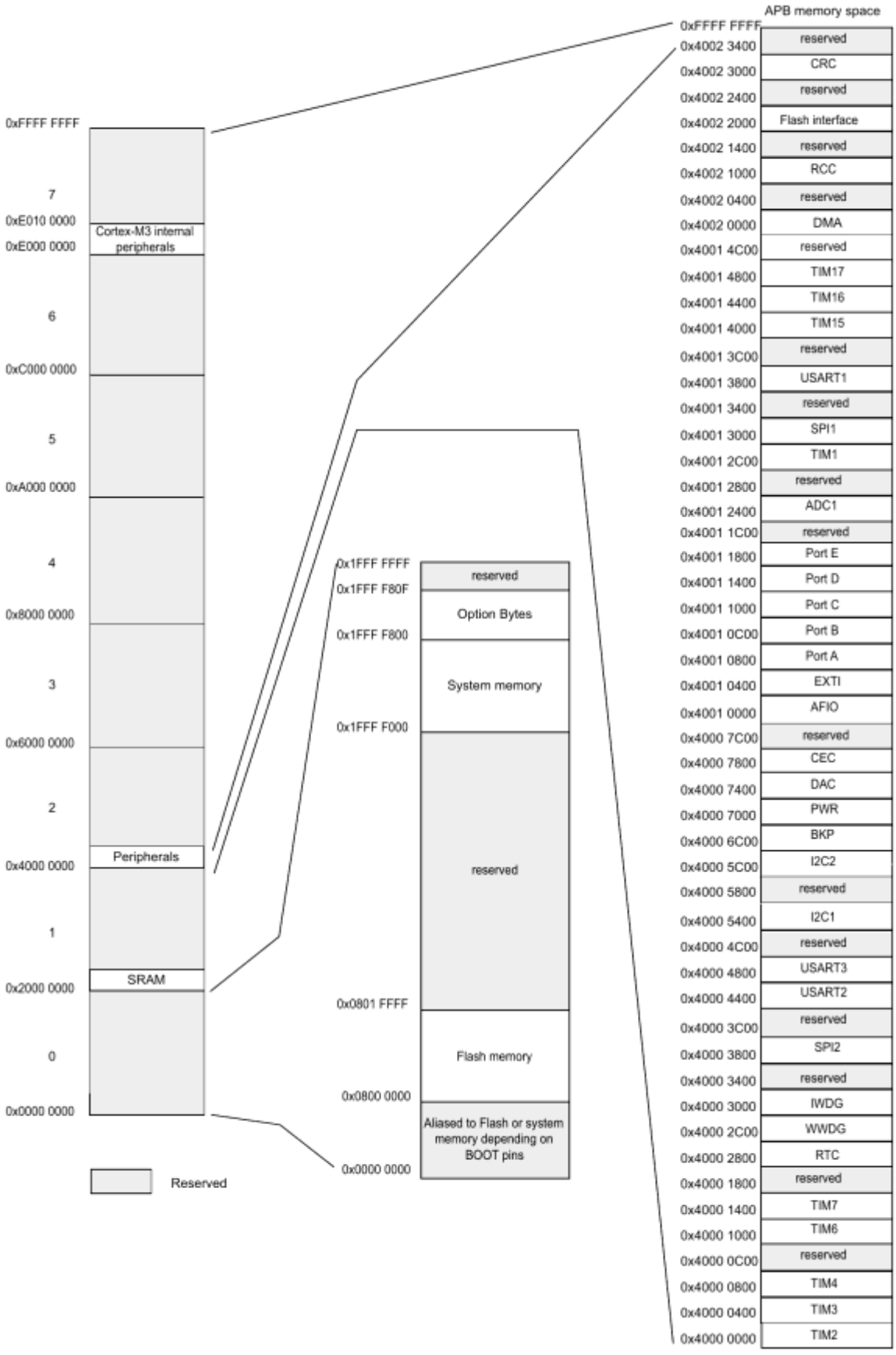
W przykładzie tym wartość rejestru specjalnego CONTROL została pobrana do rejestru R0, następnie w rejestrze tym wykorzystują instrukcję bitowego OR ustawiony bit 0 na wartość 1, a następnie tak zmienioną wartość wpisano ponownie do rejestru CONTROL, co przyniosło skutek w postaci przełączenia trybu pracy procesora na tryb użytkownika (nieuprzywilejowany), o którym więcej w rozdziale 5.4. „Tryby pracy rdzenia Cortex-M3".

Przedstawiona lista instrukcji, zawiera jedynie fragment pełnej listy rozkazów Thumb-2. Wszystkie przedstawione są i opisane w dokumentacji producenta [9].

5.3. Organizacja pamięci

Omawiany mikrokontroler posiada 32-bitową magistralę adresową, to znaczy może zaadresować 2^{32} bajtów pamięci, czyli 4 GB. Przestrzeń adresowa podzielona jest na obszary ze względu na różne fizyczne elementy widoczne pod danymi adresami. Jako, iż mikrokontroler wykonany jest w architekturze harwardzkiej, do przesyłania danych spod różnych przestrzeni adresowych wykorzystywane są różne magistrale danych. W mikrokontrolerze STM32F100RBT6B większość przestrzeni adresowej jest niewykorzystywana, gdyż nie posiada on takiej ilości zainstalowanej pamięci, ani tak znaczącej liczby urządzeń peryferyjnych. Mapa pamięci omawianego mikrokontrolera została przedstawiona na rys. 5.1. Obszar początkowy o adresach od 0x0000 0000 do 0x07FF FFFF nie jest przypisany na stałe do żadnego fizycznego elementu, w zależności od stanu wyprowadzeń BOOT0 oraz BOOT1 mikrokontrolera, obszar ten jest aliasowany albo do początku pamięci Flash (adres 0x0800 0000) albo do pamięci systemowej (ang.

System memory – adres od 0x1FFF F000), która fizycznie również znajduje się w pamięci Flash.



Rys. 5.1. Mapa pamięci mikrokontrolera STM32F100RBT6B, opracowano wg [17]

Ustawienie wyprowadzeń BOOT0 oraz BOOT1 w stan wysoki powoduje uruchomienie mikrokontrolera z pamięci SRAM (adres od 0x2000 0000), przy czym w tym wypadku obszar od 0x0000 0000 nie jest aliasowany do obszaru pamięci SRAM. Przez pojęcie aliasowania należy rozumieć odwoływanie się pod odpowiedni adres, który ma umiejscowienie w fizycznym elemencie poprzez używanie adresu symbolicznego. Zatem, w przypadku pierwszym jeżeli program odwołuje się na przykład do rejestru 0x0000 0004 to tak naprawdę odwołuje się do rejestru 0x0800 0004, przy czym w dalszym ciągu do tych samych danych można uzyskać dostęp odwołując się bezpośrednio do adresu 0x0800 0004. Nie jest to jednak skopiowanie zawartości pamięci tylko utworzenie swoistego linku, który wskazuje na odpowiednią pamięć. W tab. 5.9 zestawiono możliwe konfiguracje w zależności od stanu wyprowadzeń BOOT0 oraz BOOT1.

Tablica 5.9. Tryby rozruchu

BOOT1	BOOT0	Start do adresu	Rodzaj pamięci	Wykorzystywane magistrale
x	0	0x0800 0000	Flash	ICode, DCode bus
0	1	0x1FFF F000	Pamięc systemowa (też w pamięci Flash)	ICode, DCode bus
1	1	0x2000 0000	SRAM	System bus

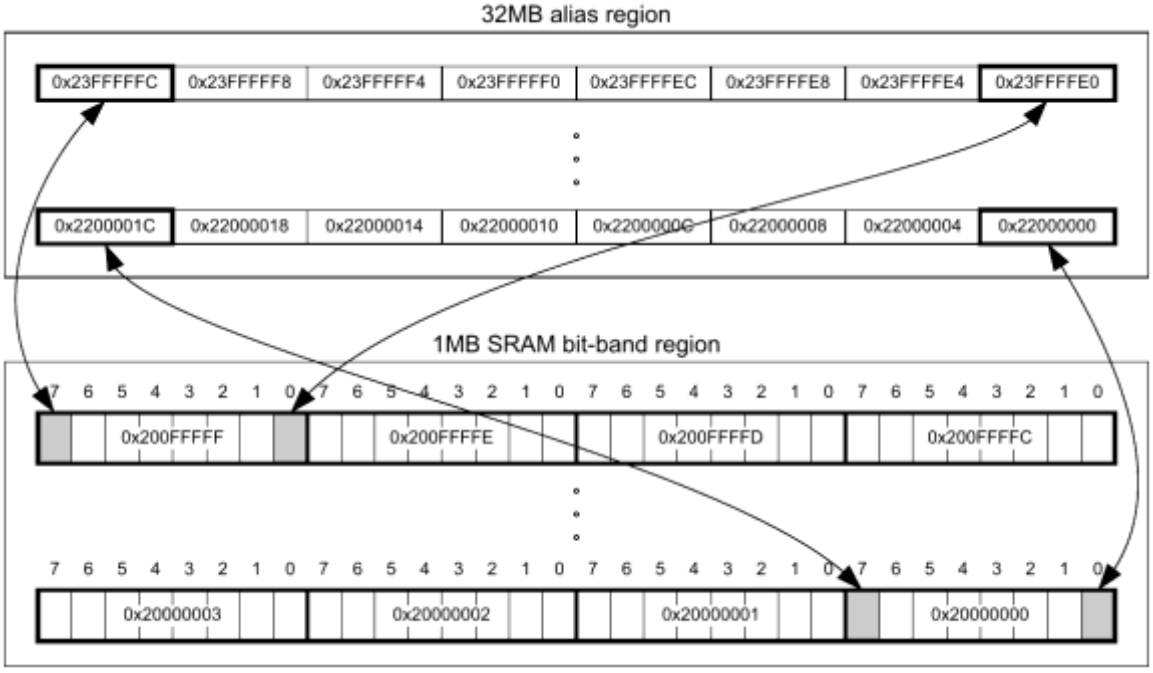
Stan pinów BOOT0 oraz BOOT1 jest zatrzaskiwany w układzie przy pojawieniu się czwartego narastającego zbocza sygnału SYSCLK po wystąpieniu resetu. Standardowo, zestaw ewaluacyjny ma wyprowadzenie BOOT0 zwarte do masy, co odpowiada dowiązaniu adresu 0x0000 0000 do początku pamięci Flash o adresie 0x0800 0000. Czytanie rozkazów z tej pamięci odbywa się poprzez magistralę ICode, natomiast odczyt i ewentualny zapis danych poprzez magistralę DCode. W przypadku używania pamięci systemowej używane są dokładnie te same magistrale, natomiast w przypadku uruchamiania z pamięci SRAM zarówno rozkazy jak i dane odczytywane są poprzez magistralę System bus. Najczęściej wykorzystywana jest dowiązanie do pamięci Flash, gdyż zazwyczaj tam zapisywany jest kod maszynowy programu. Uruchamianie z pamięci SRAM to autorskie rozwiązanie inżynierów z firmy STMicroelectronics, które może się okazać przydatne na przykład przy wielokrotnym zautomatyzowanym testowaniu różnych wariantów działania programu, co wymaga dużych ilości przeprogramowań układu, co z kolei mogłoby przekroczyć maksymalną liczbę zapisów do pamięci Flash. Pamięć systemowa, to specjalnie wydzielona przestrzeń w pamięci Flash, w której producent umieścił swój program rozruchowy (ang. *boot loader*), który służy do przeprogramowywania mikroprocesora z wykorzystaniem portu szeregowego USART1. W badanym zestawie, nie został on wykorzystany.

Od adresu 0x0800 0000 widoczna jest fizyczna pamięć Flash. W trybie rozruchu, który jest wykorzystywany w omawianym układzie w tej pamięci umieszczony jest program, który realizuje mikrokontroler. Mikrokontroler po sygnale reset odczytuje najpierw wartość spod adresu 0x0800 0000 i zapisuje ją do rejestru wskaźnika stosu SP. Pod adresem 0x0800 0000 musi się zatem znaleźć adres szczytu stosu, gdyż tak on będzie zinterpretowany przez mikrokontroler. Kolejnym etapem jest odczytanie wartości z rejestru 0x0800 0004, który jest wektorem reset. Pod tym adresem znajduje się adres, do którego nastąpi skok w celu rozpoczęcia czytania kolejnych rozkazów kodu programu.

Pojemność zainstalowanej w mikrokontrolerze pamięci Flash wynosi 128 kB, a przestrzeń ta podzielona jest na 128 stron o wielkości 1 kB. Zerowa strona zaczyna się od adresu 0x0800 0000, a kończy na adresie 0x0800 03FF, pierwsza strona zaczyna się od adresu 0x0800 0400, a kończy na adresie 0x0800 07FF, i tak dalej, aż do stodwudziestejsiódmej strony, która zaczyna się od adresu 0x0801 FC00 i kończy na adresie 0x0801 FFFF. Przestrzeń adresowa z zakresu od 0x0802 0000 do 0x1FFF EFFF

jest w omawianym mikrokontrolerze nieużywana. W przypadku mikrokontrolerów wyposażanych w większą ilość pamięci Flash jest ona umieszczana w kolejnych adresach z powyższej puli. Począwszy od adresu 0x1FFF F000 zaczyna się pamięć systemowa. Jest to obszar pamięci o rozmiarze 2 kB, a zatem kończący się na adresie 0x1FFF F7FF umieszczony w pamięci także w pamięci Flash, lecz nie w jej głównym obszarze. Jak wspomniano wcześniej, w obszarze tym znajduje się kod programu rozruchowego, który może realizować zadanie programowania głównej pamięci Flash poprzez interfejs USART1. Za tym obszarem w pamięci Flash znajduje się jeszcze pole o rozmiarze 16 B z tak zwanymi bajtami konfiguracyjnymi (ang. *Option bytes*). W rejestrach tych zapisana jest najbardziej podstawowa konfiguracja mikrokontrolera, zawierająca między innymi zabezpieczenia przed możliwością zapisu, odczytu, skasowania wybranych stron pamięci Flash. Za bajtami konfiguracyjnymi znajduje się przestrzeń adresowa, która jest niewykorzystywana w omawianym mikrokontrolerze i kończy się na adresie 0x1FFF FFFF. Generalnie, rdzeń Cortex-M3 pierwsze 0,5 GB przestrzeni adresowej traktuje jako przestrzeń programu (ang. *Code*), natomiast ile z tej przestrzeni zostanie rzeczywiście wykorzystane, zależy od producenta danego mikrokontrolera.

Począwszy od adresu 0x2000 0000 w układzie widoczna jest wewnętrzna pamięć operacyjna SRAM. W omawianym mikrokontrolerze znajduje się 8 kB tej pamięci, zatem ostatni jej bajt znajduje się pod adresem 0x2000 2000, przy czym rdzeń Cortex-M3 również rezerwuje sobie kolejne 0,5 GB na wewnętrzną pamięć operacyjną, a stopień jej wykorzystania zależy od producenta. Oprócz standardowych zadań, do których wykorzystywana jest pamięć operacyjna, mikrokontroler oferuje możliwość manipulacji poszczególnymi bitami w bajtach pamięci (ang. *bit band operation*).



Rys. 5.2. Schemat zasady działania bezpośredniego dostępu do pojedynczych bitów w pamięci SRAM, opracowano wg [12]

Pierwszy megabajt wewnętrznej pamięci SRAM (czyli w przypadku omawianego mikrokontrolera cała zawarta w nim pamięć) jest aliasowany i dostępny także pod adresami z zakresu od 0x2200 0000 do 0x23FF FFFF. W przypadku korzystania z bezpośrednich adresów pierwszego megabajtu pamięci, czyli z adresów od 0x2000 0000 do 0x2010 0000 program ma dostęp do danych znajdujących się pod tymi adresami poprzez odczyty/zapisy 8-, 16- lub 32-bitowych słów. W przypadku korzystania z adresów aliasowanych, program ma dostęp do odczytu i zapisu pojedynczych bitów. Jak można

zauważyć alias posiada rozmiar 32 MB, czyli 32 razy więcej niż przestrzeń aliasowana. Każdy bowiem pojedynczy bit z przestrzeni 0x2000 0000 – 0x2010 0000 jest odwzorowywany poprzez 4-bajtowe słowo w przestrzeni 0x2200 0000 – 0x23FF FFFF. Schemat ideowy takiego rozwiązania został przedstawiony na rys. 5.2. Na ten przykład niech w pamięci pod adresem 0x2000 0001 znajduje się jednobajtowa zmienna, w której należy zmienić bit nr 2 na wartość 1. W klasycznym wykonaniu, należy odczytać spod adresu 0x2000 0000 cały bajt, następnie przeprowadzić operację sumy logicznej OR z liczbą 0x04 i na końcu zapisać z powrotem pod ten sam adres zmienioną wartość. Korzystając z opcji manipulacji poszczególnymi bitami, można proces przeprowadzić szybciej, to znaczy przeliczyć pod jakim adresem w aliasie znajduje wskazany bit i zapisać do niego wartość 1. Bit nr 3 w bajcie pod adresem 0x2000 0000 ma odwzorowanie pod adresem 0x2200 000C, zatem tą samą czynność co wyżej opisaną uzyskuje się poprzez zapisanie wartości 1 pod adres 0x2200 000C. Wyliczenie adresu danego bitu w aliasie można przeprowadzić korzystając ze wzoru (5.1).

$$adres\_alias = ((adres\_bajtu - 0x2000\ 0000) \times 32 + numer\_bitu \times 4) + 0x2200\ 0000 \quad (5.1)$$

Dla omawianego przykładu będzie to wartość przedstawiona na (5.2).

$$((0x2000\ 0000 - 0x2000\ 0000) \times 32 + 3 \times 4) + 0x2200\ 0000 = 0x2200\ 000C \quad (5.2)$$

Manipulowanie poszczególnymi bitami w danych zawartych w pamięci operacyjnej SRAM wprowadza możliwości do optymalizacji kod pod względem ilości operacji przesyłania danych z i do pamięci, co z kolei wpływa na szybkość wykonywanego kodu programu.

Za pamięcią SRAM w przestrzeni adresowej rozciąga się nieużywany obszar, aż do adresu 0x4000 0000. Począwszy od tego adresu widoczne są poszczególne urządzenia peryferyjne omawianego mikrokontrolera przyłączone do magistral APB1 i APB2. Pierwszym widocznym urządzeniem jest timer TIM2, jego rejestry konfiguracyjne umieszczone są pod adresami od 0x4000 0000 do 0x4000 0400. Przyznane ma więc 1024 bajty, czyli 1 kB na rejestry konfiguracyjne. Każde z urządzeń peryferyjnych jest widoczne przez 1024 bajty przestrzeni adresowej, po których widoczne być zaczyna kolejne urządzenie. Ostatnim widocznym urządzeniem jest sprzętowa jednostka obliczania sumy kontrolnej CRC, która widoczna jest w zakresie adresów od 0x4002 3000 do 0x4002 33FF. Tak jak w przypadkach poprzednich, sam rdzeń rezerwuje sobie na urządzenia peryferyjne większą ilość przestrzeni adresowej, która wynosi 0,5 GB więc kończy się na adresie 0x5FFF FFFF, jednak w omawianym mikrokontrolerze nie jest ona w całości wykorzystywana. Podobnie jak w przypadku pamięci SRAM, również i w przypadku urządzeń peryferyjnych, pierwszy megabajt pamięci jest aliasowany i dostępny w przestrzeni, gdzie można manipulować poszczególnymi bitami. Przestrzeń adresowa od 0x4000 0000 do 0x4010 0000 jest dostępna w postaci odwzorowania poszczególnych bitów pod adresem od 0x4200 0000 do 0x43FF FFFF. W mikrokontrolerze STM32F100RBT6B rejestry konfiguracyjne wszystkich urządzeń peryferyjnych mieszczą się w pierwszym megabajcie przestrzeni adresowej przewidzianej dla urządzeń peryferyjnych, w związku z czym dostęp do rejestrów urządzeń jest także możliwy poprzez zmianę pojedynczych bitów.

Za przestrzenią przewidzianą dla urządzeń peryferyjnych, znajduje się obszar od adresu 0x6000 0000 do 0x9FFF FFFF (1 GB) przewidziany dla zewnętrznej pamięci RAM, której w omawianym mikrokontrolerze nie można wykorzystać, więc przestrzeń ta jest nieużywana. Kolejnym obszarem, jest znów 1 GB przestrzeni adresowej od 0xA000 000 do 0xDFFF FFFF przewidzianego dla zewnętrznych urządzeń peryferyjnych, której także w prezentowanym układzie nie można wykorzystać.

Począwszy od adresu 0xE000 0000, a kończąc na 0xE00F FFFF widoczne są wewnętrzne urządzenia peryferyjne rdzenia Cortex-M3. Komunikacja rdzenia z wewnętrznymi urządzeniami peryferyjnymi odbywa się poprzez tak zwaną Prywatną Magistralę Urządzeń Peryferyjnych (ang. *Private Peripheral Bus*), która fizycznie jest magistralą typu APB. Poprzez tą magistralę możliwy jest dostęp do urządzeń peryferyjnych rdzenia takich jak sterownik przerwań NVIC oraz różnych jednostek debugera.

Ostatni fragment przestrzeni adresowej, to znaczy od adresu 0xE010 0000 do 0xFFFF FFFF jest w omawianym mikrokontrolerze nieużywany, natomiast specyfikacja architektury umożliwia w tym miejscu umieszczanie ewentualnych urządzeń dodanych przez producenta mikrokontrolera przyłączonych do prywatnej magistrali APB.

5.4. Tryby pracy rdzenia Cortex-M3

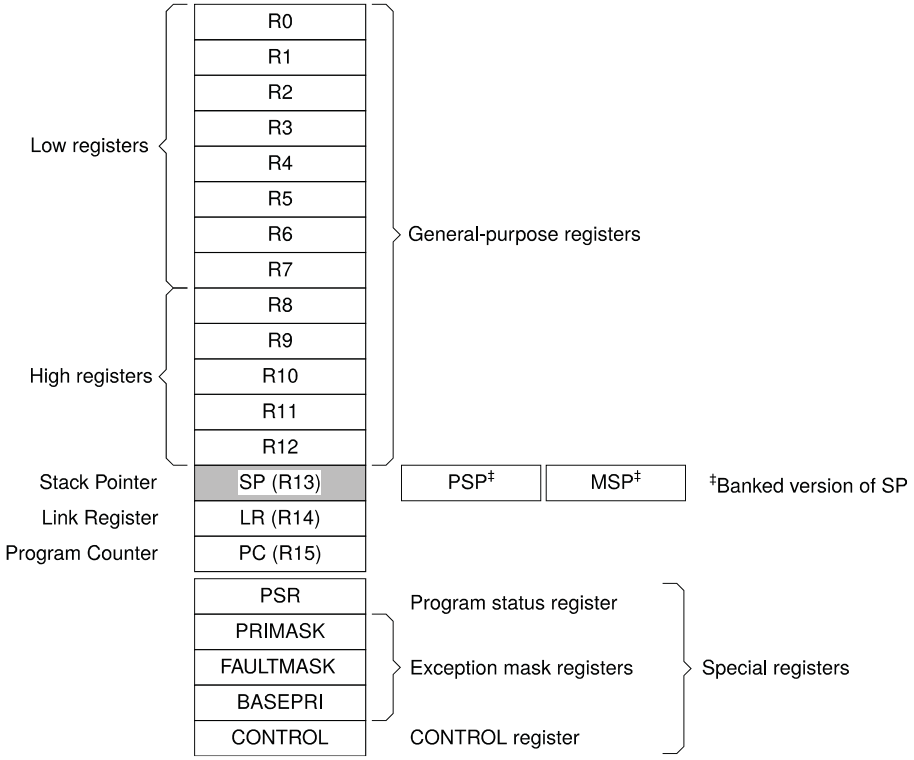
Procesor wspiera dwa tryby pracy: tryb wykonywania programu (ang. *Thread mode*) oraz tryb obsługi przerwania (ang. *Handler mode*). Tryb wykonywania program jest uruchamiany po otrzymaniu sygnału reset i trwa do momentu przerwania. Przerwanie powoduje przejście procesora w tryb obsługi przerwania, natomiast po jego zakończeniu powrót do trybu wykonywania programu. Podział ten jest istotny, ze względu na dwa poziomy praw dostępu do zasobów mikrokontrolera. Kod maszynowy zapisany w pamięci programu może być wykonywany jako uprzywilejowany (ang. *Privileged access*) lub użytkownika (ang. *Unprivileged / User access*). Kod programu wykonywany w trybie uprzywilejowanym ma dostęp do wszystkich zasobów mikrokontrolera, natomiast w trybie użytkownika dostęp jest ograniczony (na przykład nie są dostępne rejestry kontrolera NVIC). W trybie obsługi przerwania, kod jest zawsze wykonywany jako uprzywilejowany, natomiast w trybie wykonywania programu, może być wykonywany jako uprzywilejowany lub nie. Po otrzymaniu sygnału reset kod wykonywany jest w trybie uprzywilejowanym. Przejście do trybu użytkownika możliwe jest poprzez ustawienie bitu nr 0 na wartość 1 rejestrze specjalnym CONTROL wykorzystując rozkaz MSR. Od tej chwili, program będzie wykonywany z prawami użytkownika. W celu powrotu do trybu uprzywilejowanego w trybie wykonywania programu należy w trybie obsługi przerwania ustawić bit 0 rejestru CONTROL na wartość 0, co spowoduje przy powrocie do trybu wykonywania programu pozostanie w trybie uprzywilejowanym. W zależności od trybu pracy i poziomu dostępu różnią się niektóre elementy opisywane w kolejnych rozdziałach, należy zatem pamiętać o ich obecności. Wykorzystywane są głównie w systemach pracujących pod kontrolą systemu operacyjnego, ale nic nie stoi na przeszkodzie, żeby wykorzystać je w aplikacjach bez systemów operacyjnych.

5.5. Rejestry procesora

Rdzeń Cortex-M3 wyposażony jest w 21 32-bitowych rejestrów wewnętrznych, które zostały symbolicznie przedstawione na rys. 5.3. Rejestry oznaczone jako R0…R12 są rejestrami ogólnego przeznaczenia. Na nich przeprowadzane są operacje arytmetyczne, logiczne, porównywania i inne przeprowadzane przez procesor, do nich też zapisywane są dane z pamięci SRAM lub urządzeń peryferyjnych w przypadku gdy zachodzi potrzeba przeprowadzenia operacji na danych znajdujących się w tych obszarach. Z rejestrów tych następuje również zapisywanie wartości pod wskazany adres w przestrzeni adresowej pamięci SRAM czy peryferii. Z rejestrów R0…R7 mogą korzystać rozkazy zarówno 16- jak i 32-bitowe, natomiast z rejestrów R8…R12 tylko rozkazy 32-bitowe, gdyż w rozkazach 16-bitowych numer rejestru kodowany jest na trzech bitach.

Rejestr R13 jest rejestrem wskaźnika stosu (ang. *Stack Pointer – SP*). Jego adres wskazuje na pierwszy wolny bajt na stosie. Podczas wykonywania rozkazów PUSH i POP

jego wartość zostaje automatycznie zaktualizowana, tak żeby w dalszym ciągu wskazywał na szczyt stosu. Jest to jedyny z rejestrów procesora, który jest bankowany. Poprzez bankowanie należy rozumieć, możliwość przełączania rejestru pomiędzy dostępnymi. Widoczny rejestr SP w programie może odwzorowywać fizycznie jeden z dwóch rejestrów zawierających wskaźnik szczytu stosu – MSP (ang. *Main Stack* Pointer) lub PSP (ang. *Process Stack Pointer*). To, na który w danej chwili wskazuje rejestr SP zależy od bitu nr 1 w rejestrze konfiguracyjnym CONTROL. Istnieje również możliwość bezpośredniego dostępu do rejestrów MSP i PSP poprzez wykorzystanie instrukcji MRS.



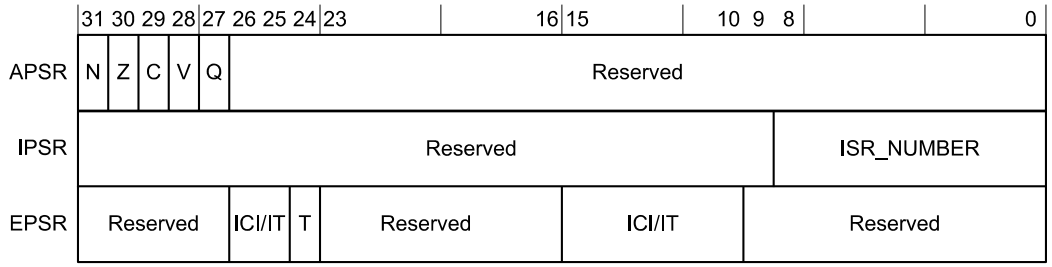
Rys. 5.3. Rejestry rdzenia Cortex-M3, opracowano wg [9]

W zawiązku z tym, że tak naprawdę istnieją dwa niezależne rejestry wskaźnika stosu, gdzie w danym momencie używany jest tylko jeden to oznacza, że procesor może wykorzystywać dwa niezależne stosy i tak też jest w rzeczywistości. Po otrzymaniu sygnału reset, rejestr SP jest przełączony na MSP. Również w trybie obsługi przerwania rejestrem tym jest zawsze MSP. Producent zaleca korzystanie z rejestru PSP w przypadku programów uruchamianych pod kontrolą systemu operacyjnego. Wówczas kod programu użytkownika powinien wykorzystywać swój własny stos, w celu uniemożliwienia uszkodzenia danych na stosie głównym. Zastosowanie przełączanego stosu wraz z ograniczonymi prawami dostępu powoduje bezpieczną pracę systemu operacyjnego przy nawet wadliwie napisanym programie użytkownika. Umiejscowienie stosu w pamięci zależy od użytkownika. Adres początku stosu należy zapisać pod adresem 0x0000 0000. Pierwszą operacją, zanim jeszcze procesor odczyta wektor reset spod adresu 0x0000 0004 jest właśnie odczytanie adresu szczytu stosu i wpisanie go do rejestru SP czyli MSP.

Kolejnym rejestrem procesora jest rejestr powrotu LR (ang. *Link Register*) R14. Procesor trzyma w nim adres miejsca w programie, do którego należy powrócić po wykonaniu podprogramu, to znaczy po wykonaniu jednej z instrukcji BL (ang. *Branch and Link*) lub BLX (ang. *Branch and Link witch Exchange*). Używany jest również w przypadku wyjścia z obsługi przerwania. W pozostałych przypadkach, można z niego korzystać jak ze zwykłego rejestru ogólnego przeznaczenia R14.

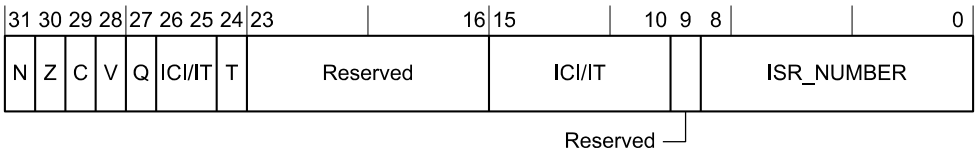
Rejestr wskaźnika instrukcji PC (ang. *Program Counter*) przechowuje adres aktualnie wykonywanej instrukcji. W przypadku programu bez rozgałęzień, przerwań, instrukcji skoków adres ten jest automatycznie inkrementowany co 2 lub co 4 bajty w zależności od rodzaju instrukcji (16- czy 32-bitowa). W przypadku wystąpienia zdarzeń, które zaburzają ciągłość programu wartość rejestru PC skokowo zmienia wartość na odpowiednią po instrukcji skoku czy przerwaniu.

Jednym z ważniejszych rejestrów procesora jest rejestr PSR (ang. *Program Status Register*). W rejestrze tym umieszczone są flagi wykonania zaktualizowane po wykonaniu poprzedniej instrukcji programu, które następnie mogą być wykorzystane w instrukcjach skoków warunkowych, zawiera także informacje na temat numeru wykonywanego przerwania, a także informacje na temat aktualnego bloku instrukcji IT. Rejestr PSR może być odczytywany w całości, ale także dostępna jest możliwość odczytywania jednego z trzech rejestrów, z których PSR się składa, a są to: APSR (ang. *Application PSR*), IPSR (ang. *Interrupt PSR*) oraz EPSR (ang. *Execution PSR*). Omawiane rejestry zostały przedstawione na rys. 5.4.



Rys. 5.4. Rejestry APSR, IPSR oraz EPSR rdzenia Cortex-M3, opracowano wg [9]

Rejestr całościowy PSR jest to suma logiczna wszystkich trzech powyższych rejestrów, w związku czym jego postać wygląda tak jak na rys. 5.5.



Rys. 5.5. Rejestr PSR rdzenia Cortex-M3, opracowano wg [9]

W rejestrze APSR poszczególne pola mają następujące znaczenia:

- flaga N – flaga znaku, ustawiana jest przez procesor wówczas, gdy wynik operacji jest ujemny, bądź w operacjach logicznych wynikiem jest „mniejsze niż”,
- flaga Z – flaga zera, ustawiana jest przez procesor wówczas, gdy wynik operacji jest równy zero,
- flaga C – flaga przeniesienia lub pożyczki, ustawiania jest wówczas, gdy nastąpiło przepełnienie przy dodawaniu, lub pożyczka przy odejmowaniu dla operacji na liczbach bez znaku
- flaga V – flaga przepełnienia, ustawiana wówczas, gdy wynik nie mieści się na 32-bitach. Podobna do flagi C, ale odnosi się do operacji na liczbach ze znakiem,
- flaga Q – flaga nasycenia, ustawiana przez procesor wówczas, gdy wynik rozkaz SSAT lub USAT jest nasycony do podanej wartości. Ten bit należy wyzerować programowo wykorzystując instrukcję MRS.

W rejestrze IPSR, znajduje się jedno pole o następującym znaczeniu:

- ISR_NUMBER – 8-bitowe pole przechowujące numer wyjątku. Równa się zero w przypadku, gdy procesor pracuje w trybie wykonywania programu, w przypadku obsługi przerwania, przyjmuje kolejne wartości dla kolejnych wyjątków z tablicy

przerwań. Dla przerwania IRQ0 przyjmuje wartość 16, dla IRQ1 wartość 17 i analogicznie dalej.

Rejestr EPSR zawiera pola związane z dwoma zagadnieniami:

- ICI – pole związane z przerywaniem i wznawianiem pracy zapisywania/odczytywania bloków danych instrukcjami LDR i STR przy wystąpieniu przerwania. W przypadku, gdy podczas wykonywania instrukcji przesyłania bloku danych nastąpi przerwanie, to proces przesyłania danych zostaje natychmiast wstrzymany, a w polach ICI zapisywane są informacje, pozwalające na wznowienie transferu po zakończeniu obsługi przerwania od danego miejsca,
- IT – pole związane z funkcjonalnością rozkazu IT, czyli warunkowego wykonania instrukcji. Rozkaz ITxyz <flaga> zostaje odpowiednio przetłumaczony na wartości poszczególnych bitów w polu IT rejestru EPSR tak, że w trakcie wykonywania kolejnych instrukcji z danego bloku IT procesor wie z jakim warunkiem (T czy E) oraz dla jakiej flagi jest wykonywany.

Z tego względu, że pola ICI oraz IT nakładają się na siebie, w przypadku wykorzystywania funkcji LDR/STR wewnątrz bloku instrukcji IT oraz wystąpienia przerwania w trakcie wykonywania LDR/STR nie zostaną zachowane wartości w polach ICI pozwalające na wznowienie przesyłu danych od miejsca wstrzymania i po zakończeniu obsługi przerwania całe przesyłanie danych LDR/STR zostanie rozpoczęte na nowo.

Kolejny z rejestrów specjalnych i zarazem pierwszy z grupy trzech rejestrów maskujących, to rejestr PRIMASK. W rejestrze tym zawarta jest tylko jedna flaga na pozycji bitu 0, o nazwie PRIMASK. Ustawienie tej flagi blokuje wszystkie przerwania, które mają konfigurowalne priorytety (między innymi wszystkie przerwania od urządzeń peryferyjnych). Może być wykorzystywany w przypadku realizacji bardzo ważnego fragmentu kodu, które nie może zostać przerwany z mniej ważnego powodu. Wyjątki od błędów nie są blokowane.

Rejestr FAULTMASK również zawiera jedną flagę konfiguracyjną na pozycji 0 o nazwie FAULTMASK. Jej ustawienie blokuje wszystkie przerwania, które mają konfigurowalne priorytety, a także wyjątki od błędów. Nie blokowane są tylko przerwania niemaskowalne oznaczone jako NMI (ang. *Non Maskable Interrupt*) oraz reset. Może być wykorzystywane w przypadku ultraważnych fragmentów programu, czy obsługi przerwań, które nie powinny być wywłaszczone przez żadne inne przerwania niezależnie od stanu mikrokontrolera.

Ostatni z rejestrów maskujących przerwania i zarazem przedostatni z rejestrów specjalnych to rejestr BASEPRI. Zawiera on 8-bitowe pole znajdujące się na bitach od 0 do 7, w którym można podać priorytet przerwań, poniżej którego przerwania mają być blokowane. Domyślnie, po resecie znajduje się w nim wartość 0x00, która nie maskuje żadnych przerwań.

Rejestrem kończącym grupę rejestrów specjalnych jest rejestr CONTROL. Zawiera w swojej strukturze dwie flagi: TPL na pozycji 0 oraz ASPSEL na pozycji 1. Flaga TPL odpowiada za uprawnienia wykonywanego programu – ustawienie tej flagi powoduje wykonywanie kodu w trybie użytkownika, natomiast wyzerowanie w trybie uprzywilejowanym. Domyślnie, flaga ta jest wyzerowana. Flaga ma znaczenie tylko w przypadku pracy w trybie wykonywania programu, w trybie obsługi przerwania kod zawsze wykonywany jest w trybie uprzywilejowanym. Flaga ASPSEL odpowiada, za przełączanie banku rejestru wskaźnika stosu SP. Domyślnie jest wyzerowana, co odpowiada wybraniu rejestru MSP, ustawienie jej powoduje wybranie rejestru PSP. Flaga ma znaczenie tylko w przypadku trybu wykonywania programu, gdyż w trybie obsługi przerwania SP jest zawsze rejestrem MSP.

5.6. Wybrane wewnętrzne układy peryferyjne

W mikrokontrolerze oprócz samego rdzenia Cortex-M3 zaimplementowanego na podstawie licencji udzielonej przez ARM Holdings istnieje również szereg urządzeń peryferyjnych, które zostały zabudowane przez samego już producenta STMicroelectronics. W przestrzeni adresowej urządzenia te widoczne są począwszy od adresu 0x4000 0000 i rozciągają się przez kolejne 0,5 GB. Ilość oraz złożoność konfiguracji poszczególnych urządzeń peryferyjnych jest na tyle duża, że w ramach niniejszej pracy nie sposób jest opisać wszystkich dostępnych. Z tego też względu zdecydowano się na opisanie wybranych trzech podstawowych układów spośród 27 dostępnych. Dalsze informacje znajdują się w instrukcji producenta [12].

5.6.1. Układy kontroli sygnałów zegarowych RCC

Zasada działania oraz schemat blokowy układu kontroli sygnałów zegarowych (rys. 4.33) zostały przedstawione w rozdziale *4.6. Dystrybucja sygnałów zegarowych*.

Offset	Register	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0x000	**RCC_CR**	Reserved						PLL RDY	PLL ON	Reserved				CSSON	HSEBYP	HSERDY	HSEON	HSICAL[7:0]								HSITRIM[4:0]					Reserved	HSIRDY	HSION
	Reset value							0	0					0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0		1	1
0x004	**RCC_CFGR**	Reserved					MCO [2:0]			Reserved	PLLMUL[3:0]				PLLXTPRE	PLLSRC	ADC PRE [1:0]		PPRE2 [2:0]			PPRE1 [2:0]			HPRE[3:0]				SWS [1:0]		SW [1:0]	
	Reset value						0	0	0		0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0x008	**RCC_CIR**	Reserved								CSSC	Reserved	PLLRDYC	HSERDYC	HSIRDYC	LSERDYC	LSIRDYC	Reserved			PLLRDYIE	HSERDYIE	HSIRDYIE	LSERDYIE	LSIRDYIE	CSSF	Reserved		PLLRDYF	HSERDYF	HSIRDYF	LSERDYF	LSIRDYF	
	Reset value									0		0	0	0	0	0				0	0	0	0	0	0			0	0	0	0	0	
0x00C	**RCC_APB2RSTR**	Reserved													TIM17RST	TIM16RST	TIM15RST	Reserved	USART1RST	Reserved	SPI1RST	TIM1RST	Reserved	ADC1	IOPGRST	IOPFRST	IOPERST	IOPDRST	IOPCRST	IOPBRST	IOPARST	Reserved	AFIORST	
	Reset value														0	0	0		0		0	0		0	0	0	0	0	0	0	0		0	
0x010	**RCC_APB1RSTR**	Reserved	CECRST	DACRST	PWRRST	BKPRST	Reserved			I2C2RST	I2C1RST	UART5RST	UART4RST	USART3RST	USART2RST	Reserved	SPI3RST	SPI2RST	Reserved		WWDGRST	Reserved		TIM14RST	TIM13RST	TIM12RST	TIM7RST	TIM6RST	TIM5RST	TIM4RST	TIM3RST	TIM2RST	
	Reset value			0	0	0				0	0	0	0	0	0		0	0			0			0	0	0	0	0	0	0	0	0	
0x014	**RCC_AHBENR**	Reserved																						FSMCEN	Reserved	CRCEN	Reserved	FLITFEN	Reserved	SRAMEN	DMA2EN	DMA1EN	
	Reset value																							0		0		1		1	0	0	
0x018	**RCC_APB2ENR**	Reserved													TIM17EN	TIM16EN	TIM15EN	Reserved	USART1EN	Reserved	SPI1EN	TIM1EN	Reserved	ADC1EN	IOPGEN	IOPFEN	IOPEEN	IOPDEN	IOPCEN	IOPBEN	IOPAEN	Reserved	AFIOEN	
	Reset value														0	0	0		0		0	0		0	0	0	0	0	0	0	0		0	
0x01C	**RCC_APB1ENR**	Reserved	CECEN	DACEN	PWREN	BKPEN	Reserved			I2C2EN	I2C1EN	UART5EN	UART4EN	USART3EN	USART2EN	Reserved	SPI3EN	SPI2EN	Reserved		WWDGEN	Reserved		TIM14EN	TIM13EN	TIM12EN	TIM7EN	TIM6EN	TIM5EN	TIM4EN	TIM3EN	TIM2EN	
	Reset value		0	0	0	0				0	0	0	0	0	0		0	0			0			0	0	0	0	0	0	0	0	0	
0x020	**RCC_BDCR**	Reserved															BDRST	RTCEN	Reserved				RTC SEL [1:0]		Reserved					LSEBYP	LSERDY	LSEON	
	Reset value																0	0					0	0						0	0	0	
0x024	**RCC_CSR**	LPWRSTF	WWDGRSTF	IWDGRSTF	SFTRSTF	PORRSTF	PINRSTF	Reserved	RMVF	Reserved																						LSIRDY	LSION
	Reset value	0	0	0	0	1	1		0																							0	0
0x02C	**RCC_CFGR2**	Reserved																												PREDIV1[3:0]			
	Reset value																													0	0	0	0

Rys. 5.6. Rejestry konfiguracyjne kontrolera sygnałów zegarowych, opracowano wg [12]

Poniżej zostaną omówione jedynie rejestry konfiguracyjne kontrolera, które zostały zgrupowane na rys. 5.6. Zgodnie z rys. 5.1 pierwszy rejestr konfiguracyjny kontrolera RCC jest widoczny pod adresem 0x4002 1000. Pierwsza od lewej kolumna w tabeli przedstawionej na rys. 5.6 określa offset czyli przesunięcie względem bazowego adresu jakim jest właśnie 0x4002 1000. Oznacza to, że rejestr o nazwie RCC_CR widoczny jest pod adresem 0x4002 1000, a kolejny rejestr, to znaczy RCC_CFGR widoczny jest pod adresem 0x4002 1004. Nazwy rejestrów umieszczone są w drugiej od lewej strony kolumnie. W przypadku pisania programu w języku C i korzystania z biblioteki dostarczanej przez producenta, do rejestrów można się odwoływać w sposób przedstawiony na listingu 5.30.

Listing 5.30. Przykład odwołania się do rejestru RCC_APB1ENR w języku C z wykorzystaniem biblioteki

```
RCC->APB1ENR = 0x10000000;
```

Taki zapis oznacza wpisanie wartości 0x1000 0000 do rejestru RCC_APB1ENR. W bibliotece bowiem, zdefiniowane są struktury z nazwami rejestrów i odpowiednimi ich adresami w pamięci mikrokontrolera dla poszczególnych urządzeń. Na listingu 5.30. RCC jest wskaźnikiem na strukturę dla kontrolera sygnałów zegarowych, a odwołanie się do konkretniej składowej tej struktury (w tym wypadku rejestru APB1ENR) dokonuje się w języku C poprzez operator strzałki „->”. Podane informacje obowiązują dla wszystkich rejestrów urządzeń peryferyjnych.

Konfiguracja układu RCC odbywa się poprzez wpisanie odpowiednich wartości do jedenastu 32-bitowych rejestrów konfiguracyjnych. Nie zawsze jednak wszystkie 32 bity w poszczególnych rejestrach są wykorzystywane. Nieużywane bity czy całe obszary oznaczone są na rys. 5.6. jako „Reserved”. Po wykonaniu resetu mikrokontrolera w rejestrach konfiguracyjnych znajdują się już pewne wartości oznaczone jako „Reset value”, dlatego w większości przypadków w programie nie ma potrzeby zapisywania wartości do wszystkich 11 rejestrów, ale tylko do wybranych według potrzeb wynikających z aplikacji.

Znaczenie ważniejszych flag (bitów) oraz obszarów w rejestrze RCC_CR:

- PLLRDY – flaga tylko do odczytu ustawiana sprzętowo, w celu sygnalizacji ustabilizowania pracy pętli PLL,
- PLLON – flaga ustawiana i kasowana programowo w celu załączenia (wartość 1) lub wyłączenia (wartość 0) pętli PLL,
- CSSON – flaga ustawiana i kasowana programowo w celu załączenia (wartość 1) lub wyłączenia (wartość 0) pętli układu nadzoru sygnału zegarowego HSE,
- HSEBYP – flaga ustawiana i kasowana programowo w celu załączenia (wartość 1) lub wyłączenia (wartość 0) bypassu generatora HSE. Przy załączonym bypassie, nie podłącza się oscylatora do wejść OSC_IN oraz OSC_OUT, ale bezpośrednio sygnał zegarowy do wejścia OSC_IN, a OSC_OUT pozostawia się niepodłączony,
- HSERDY – flaga tylko do odczytu ustawiana sprzętowo, w celu sygnalizacji ustabilizowania pracy generatora HSE,
- HSEON – flaga ustawiana i kasowana programowo w celu załączenia (wartość 1) lub wyłączenia (wartość 0) generatora HSE,
- HSIRDY – flaga tylko do odczytu ustawiana sprzętowo, w celu sygnalizacji ustabilizowania pracy generatora HSI,
- HSION – flaga ustawiana i kasowana programowo w celu załączenia (wartość 1) lub wyłączenia (wartość 0) generatora HSI.

Należy w tym miejscu zaznaczyć, że po resecie mikrokontrolera, załączony i wybrany jako źródło sygnału zegarowego jest zawsze generator HSI. Istotną informacją jest również brak reakcji mikrokontrolera na wyłączenie danego generatora w przypadku, gdy jest on

wykorzystywany bezpośrednio lub pośrednio jako źródło sygnału zegarowego. Wówczas operacja wyłączenia danego generatora nie jest przeprowadzana.

Znaczenie ważniejszych flag (bitów) oraz obszarów w rejestrze RCC_CFGR:

- MCO – obszar odczytywany i zapisywany programowo określający źródło sygnału na wyprowadzenie sygnału zegarowego MCO (brak lub SYSCLK, HSI, HSE, PLL),
- PLLMUL – obszar odczytywany i zapisywany programowo określający mnożnik częstotliwości w powielaczu PLL (od x2 do x16), zmiana wartości odnosi skutek tylko przy wyłączonej pętli PLL,
- ADCPRE – obszar odczytywany i zapisywany programowo określający preskaler dla przetwornika analogowo-cyfrowego (/2, /4, /6 lub /8),
- PPRE2 – obszar odczytywany i zapisywany programowo określający preskaler dla magistrali APB2 (brak lub /2, /4, /8, /16),
- PPRE1 – obszar odczytywany i zapisywany programowo określający preskaler dla magistrali APB1 (brak lub /2, /4, /8, /16),
- HPRE – obszar odczytywany i zapisywany programowo określający preskaler dla zegara SYSCLK (brak lub /2, /4, /8, /16, /64, /128, /256, /512),
- SWS – obszar tylko do odczytu ustawiany sprzętowo informujący o aktualnym źródle sygnału SYSCLK (HSI, HSE lub PLL),
- SW – obszar odczytywany i zapisywany programowo określający źródło sygnału SYSCLK (HSI, HSE lub PLL), zmiana wartości odnosi skutek tylko w przypadku stabilnej pracy źródła na które zostaje przełączony SYSCLK.

Kolejny rejestr RCC_CIR służy do uaktywnienia przerwań od zdarzeń w układzie RCC (na przykład przerwanie od uzyskania gotowości pętli PLL). Rejestry RCC_APB2RSTR oraz RCC_APB1RSTR służą do resetowania urządzeń peryferyjnych poprzez ustawienie odpowiednich flag. Kolejne trzy rejestry umożliwiają załączenie lub wyłączenie sygnału zegarowego dla poszczególnych urządzeń przyłączonych do magistral AHB lub APB. Rejestr RCC_AHBENR konfiguruje załączanie sygnału zegarowego dla urządzeń przyłączonych do magistral AHB, między innymi pamięci SRAM czy układu DMA. Rejestr RCC_APB2ENR konfiguruje załączanie sygnału zegarowego dla urządzeń przyłączonych do magistrali APB2, natomiast RCC_APB1ENR dla urządzeń przyłączonych do APB1. Nazwy poszczególnych flag w tych rejestrach widoczne na rys. 5.6. odpowiadają nazwom urządzeń peryferyjnych. Warto zauważyć, że domyślnie wyłączone są sygnały zegarowe dla wszystkich urządzeń przyłączonych do magistral APB. Chcąc więc korzystać na przykład z linii wejść/wyjść portu C mikrokontrolera należy ustawić flagę IOPCEN w rejestrze RCC_APB2ENR. Podczas konfiguracji danego urządzenia należy zawsze w pierwszej kolejności włączyć sygnał zegarowy dla danego urządzenia, natomiast później rozpocząć konfigurację jego rejestrów.

Rejestr RCC_BDCR odpowiedzialny jest za konfigurację kontrolera kopii zapasowej oraz zegara czasu rzeczywistego RTC. Znaczenie ważniejszych flag (bitów) oraz obszarów w rejestrze RCC_BDCR:

- RTCE – flaga ustawiana i kasowana programowo w celu załączenia (wartość 1) lub wyłączenia (wartość 0) zegara RTC,
- RTCSEL – obszar odczytywany i zapisywany programowo określający źródło sygnału zegarowego dla zegara RTC (LSE, LSI lub HSE),
- LSEBYP – flaga ustawiana i kasowana programowo w celu załączenia (wartość 1) lub wyłączenia (wartość 0) bypassu generatora LSE. Przy załączonym bypassie, nie podłącza się oscylatora do wejść OSC32_IN oraz OSC32_OUT, ale bezpośrednio sygnał zegarowy do wejścia OSC32_IN, a OSC32_OUT pozostawia się niepodłączony,

- LSERDY – flaga tylko do odczytu ustawiana sprzętowo, w celu sygnalizacji ustabilizowania pracy generatora LSE,
- LSEON – flaga ustawiana i kasowana programowo w celu załączenia (wartość 1) lub wyłączenia (wartość 0) generatora LSE.

Kolejny z rejestrów układu RCC o nazwie RCC_CSR przechowuje flagi informujące o przyczynie wystąpienia resetu. Oprócz powyższego w rejestrze tym znajdują się następujące flagi:

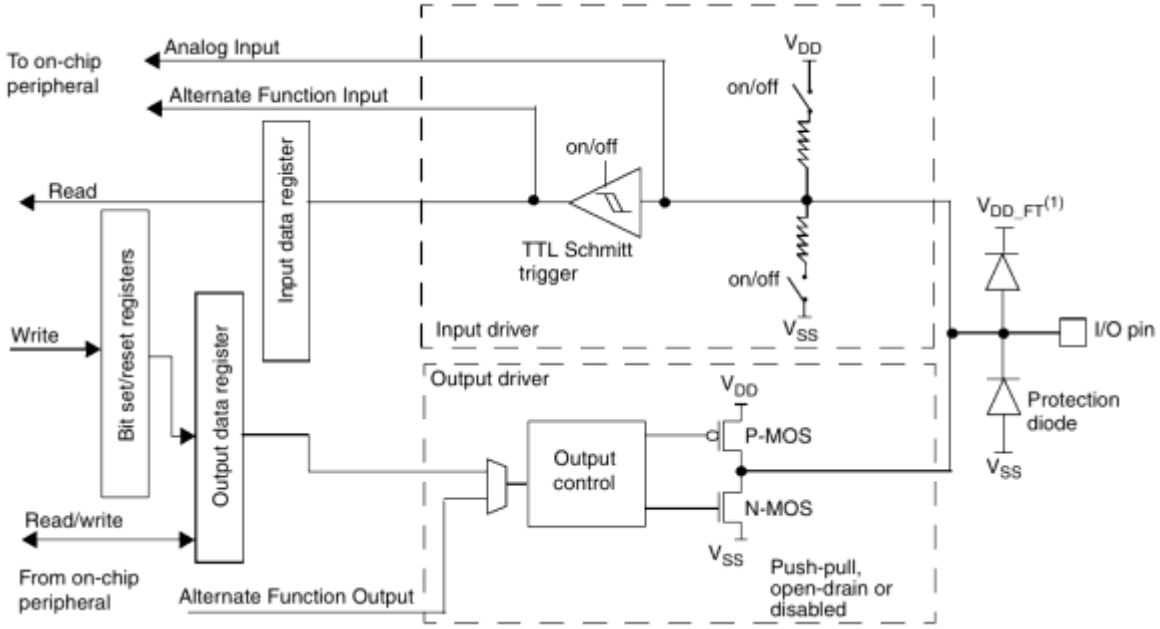
- LSIRDY – flaga tylko do odczytu ustawiana sprzętowo, w celu sygnalizacji ustabilizowania pracy generatora LSI,
- LSION – flaga ustawiana i kasowana programowo w celu załączenia (wartość 1) lub wyłączenia (wartość 0) generatora LSI.

Ostatni rejestr RCC_CFGR2 zawiera obszar, w którym konfiguruje się ustawienia preskalera PREDIV1 w zakresie od /1 do /16.

Ustawienia układu RCC powinny być jedną z pierwszych czynności wykonywanych w programie. Dopiero po ustabilizowaniu się pracy generatorów, wyborze odpowiednich źródeł i zasileniu sygnałem zegarowych wybranych układów peryferyjnych można przystąpić do dalszej konfiguracji mikrokontrolera czy wykonywania programu głównego. Nie mniej jednak w trakcie wykonywania programu również można zmieniać parametry układu RCC na przykład zmniejszając częstotliwość pracy w celu redukcji poboru energii elektrycznej w czasie, gdy procesor wykonuje mniej wymagające fragmenty kodu.

5.6.2. Porty wejścia/wyjścia

Sam mikrokontroler bez możliwości komunikacji z urządzeniami zewnętrznymi jest praktycznie bezużyteczny. Dopiero przyłączenie mikrokontrolera do urządzeń takich jak na przykład klawiatura, wyświetlacze, porty komunikacyjne, czujniki, kontrolki czy elementy wykonawcze powoduje powstanie pełnowartościowego systemu, pełniącego określone funkcje. Schemat blokowy budowy wewnętrznej jednej linii portu ogólnego przeznaczenia przedstawiono na rys. 5.7.



Rys. 5.7. Schemat blokowy jednego wyprowadzenia (jednej linii mikrokontrolera) ogólnego przeznaczenia, opracowano wg [12]

Linia wejścia/wyjścia składa się z dwóch zasadniczych bloków – sterownika wejścia i sterownika wyjścia. W zależności czy w danym momencie linia wykorzystywana jest jako wejście czy jako wyjście uaktywniony jest jeden ze sterowników. Sterownik wyjściowy zawiera w swej strukturze komplementarną parę tranzystorów P-MOS oraz N-MOS sterowaną przez układ kontrolny. W zależności od konfiguracji, wyjście może być typu otwarty dren (ang. *open-drain*) co oznacza, że wyjście pozostaje w stanie wysokiej impedancji w przypadku wystawiania logicznej jedynki i zwierania wyjścia do potencjału V_{SS} w przypadku wystawiania logicznego zera lub typu push-pull, w który w przypadku wystawiania jedynki wyjście zwierane jest do napięcia V_{DD}, a w przypadku zera analogicznie jak w wersji z otwartym drenem. Sterownik wyjściowy zawierający tranzystory może być sterowany bezpośrednio przez ustawienie bitów w rejestrze sterującym portu, bądź w trybie funkcji alternatywnej poprzez urządzenie peryferyjne, które będzie wykorzystywało dane wyprowadzenie. Sterownik wejściowy wyprowadzenia zawiera w swych strukturach rezystory podciągające do napięć V_{SS} oraz V_{DD}, których załączenie ustawia się w rejestrach konfiguracyjnych oraz przerzutnik Shmitta w celu kondycjonowania sygnału wejściowego. Wejście bez załączonego żadnego rezystora nazwane jest wejściem pływającym. Sygnał wejściowy może być przesłany w zależności od konfiguracji do rejestrów odczytu danego portu, układu peryferyjnego przy wyborze funkcji alternatywnej albo do przetwornika analogowo-cyfrowego. Samo wyprowadzenie zabezpieczone jest również dwoma diodami prostowniczymi, które ograniczają zakres pojawiających się napięć na wyprowadzeniu do zakresu od V_{SS} – 0,5 V do V_{DD_FT} + 0,5 V, gdzie napięcie V_{DD_FT} wynosi +5 V względem V_{SS} (stąd tolerancja wyprowadzeń na napięcia z zakresu TTL).
Rejestry konfiguracyjne portów zostały przedstawione na rys. 5.8.

Offset	Register	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0x00	**GPIOx_CRL**	CNF7 [1:0]		MODE7 [1:0]		CNF6 [1:0]		MODE6 [1:0]		CNF5 [1:0]		MODE5 [1:0]		CNF4 [1:0]		MODE4 [1:0]		CNF3 [1:0]		MODE3 [1:0]		CNF2 [1:0]		MODE2 [1:0]		CNF1 [1:0]		MODE1 [1:0]		CNF0 [1:0]		MODE0 [1:0]	
	Reset value	0	1	0	0	0	1	0	0	0	1	0	0	0	1	0	0	0	1	0	0	0	1	0	0	0	1	0	0	0	1	0	0
0x04	**GPIOx_CRH**	CNF 15 [1:0]		MODE1 5 [1:0]		CNF 14 [1:0]		MODE1 4 [1:0]		CNF 13 [1:0]		MODE1 3 [1:0]		CNF 12 [1:0]		MODE1 2 [1:0]		CNF 11 [1:0]		MODE1 1 [1:0]		CNF 10 [1:0]		MODE1 0 [1:0]		CNF 9 [1:0]		MODE9 [1:0]		CNF 8 [1:0]		MODE8 [1:0]	
	Reset value	0	1	0	0	0	1	0	0	0	1	0	0	0	1	0	0	0	1	0	0	0	1	0	0	0	1	0	0	0	1	0	0
0x08	**GPIOx_IDR**	Reserved																IDR[15:0]I															
	Reset value																	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0x0C	**GPIOx_ODR**	Reserved																ODR[15:0]															
	Reset value																	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0x10	**GPIOx_BSRR**	BR[15:0]																BSR[15:0]															
	Reset value	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0x14	**GPIOx_BRR**	Reserved																BR[15:0]															
	Reset value																	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0x18	**GPIOx_LCKR**	Reserved															LCKK	LCK[15:0]															
	Reset value																0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

Rys. 5.8. Rejestry konfiguracyjne portów wejścia/wyjścia mikrokontrolera, opracowano wg [12]

Na rysunku tym w nazwach rejestrów występuje oznaczenie „x”, która należy zastąpić literą A, B, C lub D określającą port mikrokontrolera. Rejestry konfiguracyjne poszczególnych portów rozpoczynają się też od różnych adresów, stąd przesunięcie podane w pierwszej od lewej kolumnie, jest prawdziwe dla każdego z portów z osobna. Adresy początkowe rejestrów konfiguracyjnych poszczególnych portów przedstawiają się następująco:

- port A od adresu 0x4001 0800,
- port B od adresu 0x4001 0C00,

- port C od adresu 0x4001 1000,
- port D od adresu 0x4001 1400.

Rejestr GPIOx_CRL zawiera konfiguracje trybu pracy linii o numerach od 0 do 7 danego portu, a GPIOx_CRH analogicznie dla linii o numerach od 8 do 15. Konfiguracja każdej z linii zawarta jest w dwóch dwubitowych polach MODE[1:0] oraz CNF[1:0]. Tryby pracy wyprowadzenia w zależności od konfiguracji zostały przedstawione w tab. 5.10.

Tablica 5.10. Tryby pracy wyprowadzenia w zależności od konfiguracji

<table>
<tr><th colspan="2">Tryb pracy wyprowadzenia</th><th>CNF1</th><th>CNF0</th><th colspan="2">MODE1</th><th>xODR</th></tr>
<tr><td rowspan="2">Wyjście ogólnego przeznaczenia</td><td>Push-pull</td><td rowspan="2">0</td><td>0</td><td colspan="2" rowspan="4">01 – maks. częstotliwość 10MHz
10 – maks. częstotliwość 2MHz
11 – maks. częstotliwość 50MHz</td><td>0 lub 1</td></tr>
<tr><td>Otwarty dren</td><td>1</td><td>0 lub 1</td></tr>
<tr><td rowspan="2">Wyjście funkcji alternatywnej</td><td>Push-pull</td><td rowspan="2">1</td><td>0</td><td>-</td></tr>
<tr><td>Otwarty dren</td><td>1</td><td>-</td></tr>
<tr><td rowspan="4">Wejście</td><td>Analogowe</td><td rowspan="2">0</td><td>0</td><td colspan="2" rowspan="3">00</td><td>-</td></tr>
<tr><td>Pływające</td><td>1</td><td>-</td></tr>
<tr><td>Z rezystorem podciągającym do V_{SS}</td><td rowspan="2">1</td><td rowspan="2">0</td><td>0</td></tr>
<tr><td>Z rezystorem podciągającym do V_{CC}</td><td>1</td></tr>
</table>

W tabeli oprócz wartości pól MODE oraz CNF znajduje się także kolumna z rejestrem GPIOx_ODR. Rejestr ten w przypadku skonfigurowania linii jako wyjścia służy do ustawiania stanu danej linii. Pierwsze szesnaście bitów tego rejestru odpowiada szesnastu liniom danego portu. W przypadku wpisania do wszystkich bitów wartości „0” i ustawieniu wszystkich linii w tryb wyjścia na każdej z linii będzie panował potencjał V_{SS} wymuszony zwartym tranzystorem N-MOS. Natomiast w przypadku ustawienia linii w tryb wejścia i ustawieniu flagi CNF1 dla danej linii, to wartością znajdującą się w bicie rejestru GPIOx_ODR przyporządkowanym danej linii ustawia się załączenie jednego z dwóch rezystorów podciągających.

Rejestr GPIOx_IDR odwzorowuje stan linii danego portu. Innymi słowy, jeżeli linia jest skonfigurowana jako wejściowa, to panujący na niej stan odczytujemy z rejestru GPIOx_IDR, natomiast jeżeli jest skonfigurowana jako wyjściowa, to stan na niej wymuszamy poprzez zapis odpowiedniej wartości do rejestru GPIOx_ODR, pamiętając aby znak „x” zastąpić odpowiednią literą portu. Producent umożliwia jeszcze jeden mechanizm zapisu wartości do rejestru GPIOx_ODR. Rozważając przypadek gdy na przykład należy zmienić stan tylko jednej linii, a pozostałe pozostawić niezmienione, należałoby najpierw odczytać wartość rejestru GPIOx_ODR, zmienić w odczytanej wartości jeden bit i zapisać z powrotem do tego rejestru. Operację można jednak przeprowadzić w prostszy sposób, wykorzystując kolejny rejestr o nazwie GPIOx_BSRR. Rejestr ten podzielony jest na dwie 16-bitowe części o nazwach BR i BSR. Każdy z 16-bitów w każdej z części odpowiada kolejno liniom danego portu. Zapisanie wartości „0” do któregokolwiek z tych rejestrów nie przynosi żadnych skutków, natomiast zapisane wartości „1” w danym bicie powoduje:

- w części BR – wyzerowanie odpowiadającego bitu w rejestrze GPIOx_ODR i tym samym ustawienie stanu niskiego na odpowiadającej linii,
- w części BSR – ustawienie odpowiadającego bitu w rejestrze GPIOx_ODR i tym samym ustawienie stanu wysokiego w przypadku skonfigurowania jako push-pull lub stanu wysokiej impedancji w przypadku ustawienia jako otwarty dren na odpowiadającej linii.

Część BR jest też dostępna w osobnym rejestrze GPIOx_BRR. Zasada działania jest analogiczna jak w przypadku części BR w rejestrze GPIOx_BSRR.

Ostatni z rejestrów konfiguracyjnych portów ogólnego przeznaczenia – GPIOx_LCKR umożliwia zablokowanie możliwości zmiany konfiguracji danych linii. Odblokowanie konfiguracji następuje dopiero po wystąpieniu sygnału reset.

Kolejną częścią konfiguracji portów jest możliwość przeprzypożądkowania (ang. *remap*) funkcji alternatywnych przypisanych do poszczególnych linii. Przypożadkowanie funkcji alternatywnych do linii o raz możliwość ich przemapowania została zestawiona w tab. 4.5. Na rys. 5.9. przedstawiono natomiast rejestry konfiguracyjne służące konfiguracji funkcji alternatywnych.

Offset	Register	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0x00	**AFIO_EVCR**	Reserved																									EVOE	PORT[2:0]			PIN[3:0]		
	Reset value																										0	0	0	0	0	0	0
0x04	**AFIO_MAPR**	Reserved					SWJ_ CFG[2:0]			Reserved							TIM5CH4_IREMAP	PD01_REMAP	Reserved		TIM4_REMAP	TIM3_REMAP[1:0]		TIM2_REMAP[1:0]		TIM1_REMAP[1:0]		USART3_REMAP[1:0]		USART2_REMAP	USART1_REMAP	I2C1_REMAP	SPI1_REMAP
	Reset value						0	0	0								0	0			0	0	0	0	0	0	0	0	0	0	0	0	0
0x08	**AFIO_EXTICR1**	Reserved																EXTI3[3:0]				EXTI2[3:0]				EXTI1[3:0]				EXTI0[3:0]			
	Reset value																	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0x0C	**AFIO_EXTICR2**	Reserved																EXTI7[3:0]				EXTI6[3:0]				EXTI5[3:0]				EXTI4[3:0]			
	Reset value																	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0x10	**AFIO_EXTICR3**	Reserved																EXTI11[3:0]				EXTI10[3:0]				EXTI9[3:0]				EXTI8[3:0]			
	Reset value																	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0x14	**AFIO_EXTICR4**	Reserved																EXTI15[3:0]				EXTI14[3:0]				EXTI13[3:0]				EXTI12[3:0]			
	Reset value																	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0x1C	**AFIO_MAPR2**	Reserved																		MISC_ REMAP	TIM12_REMAP	TIM67_DAC_ DMA_ REMAP	FSMC_NADV	TIM14_REMAP	TIM13_REMAP	Reserved		TIM1_DMA_REMAP	CEC_REMAP	TIM17_REMAP	TIM16_REMAP	TIM15_REMAP	
	Reset value																			0	0	0	0	0	0			0	0	0	0	0	

Rys. 5.9. Konfiguracja funkcji alternatywnych wyprowadzeń, opracowano wg [12]

Pierwszy rejestr AFIO_EVCR jest widoczny w przestrzeni adresowej procesora pod adresem 0x4001 0000. Rejestr ten umożliwia ustawienie wybranej linii wybranego portu w tryb sygnalizowania wystąpienia zdarzenia pochodzącego z rdzenia Cortex (ang. *event output*). Kolejny rejestr AFIO_MAPR umożliwia wcześniej wspomniane przemapowania funkcji alternatywnych. Znaczenie ważniejszych flag (bitów) oraz obszarów w rejestrze AFIO_MAPR:

- SWJ_CFG – obszar odczytywany i zapisywany programowo określający tryb pracy interfejsu debugera (pełny JTAG + SW, sam SW lub brak),
- USART3_REMAP – obszar odczytywany i zapisywany programowo określający przeprzypożądkowanie linii interfejsu USART3:
    - 00 – brak przeprzypożadkowania (TX/PB10, RX/PB11, CK/PB12, CTS/PB13, RTS/PB14),
    - 01 – częściowe przeprzypożądkowanie (TX/PC10, RX/PC11, CK/PC12, CTS/PB13, RTS/PB14),
    - 11 – całkowite przeprzypożądkowanie (TX/PD8, RX/PD9, CK/PD10, CTS/PD11, RTS/PD12),

- USART2_REMAP – obszar odczytywany i zapisywany programowo określający przeprzypożądkowanie linii interfejsu USART2:
  - 0 – brak przeprzypożądkowania (CTS/PA0, RTS/PA1, TX/PA2, RX/PA3, CK/PA4),
  - 1 – przeprzypożądkowanie (CTS/PD3, RTS/PD4, TX/PD5, RX/PD6, CK/PD7),
- USART1_REMAP – obszar odczytywany i zapisywany programowo określający przeprzypożądkowanie linii interfejsu USART1:
  - 0 – brak przeprzypożądkowania (TX/PA9, RX/PA10),
  - 1 – przeprzypożądkowanie (TX/PB6, RX/PB7).

Rejestry AFIO_EXTICR1, -2, -3, -4 służą do wyboru wyprowadzeń mikrokontrolera, które będą skonfigurowane jako źródła przerwań zewnętrznych. W rejestrze AFIO_MAPR2 natomiast kontynuowane są przeprzypożądkowania elementów, które nie zostały uwzględnione w pierwszym rejestrze AFIO_MAPR.

Dzięki wielu możliwościom konfiguracji poszczególnych wyprowadzeń i całych portów mikrokontrolera, staje się on bardzo wygodny do zastosowania w różnych aplikacjach.

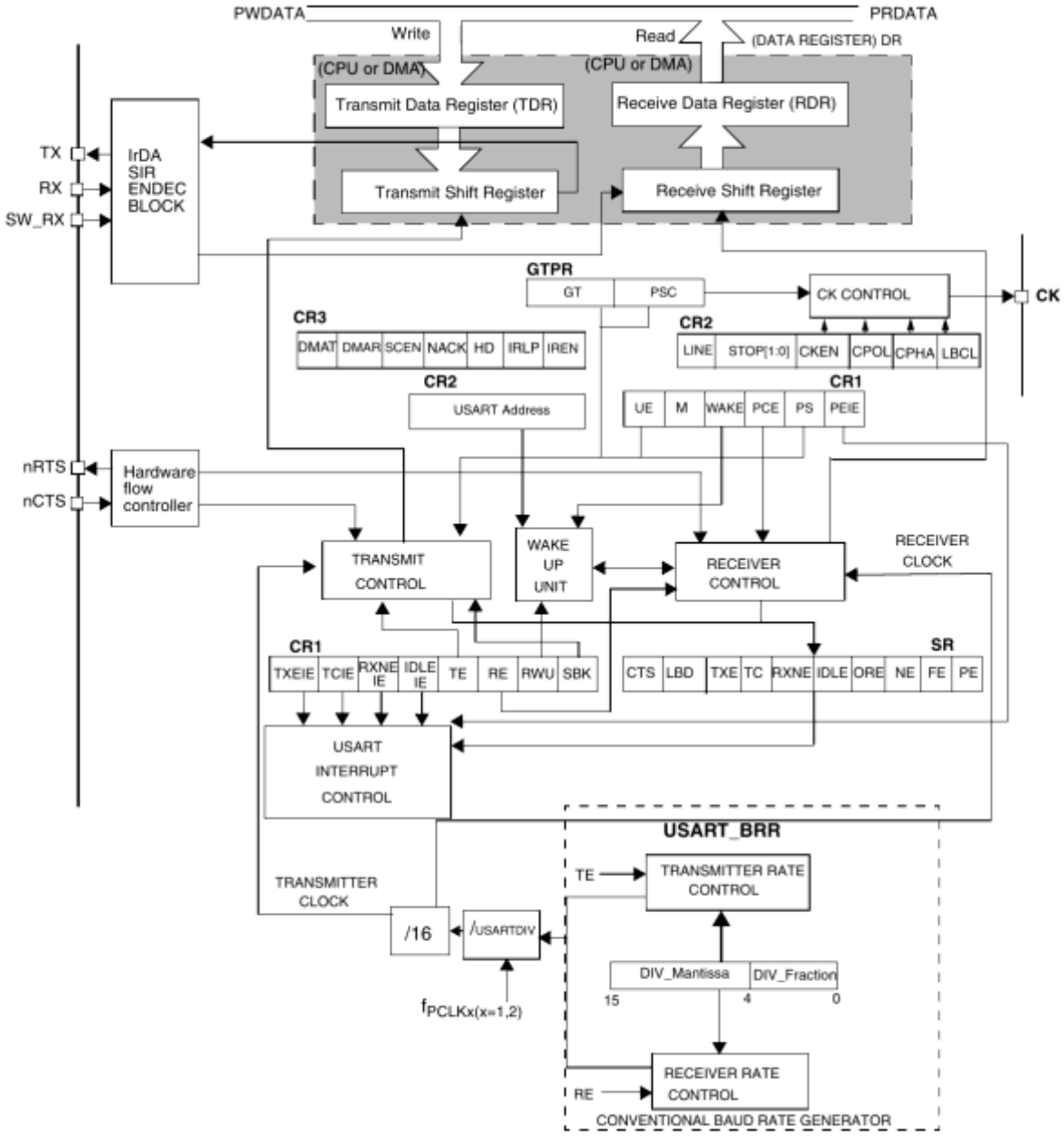
5.6.3. Interfejs szeregowy USART

Jednym z popularniejszych interfejsów komunikacyjnych wykorzystywanych w systemach komputerowych dawniej i obecnie jest uniwersalny port szeregowy mogący pracować w trybie transmisji synchronicznej i asynchronicznej (ang. *USART – Universal Synchronous and Asynchronous Receiver and Transmitter*). Najczęściej jest on wykorzystywany do komunikacji w standardach RS232 lub RS485 do dziś szeroko stosowanych w automatyce przemysłowej pomimo upowszechnienia się standardu Ethernet. Interfejs USART wspiera również inne standardy takie jak: Smartcard, IrDA, LIN. Układ jest też w stanie po wcześniejszej konfiguracji zgłaszać zapotrzebowania obsłużenia przerwań, czy korzystać z możliwości bezpośredniego dostępu do pamięci (DMA).

Offset	Register	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0x00	**USART_SR**	Reserved																						CTS	LBD	TXE	TC	RXNE	IDLE	ORE	NE	FE	PE
	Reset value																							0	0	1	1	0	0	0	0	0	0
0x04	**USART_DR**	Reserved																							DR[8:0]								
	Reset value																								0	0	0	0	0	0	0	0	0
0x08	**USART_BRR**	Reserved																DIV_Mantissa[15:4]												DIV_Fraction [3:0]			
	Reset value																	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0x0C	**USART_CR1**	Reserved																		UE	M	WAKE	PCE	PS	PEIE	TXEIE	TCIE	RXNEIE	IDLEIE	TE	RE	RWU	SBK
	Reset value																			0	0	0	0	0	0	0	0	0	0	0	0	0	0
0x10	**USART_CR2**	Reserved																	LINEN	STOP [1:0]		CLKEN	CPOL	CPHA	LBCL	Reserved	LBDIE	LBDL	Reserved	ADD[3:0]			
	Reset value																		0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0x14	**USART_CR3**	Reserved																					CTSIE	CTSE	RTSE	DMAT	DMAR	SCEN	NACK	HDSEL	IRLP	IREN	EIE
	Reset value																						0	0	0	0	0	0	0	0	0	0	0
0x18	**USART_GTPR**	Reserved																GT[7:0]							PSC[7:0]							
	Reset value																	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

Rys. 5.10. Rejestry konfiguracyjne interfejsu USART, opracowano wg [12]

Omawiany mikrokontroler wyposażony jest w trzy niezależne interfejsy USART. Schemat blokowy wewnętrzny takiego interfejsu został przedstawiony na rys. 5.11. Na schemacie występują też nazwy rejestrów zestawione na rys. 5.10.



Rys. 5.11. Schemat blokowy interfejsu USART, opracowano wg [12]

Przedstawiony na schemacie układ USART składa się z kilku podstawowych elementów. Patrząc od góry widoczny jest rejestr DR (ang. *Data Register*). Jest to jeden rejestr, jednak w zależności od kierunku przepływu danych (zapis lub odczyt) użytkownik uzyskuje dostęp do jednego z dwóch wewnętrznych rejestrów układu USART – TDR (ang. *Transmit Data Register*) przy zapisie danych lub RDR (ang. *Receive Data Register*) przy odczycie z rejestru DR. Poniżej znajdują się układy kontroli nadawcze i odbiorcze, a także układ kontroli sygnału zegarowego na linii CK przy transmisjach w trybie synchronicznym. Poniżej znajduje się także jednostka odpowiedzialna za generowanie zgłoszenia zapotrzebowania na obsłużenie przerwania. U dołu rys. 5.11. widoczne są elementy odpowiedzialne za generowanie sygnały zegarowego do próbkowania przychodzącego sygnału oraz przy nadawaniu. W celu wykrywania zakłóceń pojawiających się na linii,

sygnał przychodzący jest próbkowany trzy razy w ciągu trwania jednego bitu. Sygnał jest uznawany za niezakłócony w przypadku gdy wszystkie trzy próbki danego bitu mają tą samą wartość logiczną.

Rejestry konfiguracyjne zestawione na rys. 5.10 są przedstawione w sposób uniwersalny do zastosowania w każdym z interfejsów: USART1, USART2 lub USART3. Chcąc w języku C odwołać się do rejestru konkretnego interfejsu, w nazwie rejestru po słowie USART należy podać cyfrę oznaczającą numer interfejsu. Na przykład odwołując się do rejestru USART_CR1 interfejsu USART2 należy użyć konstrukcji USART2−>CR1. Dla różnych interfejsów różne jest też umiejscowienie rejestrów konfiguracyjnych w przestrzeni adresowej procesora. Pierwszy z rejestrów konfiguracyjnych jest widoczny pod adresem:

- 0x4001 3800 – dla USART1,
- 0x4000 4400 – dla USART2,
- 0x4000 4800 – dla USART3.

Znaczenie ważniejszych flag (bitów) oraz obszarów we wspomnianym pierwszym rejestrze konfiguracyjnym USART_SR jest następujące:

- CTS – flaga ustawiana sprzętowo, sygnalizująca zmianę stanu na linii CTS,
- TXE – flaga ustawiana sprzętowo, sygnalizująca pusty rejestr TDR,
- TC – flaga ustawiana sprzętowo sygnalizująca zakończenie transmisji (nadawania),
- RXNE – flaga ustawiana sprzętowo sygnalizująca gotowość danych do odczytania z rejestru RDR,
- NE – flaga ustawiana sprzętowo sygnalizująca wykrycie zakłóceń w danych odbiorczych,
- FE – flaga ustawiana sprzętowo sygnalizująca błąd ramki,
- PE – flaga ustawiana sprzętowo sygnalizująca błąd parzystości.

Kolejny z rejestrów – USARD_DR jak zostało wspomniane wcześniej służy do zapisu danych, które mają być nadawane, bądź odczytu danych, które zostały odebrane.

Rejestr USART_BRR służy do konfiguracji prędkości transmisji (ang. *baud rate*). Składa się on z dwóch pól: 12-bitowej części całkowitej (DIV_Mantisa) i 4-bitowej części ułamkowej (DIV_Fraction). Prędkość transmisji wyznacza się z zależności przedstawionej w równaniu (5.3).

$$Tx(Rx)baud = \frac{f_{CK}}{16 \cdot USARTDIV} \tag{5.3}$$

Gdzie:

f_{CK} - częstotliwość magistrali APB: APB1 dla USART2 i USART2, APB2 dla USART1.

Wartość USARTDIV jest wartością dodatnią, którą należy zakodować w postaci części całkowitej i ułamkowej w rejestrze USART_BRR. W celu zrozumienia zachodzących procesów należy przeanalizować poniższy przykład.

Zakładając, że częstotliwość pracy magistrali APB wynosi 8 MHz, a prędkość transmisji ma wynosi 9,6 kb/s w pierwszej kolejności należy wyznaczyć wartość dzielnika USARTDIV. Po przekształceniach równania (5.3) do postaci (5.4) została wyznaczona wartość USARTDIV równa 52,0833.

$$USARTDIV = \frac{f_{CK}}{16 \cdot Tx(Rx)baud} = \frac{8000000}{16 \cdot 9600} = 52{,}0833 \tag{5.4}$$

Z wyznaczonej wartości należy wyznaczyć część ułamkową poprzez wykonanie działania (5.5) i zaokrąglić do najbliższej całkowitej wartości.

$$DIV\_Fraction = 16 \cdot 0{,}0833 = 1{,}33 \approx 1 = 0x1 \tag{5.5}$$

Następnie z części całkowitej USARTDIV należy wyznaczyć wartość, którą należy zakodować w części całkowitej rejestru USART_BRR według zależności (5.6).

$$DIV\_Mantisa = 52 = 0x34 \tag{5.6}$$

Po złożeniu ze sobą wartości DIV_Mantisa i DIV_Fraction zostaje otrzymana wartość, którą należy wpisać do rejestru USART_BRR, która w omawianym przykładzie wynosi 0x341.

Oprócz prędkości w interfejsie USART ważne są także inne parametry transmisji, których konfiguracja znajduje się w kolejnych rejestrach USART_CR1, -2 i -3. Znaczenie ważniejszych flag (bitów) oraz obszarów w rejestrze USART_CR1:

- UE – flaga ustawiana i kasowana programowa w celu włączenia (1) lub wyłączenia (0) interfejsu USART,
- M – flaga ustawiana i kasowana programowa w celu ustawienia długości słowa: 0 – 1 bit startu, 8 bitów danych, n bitów stop lub 1 – 1 bit startu, 9 bitów danych, n bitów stop,
- PCE – flaga ustawiana i kasowana programowa w celu włączenia (1) lub wyłączenia (0) kontroli parzystości,
- PS – flaga ustawiana i kasowana programowa w celu ustawienia parzystości, 0 – bit parzystości even, 1 – bit parzystości odd,
- TE – flaga ustawiana i kasowana programowa w celu włączenia (1) lub wyłączenia (0) nadajnika interfejsu USART,
- RE – flaga ustawiana i kasowana programowa w celu włączenia (1) lub wyłączenia (0) odbiornika interfejsu USART.

Znaczenie ważniejszych flag (bitów) oraz obszarów w rejestrze USART_CR2:

- STOP – dwubitowe pole ustawiane i kasowane programowo w celu ustawienia ilości bitów stopu (1, 0,5, 2 lub 1,5),
- CLKEN – flaga ustawiana i kasowana programowa w celu włączenia (1) lub wyłączenia (0) sygnału zegarowego na linii CK,

Rejestr USART_CR3 zawiera konfigurację innych trybów pracy interfejsu USART, takich jak Smartcard, IrDA oraz konfigurację sprzętowej kontroli przepływu i DMA. Ostatni z rejestrów USART_GTPR natomiast wykorzystywany jest tylko w przypadku korzystania z trybu pracy Smartcard.

5.6.4. Podstawowy timery TIM6/TIM7

Często w urządzeniach zachodzi potrzeba uruchamiania pewnej czynności co określony czas. Na przykład pomiar wartości przetwornikiem analogowo-cyfrowym, zliczanie impulsów z enkodera w ściśle określonym przedziale czasowym, czy zmiana stanu wyjścia dwustanowego dokładnie co kilkadziesiąt milisekund. Proste procedury opóźniające w programie nie spełniają wówczas swojej roli, gdyż to co jaki czas wykona się dana funkcja zależy od ilości kodu zawartego w programie. Nie można tu w żadnym wypadku mówić o precyzyjnym odliczaniu czasu. Dla zadań, które powinny wykonywać się cyklicznie w trakcie wykonywania programu powstały specjalne układy zwane timerami, które są niezależnymi układami peryferyjnymi zliczającymi impulsy z linii zegarowej (wówczas nazywane są timerami) lub impulsy pojawiające się na określonym wyprowadzeniu mikrokontrolera (wówczas nazywane są licznikami), a po zliczeniu do określonej wartości lub przepełnieniu generujące na przykład zgłoszenie obsłużenia przerwania, w którym to przerwaniu program może realizować cyklicznie wykonywaną czynność.

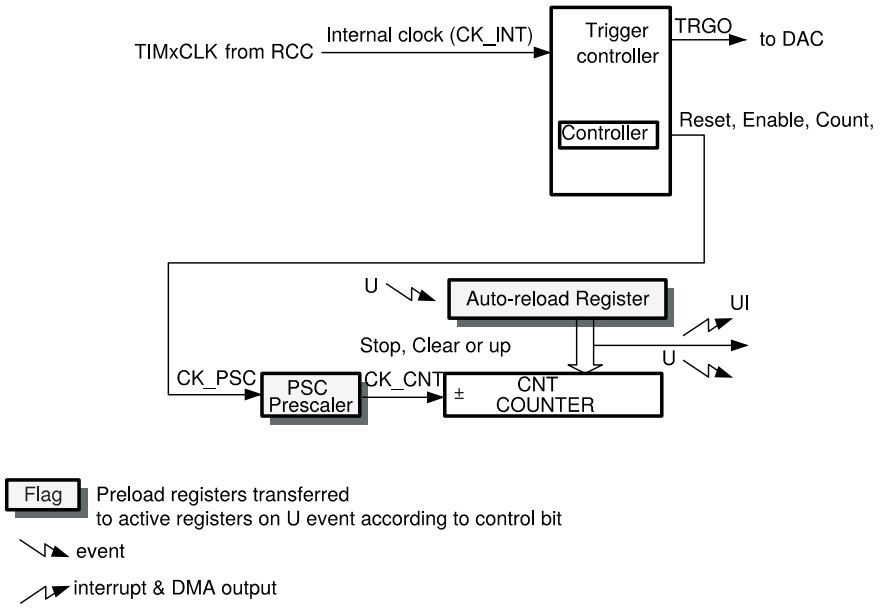
Omawiany mikrokontroler wyposażony jest w 9 timerów o różnym stopniu skomplikowania. Najbardziej zaawansowanym jest timer TIM1, a najmniej timery TIM6

i TIM7. Ze względu na ograniczenia pracy, opis wykorzystania timerów będzie dotyczył tylko tych najprostszych.

Timery TIM6/TIM7 charakteryzują się następującymi cechami:

- 16-bitowy rejestr zliczający,
- 16-bitowy preskaler sygnału zegarowego,
- 16-bitowy rejestr automatycznie-przeładowujący stan początkowy timera,
- obwód synchronizacji z przetwornikiem C/A,
- generowanie żądań obsłużenia przerwań/bezpośredniego dostępu do pamięci przy przepełnieniu.

Schemat wewnętrzny układu timera TIM6 (TIM7) przedstawiony jest na rys. 5.12.



Rys. 5.12. Schemat blokowy timera TIM6/TIM7, opracowano wg [12]

Sygnał zegarowy, którego zbocza przez układ timera są zliczane pochodzi z układu kontroli sygnału zegarowego RCC. Sygnałem tym jest: TIM6CLK dla timera TIM6 i TIM7CLK dla timera TIM7. Sposób jego powstawania przedstawiony jest na rys. 4.33. W celu uruchomienia timera TIM6/TIM7 należy więc w pierwszej kolejności uaktywnić doprowadzanie sygnału zegarowego w rejestrach konfiguracyjnych układu RCC tak jak w przypadku każdego innego urządzenia peryferyjnego. Sygnał zegarowy trafia do układu kontroli wyzwalania. Układ ten umożliwia synchronizację kilku timerów ze sobą (na przykład w celu synchronicznego startu zliczania zboczy sygnału zegarowego. Następnie sygnał trafia do preskalera PSC. Jest to układ, który potrafi podzielić przychodzący sygnał przez pewną wartość. W omawianym przypadku może to być wartość 16-bitowa, czyli podział sygnału zegarowego może następować w zakresie od 1 do 65535. Wpisanie liczby, przez którą ma nastąpić podział dokonuje się w rejestrze TIMx_PSC. Następnie sygnał po podzieleniu (już o nazwie CK_CNT) trafia do układu rejestru zliczającego. Rejestr ten (TIMx_CNT) po pojawieniu się każdego kolejnego zbocza narastającego sygnału zwiększa swoją wartość począwszy od wartości 0. Wartość jest zwiększana aż do osiągniecia wartości automatycznie-przeładowującej (TIMx_ARR) , po osiągnieciu której może (w zależności od konfiguracji) zostać zatrzymany timer, bądź uruchomione zliczanie od nowa, czyli od wartości 0. Chwila osiągniecia wartości w rejestrze TIMx_ARR nazywana także przepełnieniem timera, może również być źródłem wygenerowania zapotrzebowania obsłużenia przerwania czy też sygnałem sterującym dla wybranego kanału DMA.

Konfiguracja timerów TIM 6 i TIM7 zawarta jest w rejestrach przedstawionych na rys. 5.13. Symbol „x” w nazwach rejestrów należy zamienić na cyfrę 6 lub 7 w zależności

od tego, którego timera TIM6 czy TIM7 ma dotyczyć konfiguracja. Pierwszy rejestr konfiguracyjny timera TIM6 o nazwie TIM6_CR1 znajduje się w pamięci pod adresem 0x4000 1000, natomiast analogiczny pierwszy rejestr timera TIM7 pod adresem 0x4000 1400.

Offset	Register	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0x00	**TIMx_CR1**	Reserved																								ARPE	Reserved			OPM	URS	UDIS	CEN
	Reset value																									0				0	0	0	0
0x04	**TIMx_CR2**	Reserved																									MMS[2:0]			Reserved				
	Reset value																										0	0	0				
0x08	Reserved																																
0x0C	**TIMx_DIER**	Reserved																							UDE	Reserved							UIE
	Reset value																								0								0
0x10	**TIMx_SR**	Reserved																															UIF
	Reset value																																0
0x14	**TIMx_EGR**	Reserved																															UG
	Reset value																																0
0x18	Reserved																																
0x1C	Reserved																																
0x20	Reserved																																
0x24	**TIMx_CNT**	Reserved																CNT[15:0]															
	Reset value																	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0x28	**TIMx_PSC**	Reserved																PSC[15:0]															
	Reset value																	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0x2C	**TIMx_ARR**	Reserved																ARR[15:0]															
	Reset value																	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

Rys. 5.13. Rejestry konfiguracyjne timerów TIM6/TIM7, opracowano wg [12]

Znaczenie ważniejszych flag (bitów) oraz obszarów w rejestrze TIMx_CR1:

- ARPE – flaga odczytywana i zapisywana programowo określająca tryb przeładowania rejestru TIMx_ARR. Jeżeli jest ustawiona, to zmiana wartości w rejestrze TIMx_ARR odnosi skutek dopiero po przepełnieniu timera, inaczej mówiąc, po zmianie wartości przepełnienie timera nastąpi jeszcze do starej wartości, a dopiero kolejne do nowej,
- OPM – flaga odczytywana i zapisywana programowo określająca tryb pracy timera: 0 – praca ciągła, 1 – praca jednorazowa, czyli timer zlicza tylko jeden raz i po osiągnieciu przepełnienia zatrzymuje się,
- CEN – flaga odczytywana i zapisywana programowo załączająca (1) lub wyłączająca (0) zliczanie impulsów. W pracy jednorazowej, flaga jest kasowana sprzętowo po przepełnieniu.

Rejestr TIMx_CR2 zawiera pole MMS, w którym konfiguruje się tryb synchronizacji z pozostałymi timerami. Rejestr TIMx_DIER zawiera natomiast flagi konfigurujące żądanie przerwań i zgłoszeń do DMA. Znaczenie ważniejszych flag (bitów) oraz obszarów w tym rejestrze:

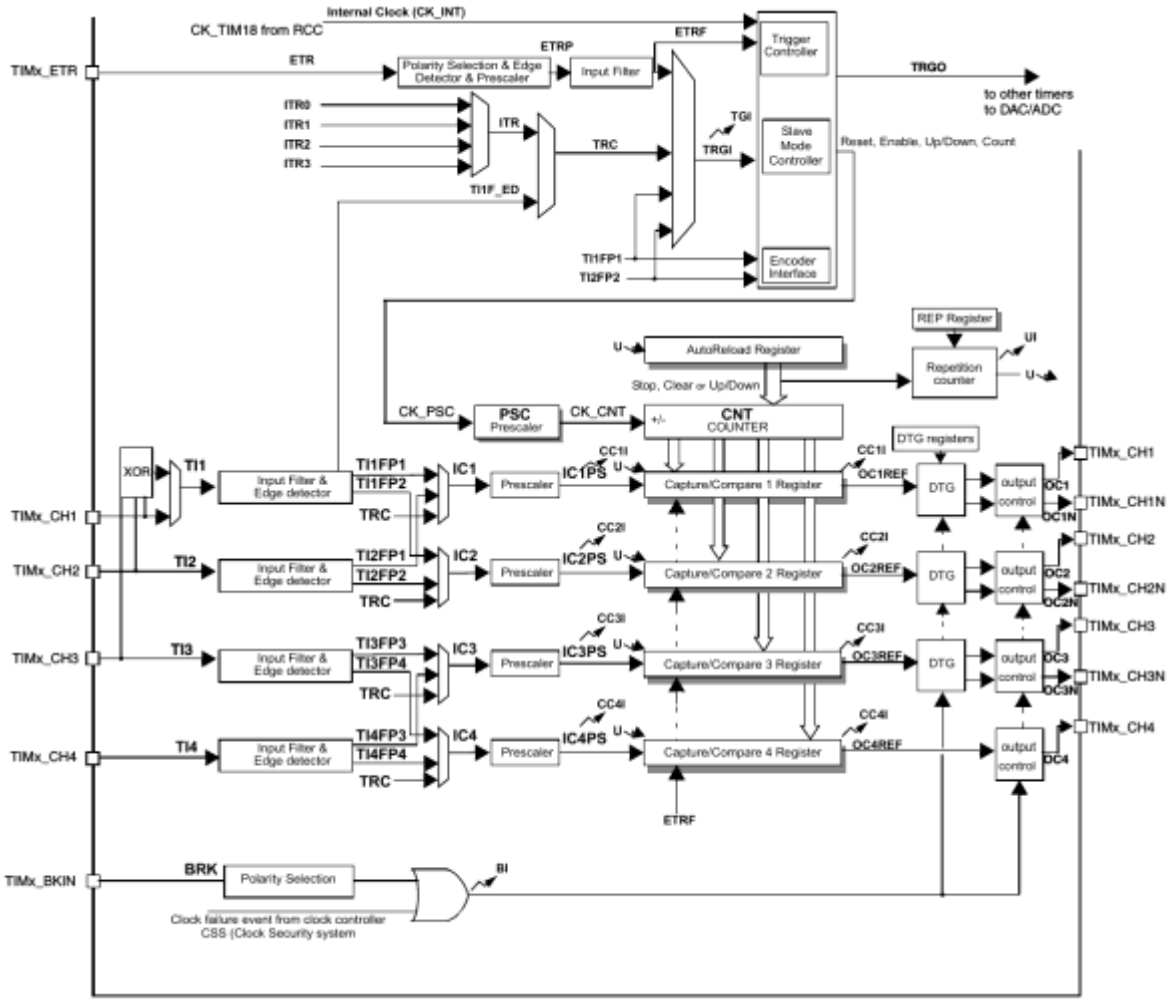
- UDE – flaga odczytywana i zapisywana programowo załączająca (1) lub wyłączająca (0) zgłoszenie do DMA,
- UIE – flaga odczytywana i zapisywana programowo załączająca (1) lub wyłączająca (0) żądanie obsłużenia przerwania.

Znaczenie ważniejszych flag (bitów) oraz obszarów w rejestrze TIMx_SR:

- UIF – flaga ustawiana sprzętowo i kasowana programowo określająca wystąpienie zgłoszenia żądania obsłużenia przerwania od przepełnienia timera.

Rejestr TIMx_CNT jest 16-bitowym rejestrem przechowującym aktualnie zliczoną liczbę zboczy sygnału zegarowego. Rejestr TIMx_PSC jest również 16-bitowym rejestrem, ale określającym współczynnik podziału sygnału zegarowego (od 1 do 65535), natomiast rejestr TIMx_ARR zawiera liczbę, po osiągnięciu której w rejestrze TIMx_CNT nastąpi przepełnienie timera i uruchomienie zliczania od początku czyli od wartości 0.

Dla porównania jak relatywnie prostym układem są timery TIM6/TIM7 na rys. 5.14 przedstawiono schemat blokowy timera TIM1.



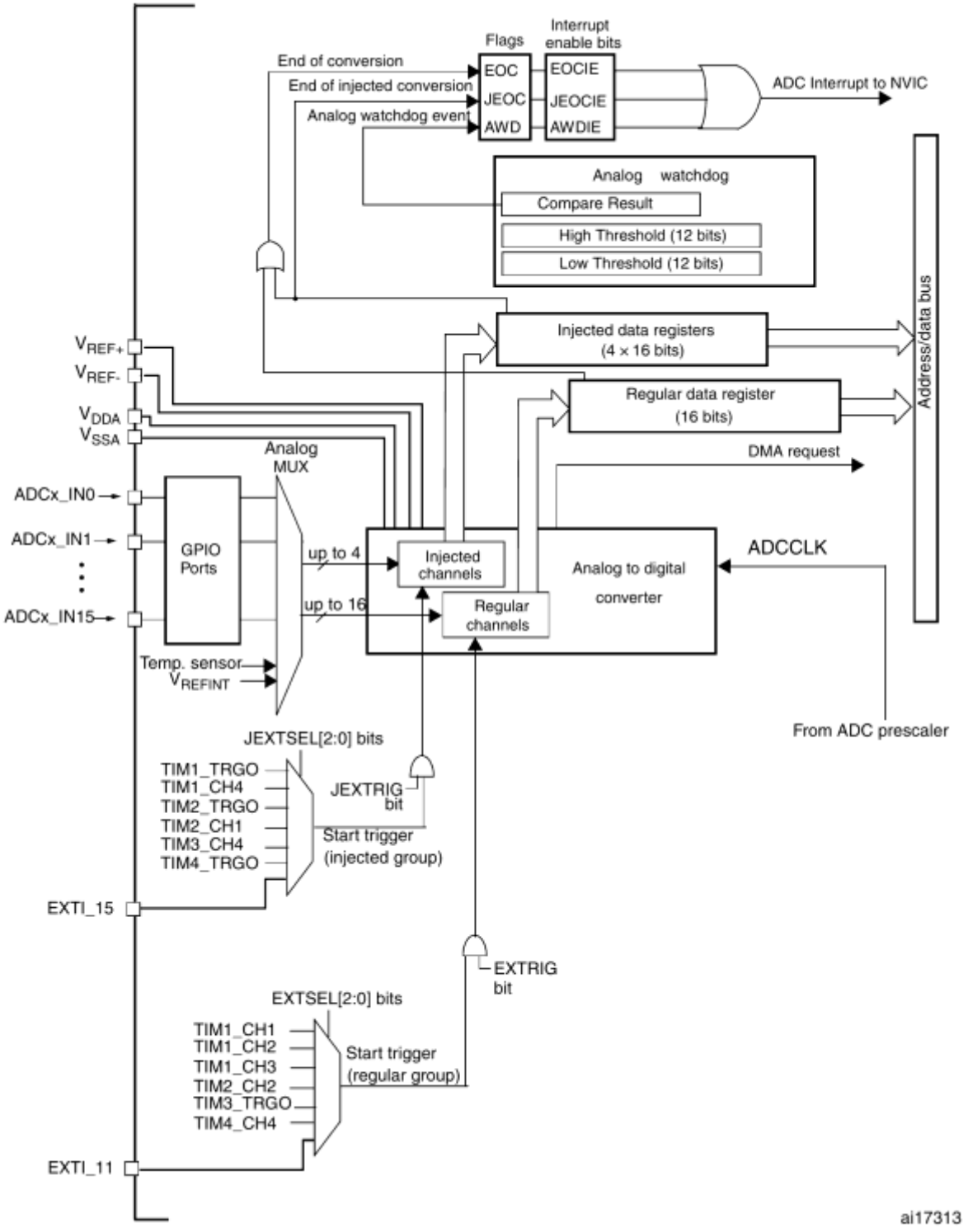
Rys. 5.14. Schemat blokowy timera TIM1, opracowano wg [12]

Opis jego wykracza jednak po za ramy niniejszej pracy magisterskiej.

5.6.5. Przetwornik analogowo-cyfrowy

Omawiany mikrokontroler wyposażony jest w przetwornik analogowo-cyfrowy (A/C). Przetwornik taki ma zastosowanie w przypadku pomiarów wielkości analogowych takich jak temperatura, napięcie, natężenie prądu, rezystancja oraz inne po zastosowaniu

wcześniejszych bloków kondycjonujących sygnał do postaci akceptowalnej przez wejście przetwornika analogowego.



Rys. 5.15. Schemat blokowy wbudowanego przetwornika analogowo-cyfrowego, opracowano wg [12]

Schemat budowy wewnętrznej omawianego przetwornika analogowo-cyfrowego został przedstawiony na rys. 5.15. Charakteryzuje się on następującymi cechami:

- zasada pomiaru: sukcesywna aproksymacja,
- rozdzielczość: 12 bitów,
- ilość multipleksowanych kanałów: 18 (16 zewnętrznych i 2 wewnętrzne),
- czas przetwarzania: 1,17 µs przy taktowaniu 24 MHz,

- zakres pomiarowy: od V_{SSA} do V_{DDA} (maksymalnie 3,6 V),
- autokalibracja,
- możliwość zgłaszania żądań obsłużenia przerwania,
- współpraca z DMA,
- przetwarzanie pojedyncze, ciągłe, ciągłe dla wielu kanałów,
- watchdog pilnujący ustawionego min i max dla mierzonej wartości.

Przetwornik posiada największą spośród omówionych urządzeń peryferyjnych liczbę rejestrów konfiguracyjnych. Zostały one przedstawione na rys. 5.16 i rys. 5.17.

Offset	Register	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0x00	**ADC_SR**	Reserved																											STRT	JSTRT	JEOC	EOC	AWD
	Reset value																												0	0	0	0	0
0x04	**ADC_CR1**	Reserved								AWDEN	JAWDEN	Reserved						DISC NUM [2:0]			JDISCEN	DISCEN	JAUTO	AWD SGL	SCAN	JEOC IE	AWDIE	EOCIE	AWDCH[4:0]				
	Reset value									0	0							0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0x08	**ADC_CR2**	Reserved								TSVREFE	SWSTART	JSWSTART	EXTTRIG	EXTSEL [2:0]			Reserved	JEXTTRIG	JEXTSEL [2:0]			ALIGN	Reserved		DMA	Reserved			RSTCAL	CAL	CONT	ADON
	Reset value									0	0	0	0	0	0	0		0	0	0	0	0			0				0	0	0	0
0x0C	**ADC_SMPR1**	Sample time bits SMPx_x																															
	Reset value	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0x10	**ADC_SMPR2**	Sample time bits SMPx_x																															
	Reset value	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0x14	**ADC_JOFR1**	Reserved																					JOFFSET1[11:0]										
	Reset value																						0	0	0	0	0	0	0	0	0	0	0
0x18	**ADC_JOFR2**	Reserved																					JOFFSET2[11:0]										
	Reset value																						0	0	0	0	0	0	0	0	0	0	0
0x1C	**ADC_JOFR3**	Reserved																					JOFFSET3[11:0]										
	Reset value																						0	0	0	0	0	0	0	0	0	0	0
0x20	**ADC_JOFR4**	Reserved																					JOFFSET4[11:0]										
	Reset value																						0	0	0	0	0	0	0	0	0	0	0
0x24	**ADC_HTR**	Reserved																					HT[11:0]										
	Reset value																						0	0	0	0	0	0	0	0	0	0	0
0x28	**ADC_LTR**	Reserved																					LT[11:0]										
	Reset value																						0	0	0	0	0	0	0	0	0	0	0
0x2C	**ADC_SQR1**	Reserved								L[3:0]				SQ16[4:0] 16th conversion in regular sequence bits					SQ15[4:0] 15th conversion in regular sequence bits					SQ14[4:0] 14th conversion in regular sequence bits					SQ13[4:0] 13th conversion in regular sequence bits				
	Reset value									0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0x30	**ADC_SQR2**	Reserved	SQ12[4:0] 12th conversion in regular sequence bits					SQ11[4:0] 11th conversion in regular sequence bits						SQ10[4:0] 10th conversion in regular sequence bits					SQ9[4:0] 9th conversion in regular sequence bits					SQ8[4:0] 8th conversion in regular sequence bits					SQ7[4:0] 7th conversion in regular sequence bits				
	Reset value		0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0x34	**ADC_SQR3**	Reserved	SQ6[4:0] 6th conversion in regular sequence bits					SQ5[4:0] 5th conversion in regular sequence bits						SQ4[4:0] 4th conversion in regular sequence bits					SQ3[4:0] 3rd conversion in regular sequence bits					SQ2[4:0] 2nd conversion in regular sequence bits					SQ1[4:0] 1st conversion in regular sequence bits				
	Reset value		0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

Rys. 5.16. Rejestry konfiguracyjne przetwornika A/C – część 1, opracowano wg [12]

Rejestr ADC_SR zawiera flagi informujące o rozpoczęciu i zakończeniu przetwarzania oraz flagę od analogowego watchdoga. Dwa kolejne rejestry ADC_CR1 i ADC_CR2 są rejestrami konfigurującymi pracę przetwornika.

Offset	Register	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0x38	**ADC_JSQR**	Reserved										JL[1:0]		JSQ4[4:0] 4th conversion in injected sequence bits					JSQ3[4:0] 3rd conversion in injected sequence bits					JSQ2[4:0] 2nd conversion in injected sequence bits					JSQ1[4:0] 1st conversion in injected sequence bits				
	Reset value											0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0x3C	**ADC_JDR1**	Reserved																	JDATA[15:0]														
	Reset value																		0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0x40	**ADC_JDR2**	Reserved																	JDATA[15:0]														
	Reset value																		0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0x44	**ADC_JDR3**	Reserved																	JDATA[15:0]														
	Reset value																		0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0x48	**ADC_JDR4**	Reserved																	JDATA[15:0]														
	Reset value																		0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0x4C	**ADC_DR**	Reserved																	Regular DATA[15:0]														
	Reset value																		0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

Rys. 5.17. Rejestry konfiguracyjne przetwornika A/C – część 2, opracowano wg [12]

Rejestry ADC_SMPR1 i ADC_SMPR2 określają czas przetwarzania w cyklach zegara. Istnieje możliwość zmiany czasu przetwarzania w granicach od 1,5 cyklu do 239,5 cyklu zegara. Przy czym całkowity czas przetwarzania wynosi czas zaprogramowany w rejestrze ADC_SMPR1/2 powiększony o 12,5 cyklu zegara. Przy taktowaniu zegarem 24 MHz, sygnał zegarowy dla przetwornika ADC wynosi 12 MHz, co oznacza, że najkrótszy możliwy czas przetwarzania wynosi 1,17 µs. Rejestry ADC_JOFR1…4 umożliwiają wprowadzenie offsetu do pomiarów. 12-bitowe rejestry ADC_HTR i ADC_LTR służą do wprowadzenia wartości minimalnej i maksymalnej dla analogowego watchdoga, po przekroczeniu których można skonfigurować wystąpienie żądania obsłużenia przerwania. Rejestry ADC_SQR1…3 oraz ADC_JSQR umożliwiają wprowadzenie określonej sekwencji przetwarzania poszczególnych kanałów analogowych. Ostatnie rejestry, to znaczy: ADC_JDR1…4 oraz ADC_DR przechowują już wynik przetwarzania analogowo-cyfrowego.

Dokładny opis rejestrów oraz samego przetwornika A/C wykracza po za ramy niniejszej pracy magisterskiej. Więcej treści z nim związanych znajduje się w dokumentacji producenta [12].

Interfejs USART oraz kilka innych wybranych układów peryferyjnych omawianego mikrokontrolera zostały przetestowane podczas testów funkcjonalnych wybranego rozwiązania, jakim było zabudowanie całego zestawu ewaluacyjnego w roli nowej części mikroprocesorowej zestawu DSM-51.

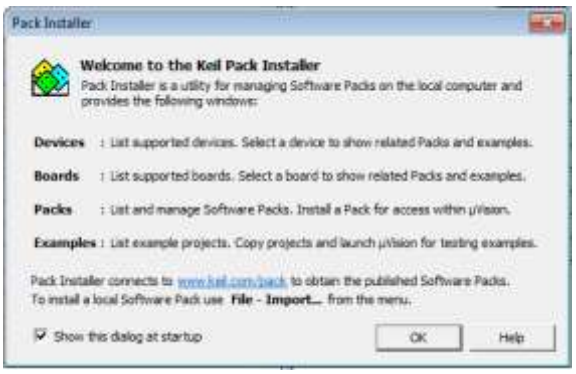
6. Programowanie mikrokontrolerów ARM Cortex

6.1. Konfiguracja środowiska programistycznego

Producent poleca trzy środowiska programistyczne dla mikrokontrolerów ARM, są to:

- Atolic, TrueSTUDIO®,
- IAR, Embedded Workbench® for ARM,
- Keil, MDK-ARM™.

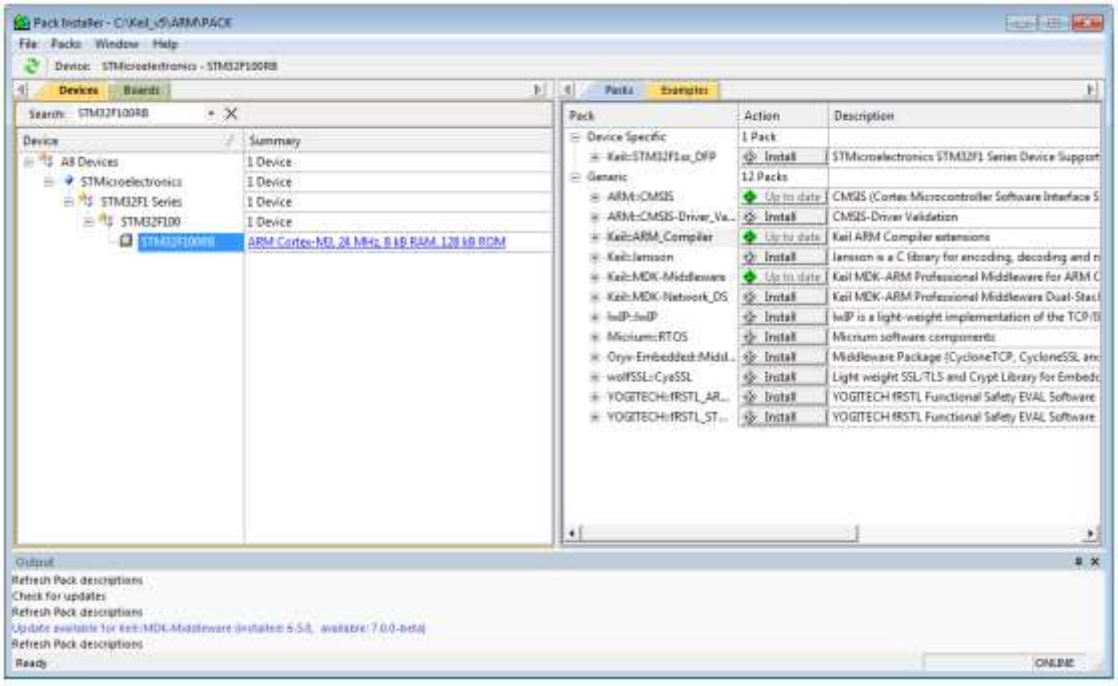
W niniejszym opisie autor zdecydował się na użytkowanie środowiska firmy Keil ze względu na to, że jest to środowisko znane już studentom Wydziału Elektrycznego Politechniki Warszawskiej z laboratorium Podstaw Techniki Mikroprocesorowej oraz na fakt, iż dla projektów nieprzekraczających 32 kB pamięci programu jest dostępny bez opłat licencyjnych.



Rys. 6.1 Widok okna z wyborem pakietów przy pierwszym uruchomieniu środowiska Keil MDK-ARM

W pierwszym etapie należy pobrać MDK-Core ze strony producenta Keil (http://www2.keil.com/mdk5/install/). W celu pobrania należy podać dane do kontaktu. Rozmiar pliku instalacyjnego *MDK517.EXE* wynosi 395 MB. Następnie należy przystąpić do instalacji pobranego oprogramowania. W trakcie instalacji należy postępować zgodnie ze wskazówkami instalatora, na pytanie o zgodę na zainstalowanie sterownika uniwersalnej magistrali systemowej należy odpowiedzieć twierdząco. Po prawidłowym zainstalowaniu należy uruchomić oprogramowanie Keil µVision. Przy pierwszym uruchomieniu zostanie przedstawione okno widoczne na rys. 6.2. W oknie tym zarządzamy pobieraniem pakietów oprogramowania dla konkretnych mikrokontrolerów. Środowisko Keil MDK-ARM™ wspiera wielu producentów i wiele rodzajów mikrokontrolerów. Domyślne instalowanie bibliotek i oprogramowania dla wszystkich wspieranych układów zajmowałoby niepotrzebnie dużą ilość miejsca na dysku twardym komputera, w związku z tym producent postanowił na pozostawienie użytkownikowi możliwości instalowania wybranych przez siebie modułów dla danego typu mikroprocesora. Producent przedstawia swoje środowisko programistyczne jako dwuczęściowe – pierwsza część (MDK-Core), która zostaje zainstalowana na komputerze zawiera w swoich strukturach m.in. kompilator oraz edytor i mechanizm instalowania odpowiednich bibliotek. Druga część to właśnie omawiane biblioteki, które należy zainstalować według potrzeb. Przed instalacją odpowiedniej paczki z oprogramowaniem należy zaktualizować obecnie zainstalowane. W tym celu należy klikać w żółte pola z napisem Update w prawej części okna Pack Installer. W opisywanym zestawie ewaluacyjnym występuje mikrokontroler STM32F100RB i taki też model należy wpisać w oknie „Search” w lewej części okna Pack Installer. Zostanie wyświetlone okno przedstawione na rys. 6.3. W nim należy kliknąć myszą na nazwę STM32F100RB wyświetlaną w lewej części okna, wówczas w prawej części okna w grupie Device Specific pojawi się nowy element, na którym należy kliknąć

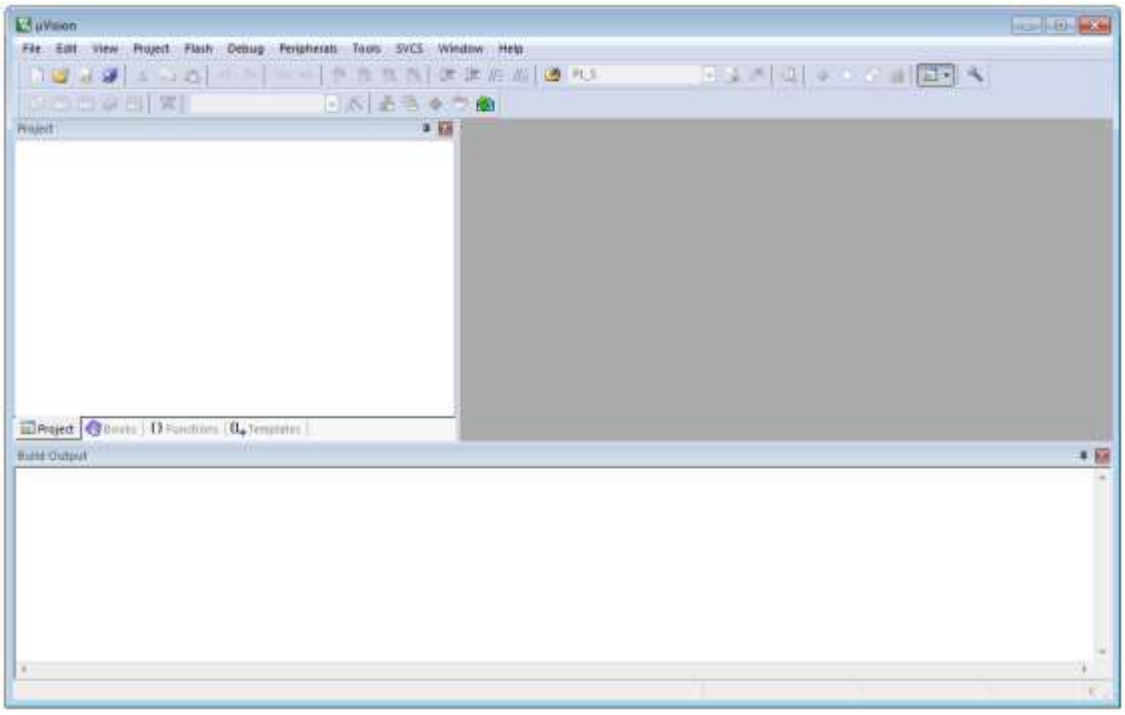
przycisk Install. Po prawidłowym zainstalowaniu oprogramowania, nazwa Install na przycisku powinna zmienić się w Up to date. W ten sposób zostało przygotowane środowisku Keil MDK-ARM™ do tworzenia projektów na mikrokontroler STM32F100RB.



Rys. 6.2. Widok okna z wyborem bibliotek w środowisku Keil MDK-ARM

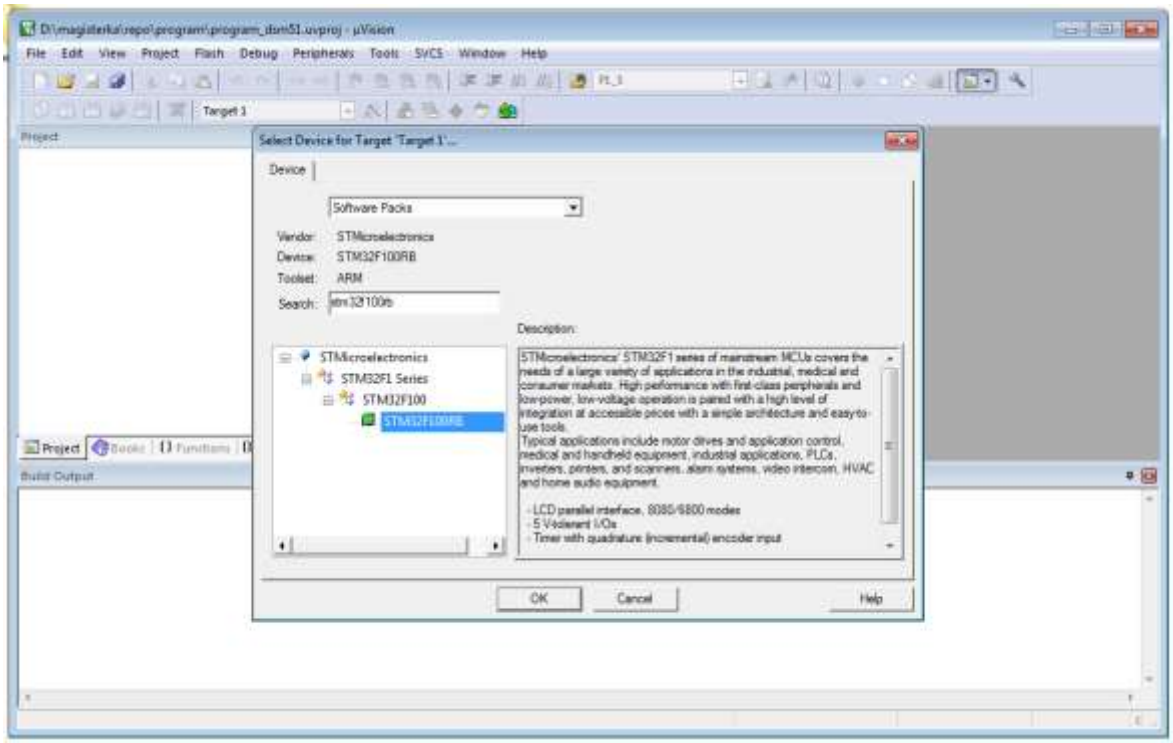
6.2. Tworzenie nowego projektu

Po uruchomieniu środowiska Keil µVision zostanie przedstawione okno widoczne na rys. 6.4.



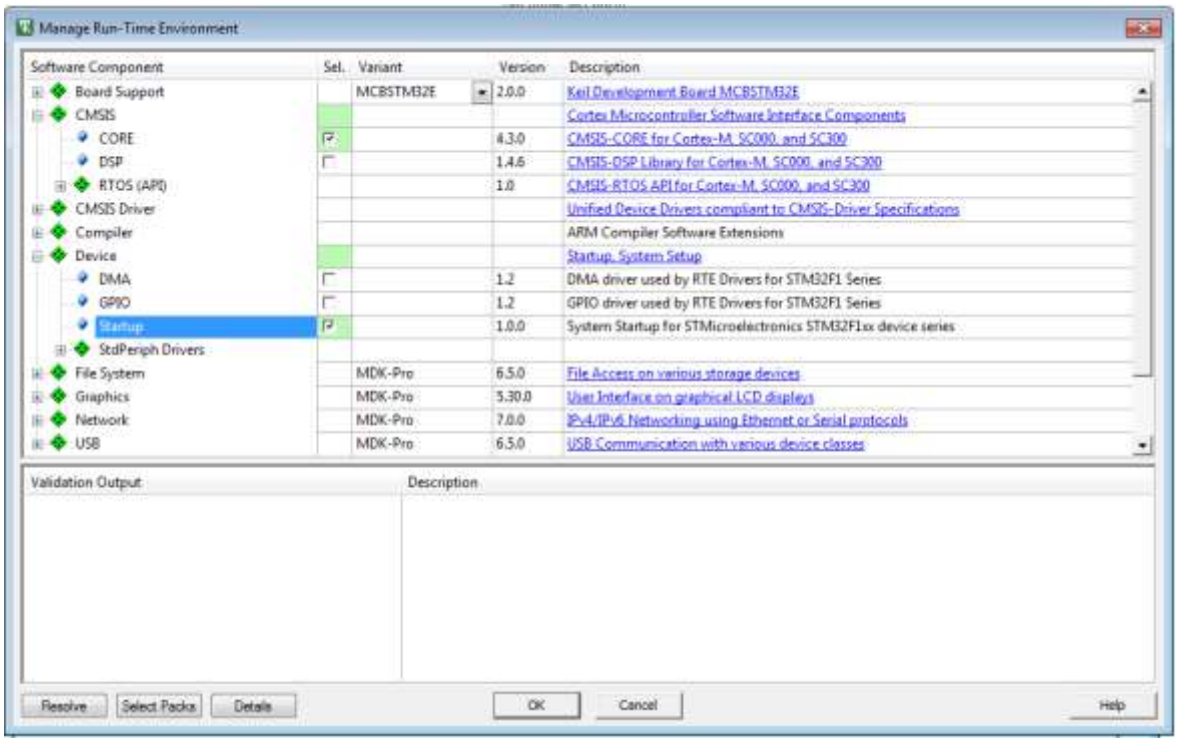
Rys. 6.3. Widok głównego okna środowiska Keil MDK-ARM przy braku nowego projektu

W nim należy z menu Project wybrać opcję New µVision Project… .



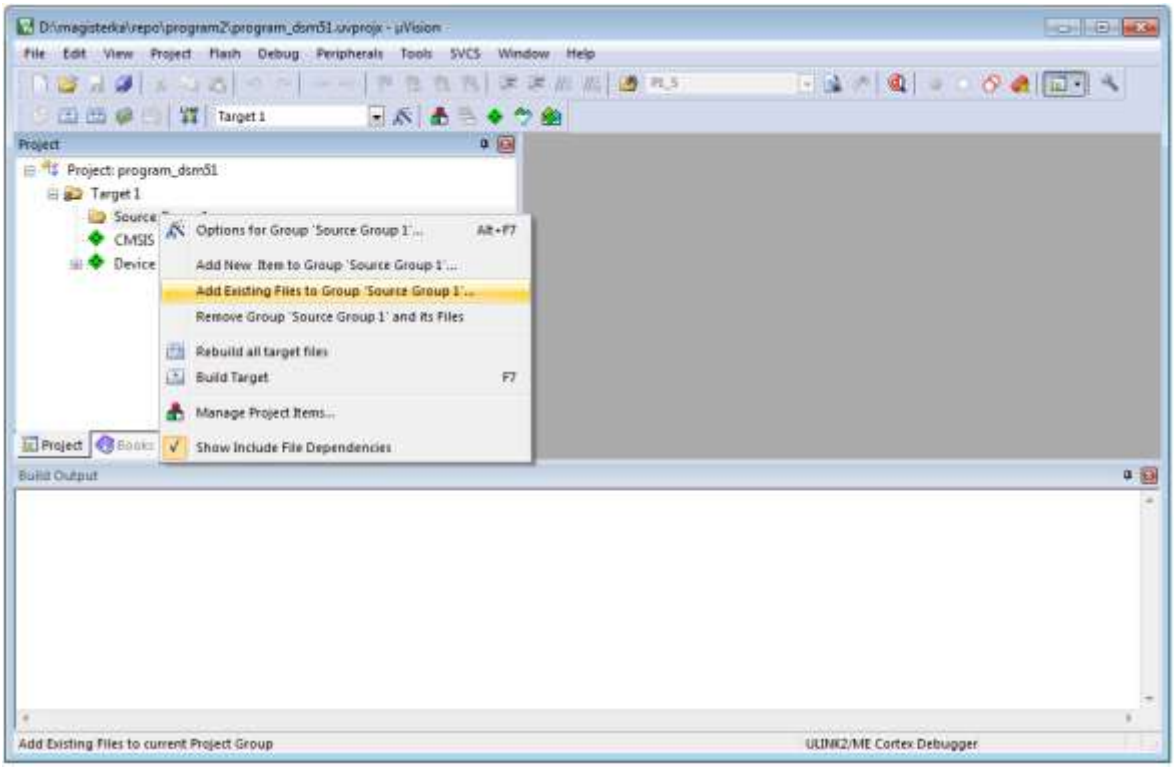
Rys. 6.4. Pierwszy krok tworzenia nowego projektu w Keil MDK-ARM

Zostanie wówczas przedstawione okno przedstawione na rys. 6.5., w którym to dokonuje się wyboru modelu mikrokontrolera, którego dotyczy projekt. W trakcie wyboru można wykorzystywać okienko Search. Po zatwierdzeniu wyboru, środowisko prosi użytkownika o określenie, z jakich dostępnych sterowników i bibliotek będzie w danym projekcie korzystał. Do celów niniejszej pracy magisterskiej zostają wykorzystane dwie biblioteki: CMSIS-Core i System Startup for STMicroelectronics.



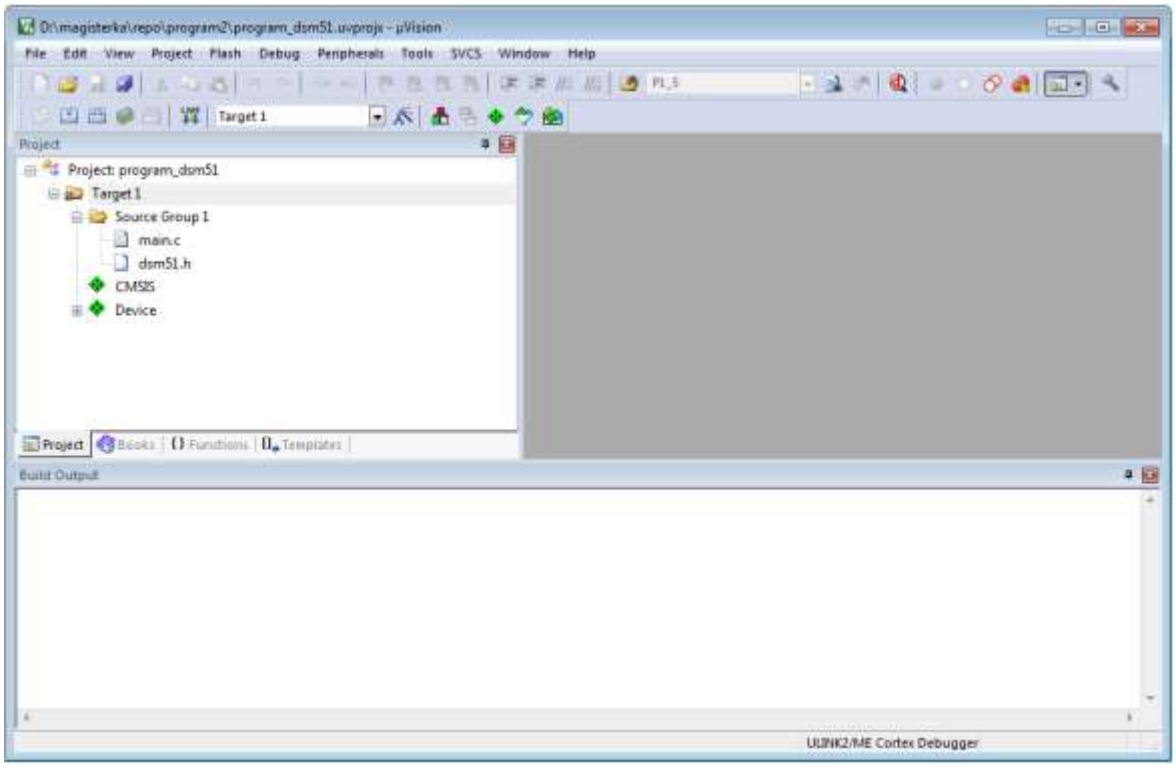
Rys. 6.5. Wybór odpowiednich bibliotek dla nowego projektu

Taki wybór został przedstawiony na rys. 6.6.



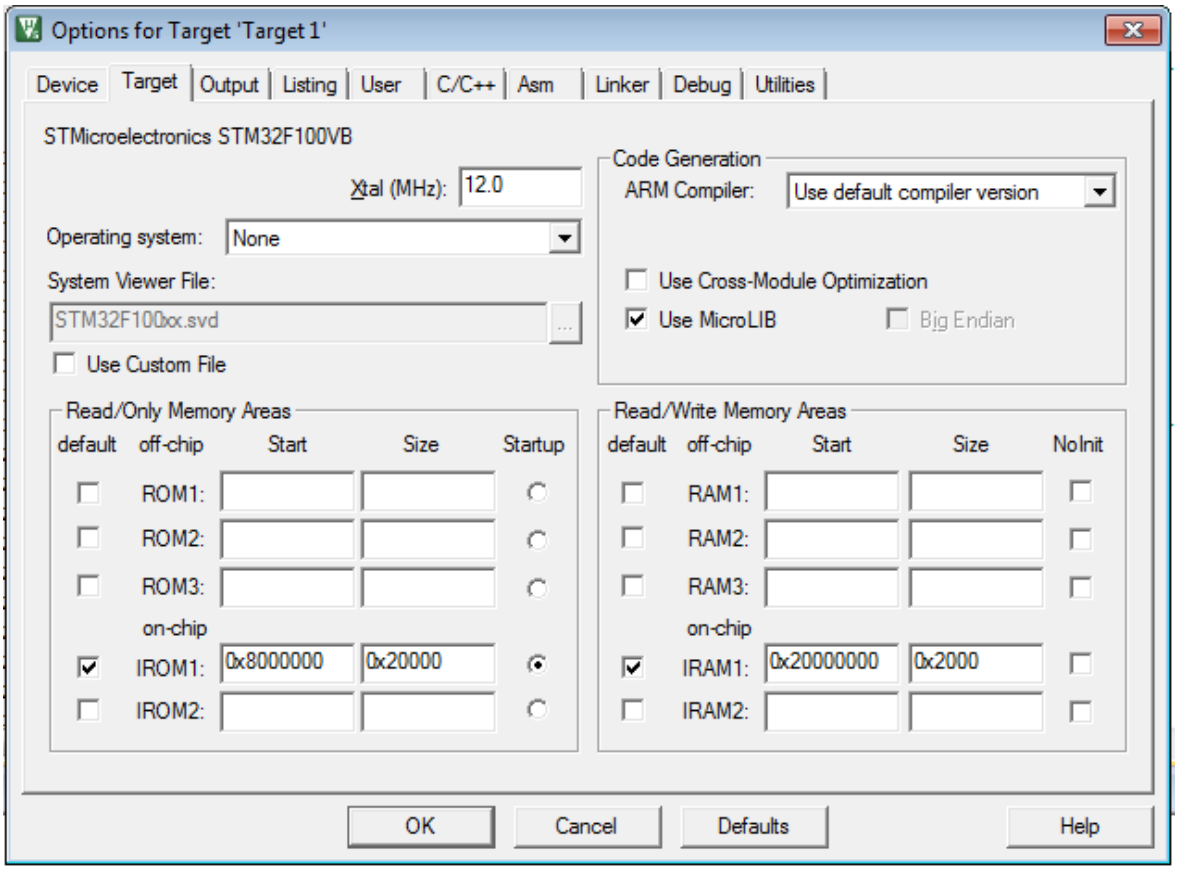
Rys. 6.6. Dodawanie istniejących plików do nowego projektu

Biblioteka CMSIS-Core zapewnia możliwość odwoływania się do rejestrów mikroprocesora poprzez ich nazwy, a nie operowanie na adresach pamięci. System Startup natomiast pozwala na uruchomienie napisanych, skompilowanych i przesłanych do mikrokontrolera programów.

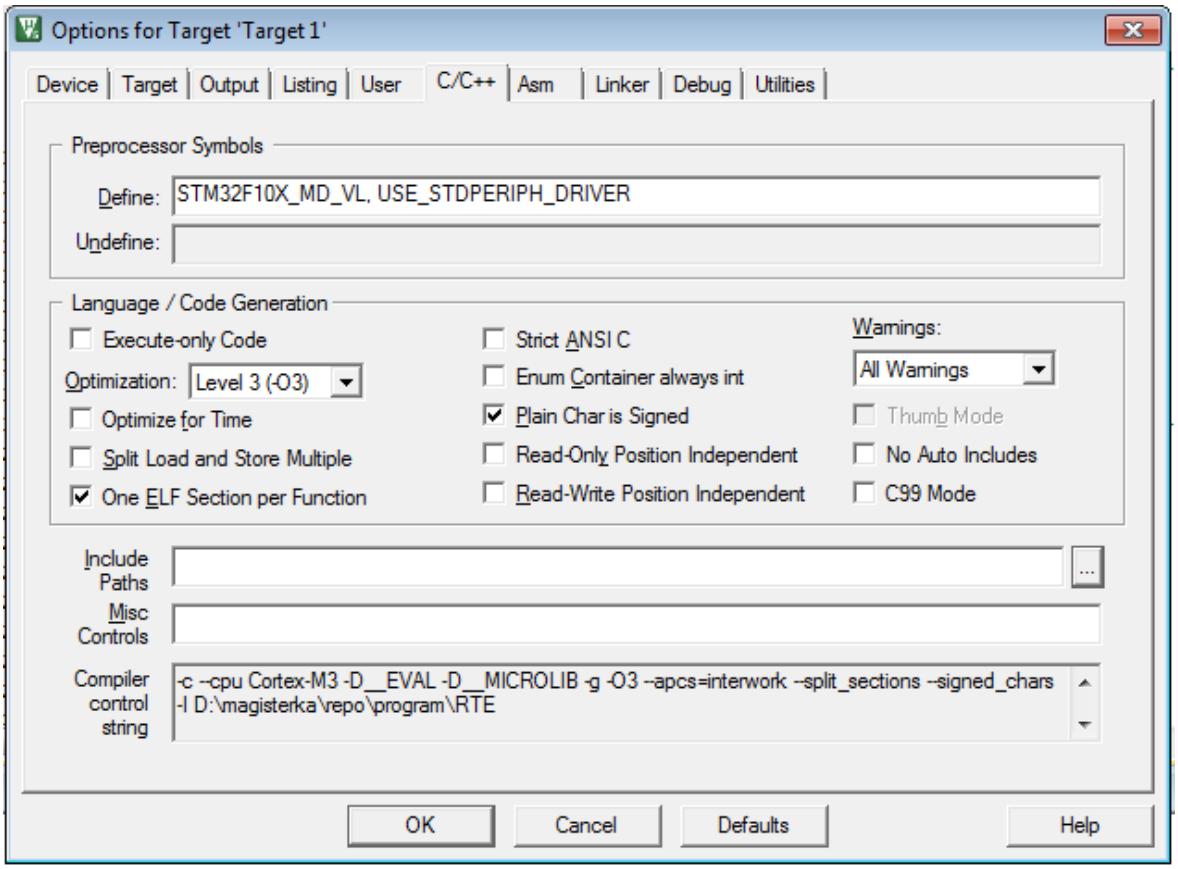


Rys. 6.7. Widok drzewa projektu z dodanymi niezbędnymi plikami

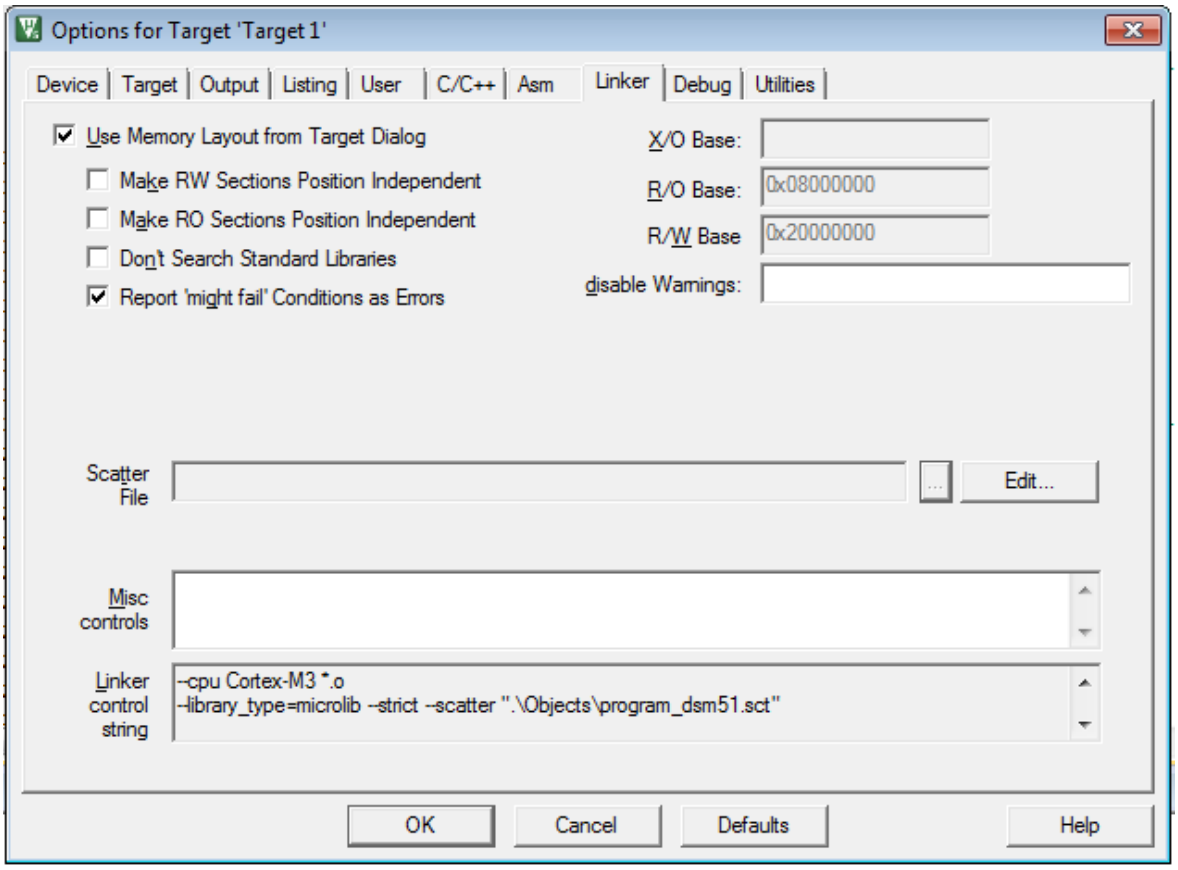
Autor niniejszej pracy zrezygnował z pozostałych bibliotek, które w profesjonalnych rozwiązaniach są stosowane, lecz zaciemniają obraz działania mikrokontrolera i pozbawiają napisany program wartości dydaktycznych. Po zatwierdzeniu wyboru bibliotek można przystąpić do pisania pierwszego programu od podstaw, bądź skorzystać z programu demonstracyjnego napisanego przez autora pracy magisterskiej. W tym celu należy w drzewie projektu po lewej stronie środowiska µVision kliknąć prawym przyciskiem myszy na katalog Source Group 1 i wybrać Add Existing Files to Group 'Source Group 1'…. co zostało przedstawione na rys. 6.7. Tam należy wybrać dwa pliki z dołączonej płyty CD znajdujące się w katalogu Program_DEMO o nazwie main.c oraz dsm51.h. Po poprawnie zakończonej operacji drzewo projektu powinno wyglądać tak ja na rys. 6.8. Ostatnim już etapem przed pierwszą kompilacją jest ustawienie opcji kompilacji, linkowania i debugera. Opcje te dostępne są po kliknięciu prawym przyciskiem myszy w katalog Target znajdujący się w drzewie projektu i wybraniu z menu kontekstowego Options for Target 'Target 1'…. Opcje podzielone są na kilka zakładek, wśród których należy dokonać zmian w następujących: Target, C/C++, Linker, Debug i Utilites. Dodatkowo w zakładce Utilites po kliknięciu w przycisk Settings należy wybrać rodzaj programatora. Ustawienia wszystkich opcji przedstawione zostały na rysunkach od rys. 6.9. do rys. 6.13. Na rys. 6.14. zostały przedstawione opcje programatora, do których dostęp uzyskuje się poprzez naciśnięcie przycisku Settings w zakładce Utility ustawień projektu. W ustawieniach programatora zarówno w zakładce Debug jak i Flash Download należy wybrać typ programatora ST-Link, natomiast port ustawić w tryb SW (od ang. *Serial Wire Interface*), gdyż zabudowany programator ST-Link na płytce ewaluacyjnej posiada jedynie ten interfejs programowania mikrokontrolerów.



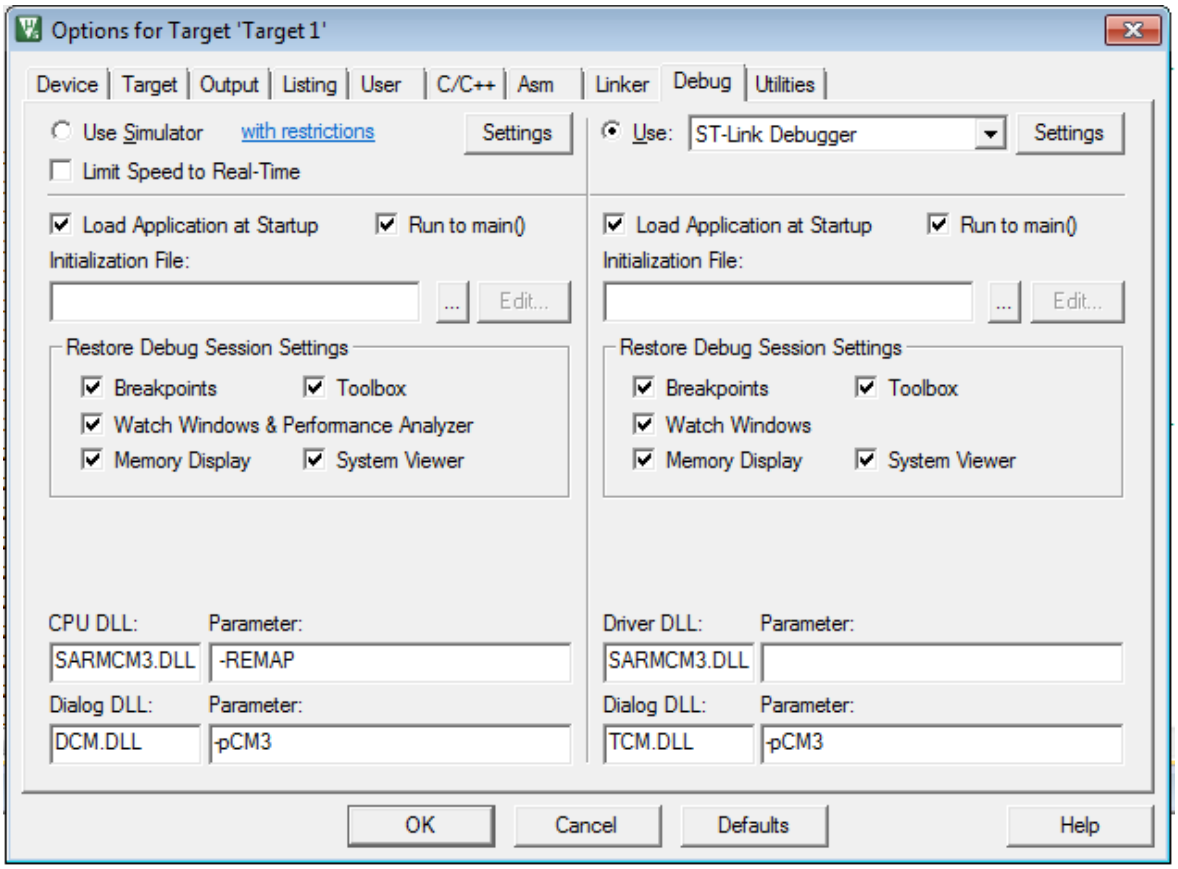
Rys. 6.8. Widok zakładki Target w opcjach projektu



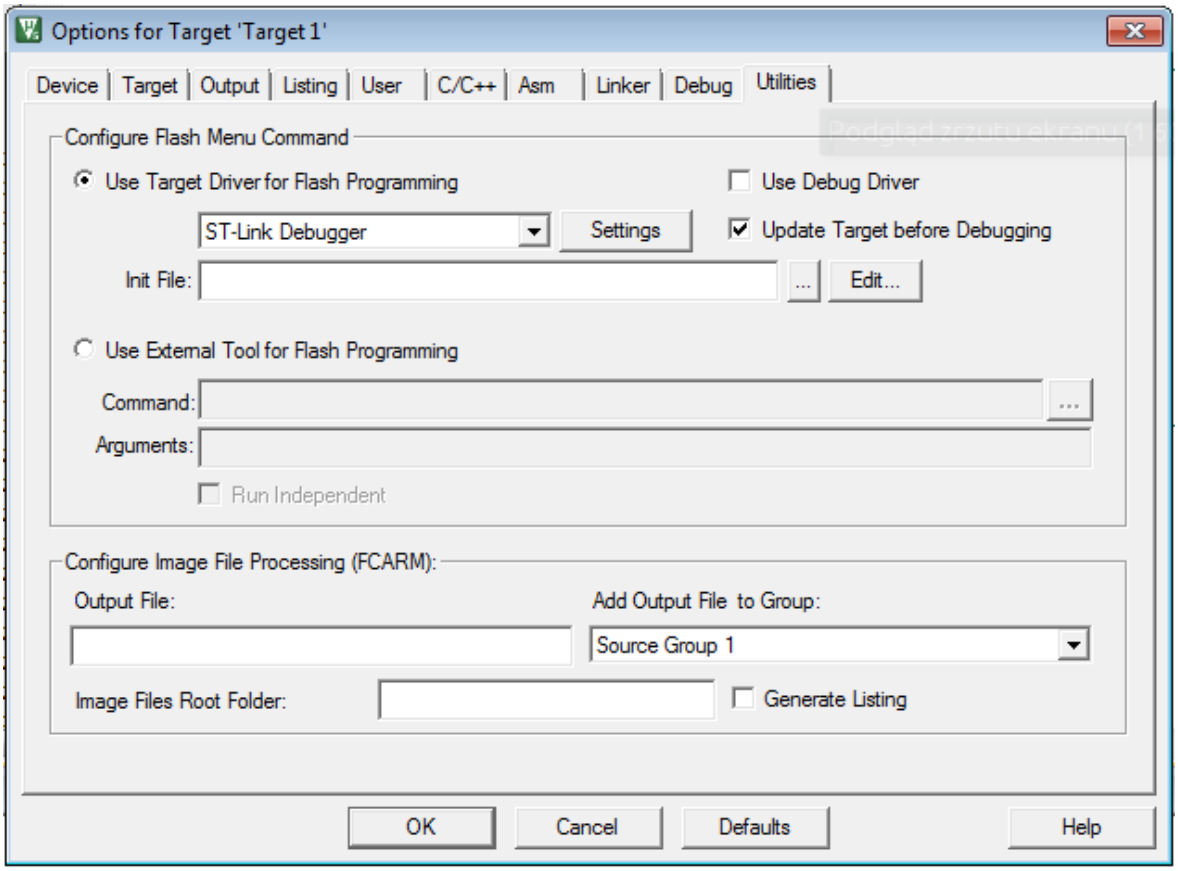
Rys. 6.9. Widok zakładki C/C++ w opcjach projektu



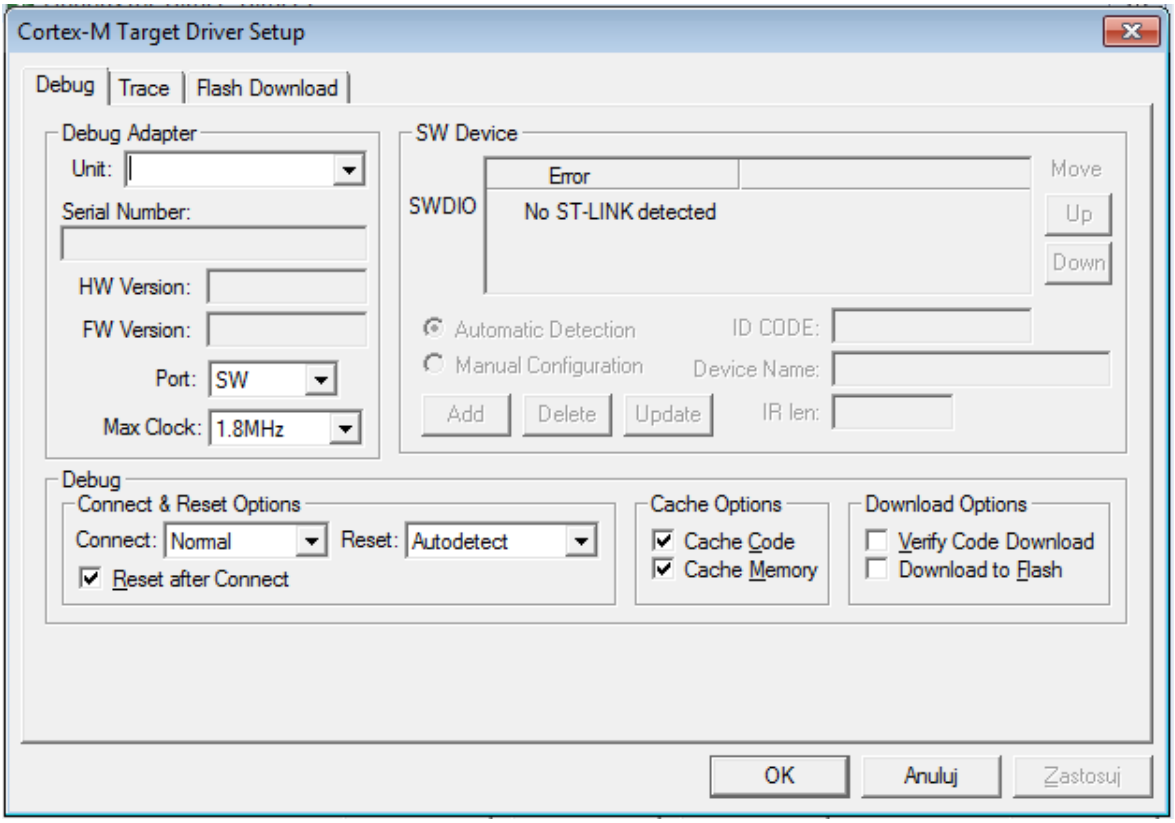
Rys. 6.10. Widok zakładki Linker w opcjach projektu



Rys. 6.11. Widok zakładki Debug w opcjach projektu

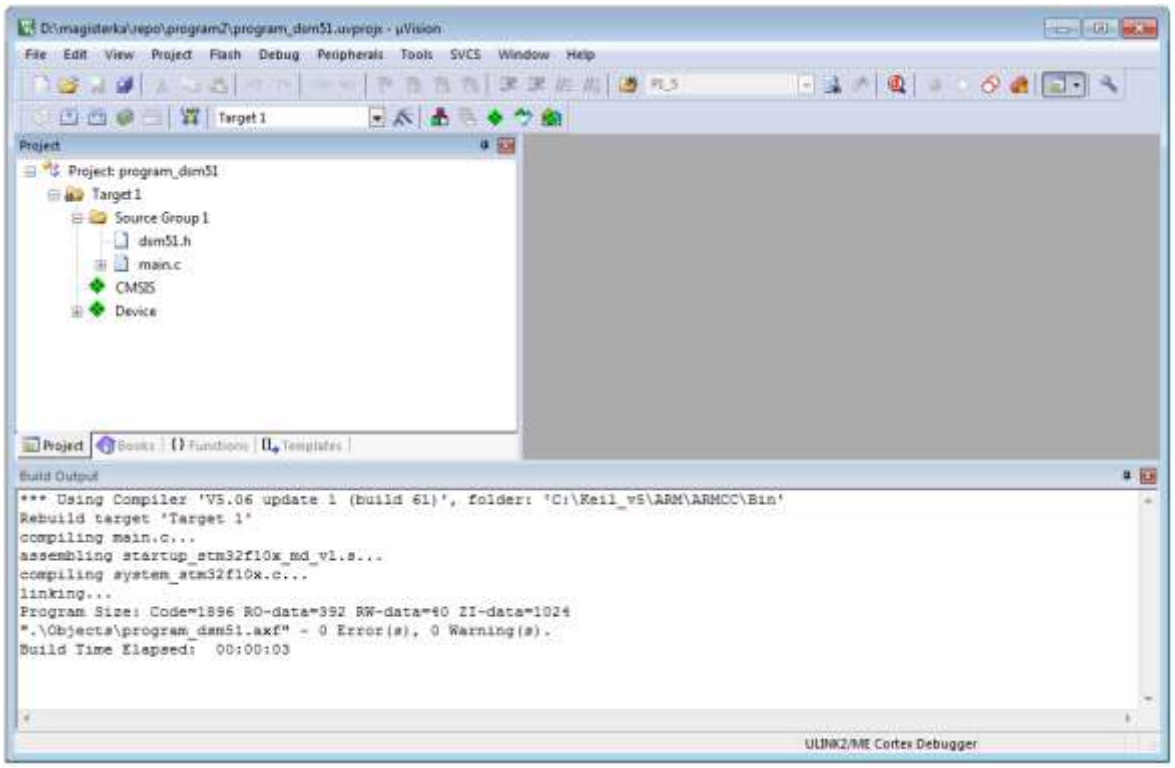


Rys. 6.12. Widok zakładki Utilites w opcjach projektu



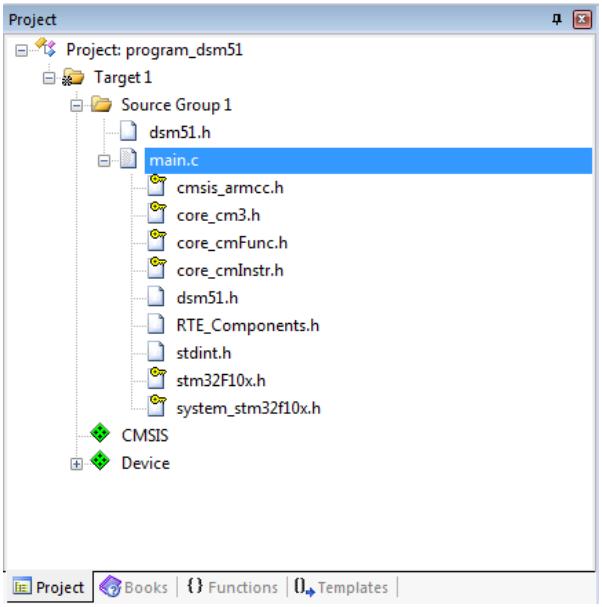
Rys. 6.13. Widok ustawień programatora po wyborze przycisku Settings w zakładce Utilites.

Po dokonaniu powyższych czynności można przystąpić do pierwszej kompilacji programu i wgrania do zestawu ewaluacyjnego. W celu skompilowania należy wybrać z menu Project opcję Build Target, bądź wcisnąć przycisk F7. Wynik poprawnej kompilacji widoczny jest na rys. 6.15.



Rys. 6.14. Wynik poprawnej kompilacji projektu

Po kompilacji zmienia się także zawartość drzewa projektu na przedstawiony na rys. 6.16.



Rys. 6.15. Widok drzewa projektu po prawidłowo zakończonym procesie kompilacji

Można zauważyć, że do projektu zostały dodane pliki pochodzące z bibliotek zainstalowanych na etapie konfiguracji środowiska Keil. Po poprawnej kompilacji należy rozpocząć proces wgrywania kodu maszynowego do mikrokontrolera poprzez wykorzystanie wbudowanego na płycie ewaluacyjnej programatora ST-Link. Jeżeli proces konfiguracji opcji Target został przeprowadzony prawidłowo, to cały proces wgrywania powinien zostać rozpoczęty poprzez naciśnięcie klawisza F8 lub wybranie z menu Flash opcji Download. Po naciśnięciu przycisku RESET na płycie ewaluacyjnej mikrokontroler powinien zacząć realizować program DEMO.

7. Opracowanie materiałów dydaktycznych

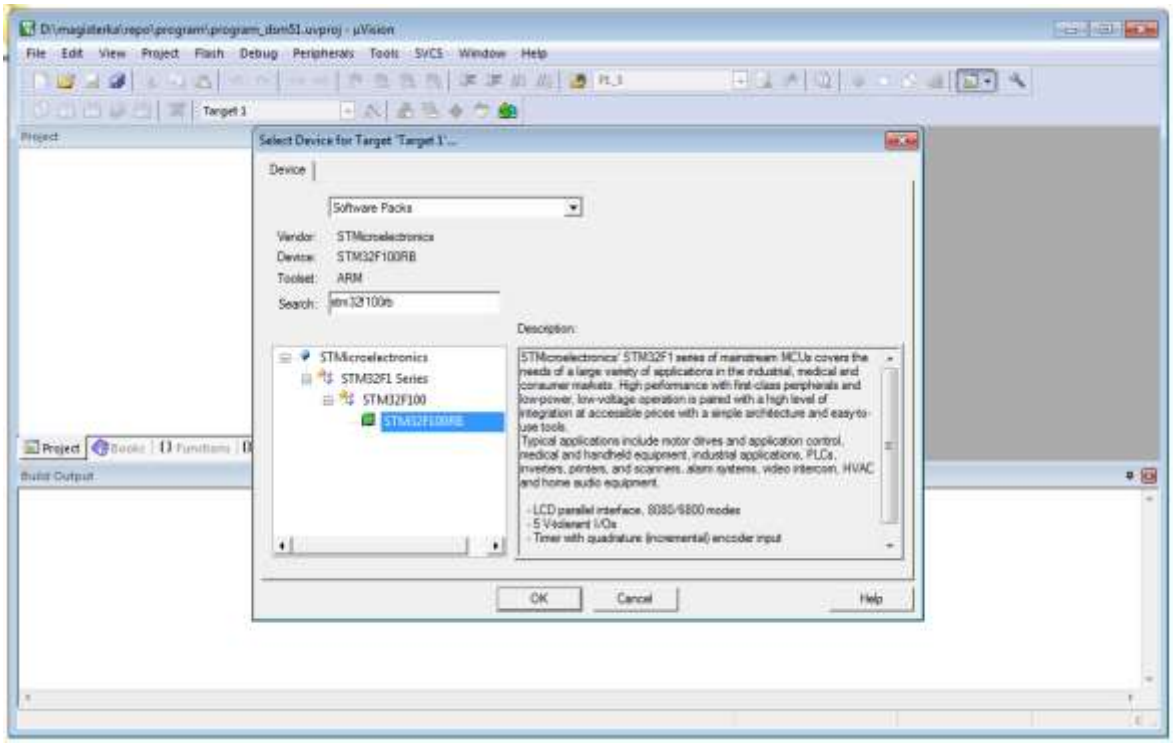
W ramach niniejszej pracy zostały opracowane przykładowe skrypty do wykorzystania podczas zajęć laboratoryjnych z podstaw techniki mikroprocesorowej wykorzystujące zmodyfikowany Dydaktyczny System Mikroprocesorowy DSM-51. W skład skryptów wchodzi część teoretyczna będąca fragmentami opisów zawartych w poprzednich rozdziałach oraz sam opis przebiegu ćwiczenia. Zostały opracowane następujące skrypty laboratoryjne:

1. Skrypt do zajęć wprowadzających: „Budowa oraz organizacja pamięci mikrokontrolera STM32F100RBT6B z rdzeniem ARM Cortex-M3".
2. Skrypt do zajęć wprowadzających – język asembler: „Lista instrukcji mikrokontrolera STM32F100RBT6B z rdzeniem ARM Cortex-M3".
3. Skrypt do zajęć wprowadzających – język C: „Programowanie mikrokontrolera STM32F100RBT6B z rdzeniem ARM Cortex-M3 z wykorzystaniem środowiska Keil uVision 5".
4. Skrypt do zajęć M.01: „Linie wejść/wyjść mikrokontrolera – asembler".
5. Skrypt do zajęć M.21: „Linie wejść/wyjść mikrokontrolera – język C".

W celu uniknięcia duplikowania treści, poniżej zostaną przedstawione jedynie części skryptów, które nie są częściami teoretycznymi pochodzącymi z poprzednich rozdziałów.

7.1. Skrypt do zajęć wprowadzających – język asembler

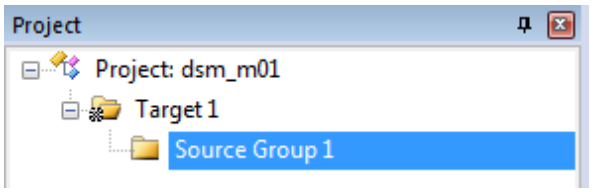
Na komputerze, na który będą przeprowadzane ćwiczenia laboratoryjne należy utworzyć katalog (np. w katalogu domowym użytkownika albo na Pulpicie systemu Windows), w którym później będą znajdowały się pliki źródłowe pisanych programów.



Rys. 7.1. Pierwszy krok tworzenia nowego projektu w Keil MDK-ARM

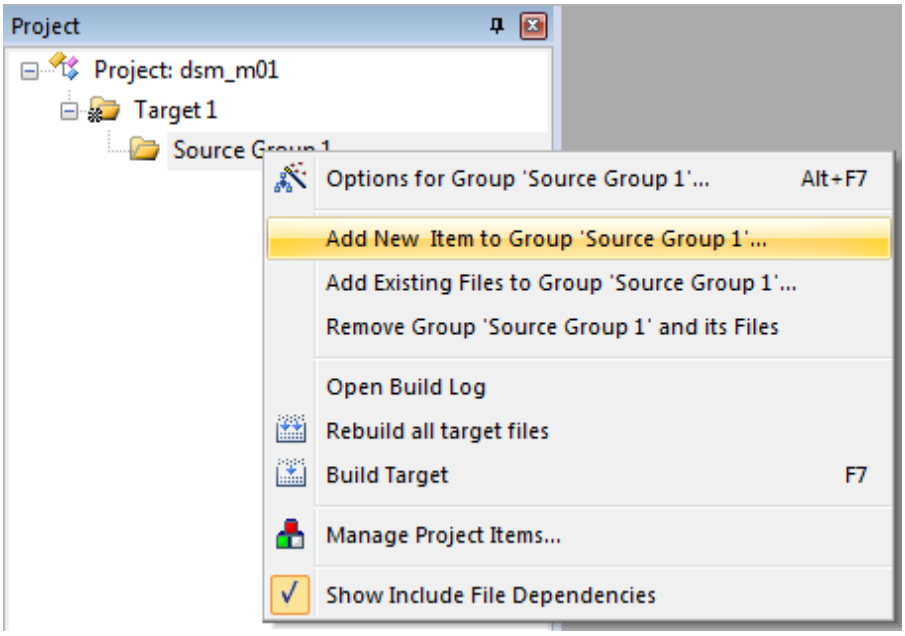
Do tego katalogu należy przekopiować pliki: `stm32f100rb_reg.inc` oraz `dsm_procedures.inc` z miejsca wskazanego przez prowadzącego zajęcia. Pierwszy

z nich zawiera przyporządkowanie adresów rejestrów mikrokontrolera ich nazwom, drugi – procedury pomocne w trakcie wykonywania ćwiczeń laboratoryjnych. Wykorzystanie tych plików będzie różne podczas różnych zajęć laboratoryjnych.
Po przygotowaniu katalogu należy uruchomić oprogramowanie Keil uVision5. Domyślnie otwiera się on na projekcie, który był ostatnio uruchamiany. W celu utworzenia nowego należy z menu Project wybrać opcję New µVision Project... . Zostanie wyświetlone okno, w którym należy wskazać wcześniej utworzony katalog oraz podać nazwę nowego projektu, na przykład `dsm_m01.uvprojx` przy czym rozszerzenie pliku może także zostać dodane automatycznie po naciśnięciu przycisku Zapisz. Nazwa nie powinna zawierać polskich znaków, spacji oraz znaków specjalnych. Po naciśnięciu przycisku Zapisz, zostanie przedstawione okno przedstawione na rys. 7.1., w którym to dokonuje się wyboru modelu mikrokontrolera, którego dotyczy projekt. W trakcie wyboru można wykorzystywać okienko Search. W systemie DSM-51 zastosowany jest mikrokontroler produkcji STMicroelectronics o nazwie STM32F100RB i taki też należy zaznaczyć w wyświetlonym drzewie to lewej stronie. Po naciśnięciu przycisku OK, środowisko prosi użytkownika o określenie, z jakich dostępnych sterowników i bibliotek będzie w danym projekcie korzystał.
Podczas ćwiczeń laboratoryjnych z języka asembler nie zostaną wykorzystana żadne biblioteki należy więc nacisnąć przycisk OK nie dokonując żadnych wyborów.



Rys. 7.2. Drzewo nowego projektu bez plików źródłowych

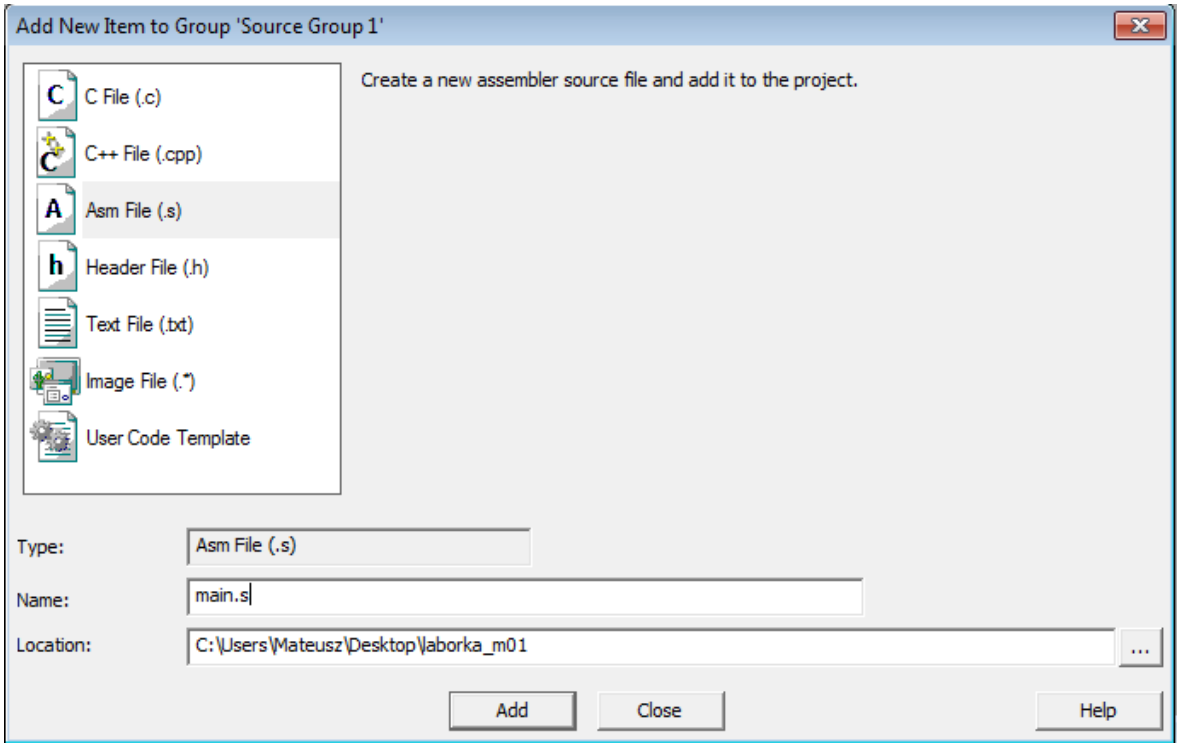
Po zatwierdzeniu zostanie ukazany okno programu Keil uVision z drzewem projektu po lewej stronie zawierającym jedynie katalog Target1 z podkatalogiem Source Group 1, przedstawione na rys. 7.2.



Rys. 7.3. Tworzenie nowego pliku źródłowego i dodawanie go do projektu

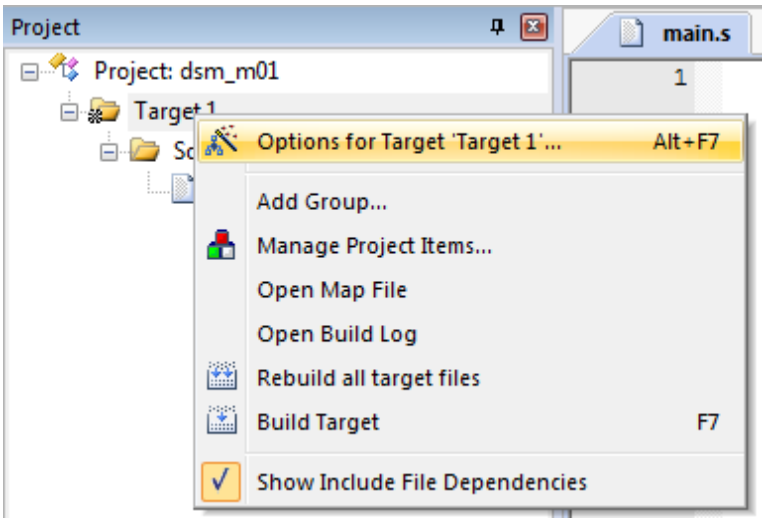
W celu utworzenia i dodania nowego pliku źródłowego do obecnego projektu należy kliknąć prawym przyciskiem myszy na katalog Source Group 1 w drzewie projektu

i w nim wybrać opcję Add New Item to Group ‘Source Group 1’… co zostało przedstawione na rys. 7.3. Zostanie wówczas wyświetlone okno na rys. 7.4., w którym należy wybrać typ pliku Asm File (.s) oraz nadać mu nazwę na przykład main.s. Nazwa nie powinna zawierać polskich znaków, spacji oraz znaków specjalnych.



Rys. 7.4. Parametryzowanie nowego pliku źródłowego

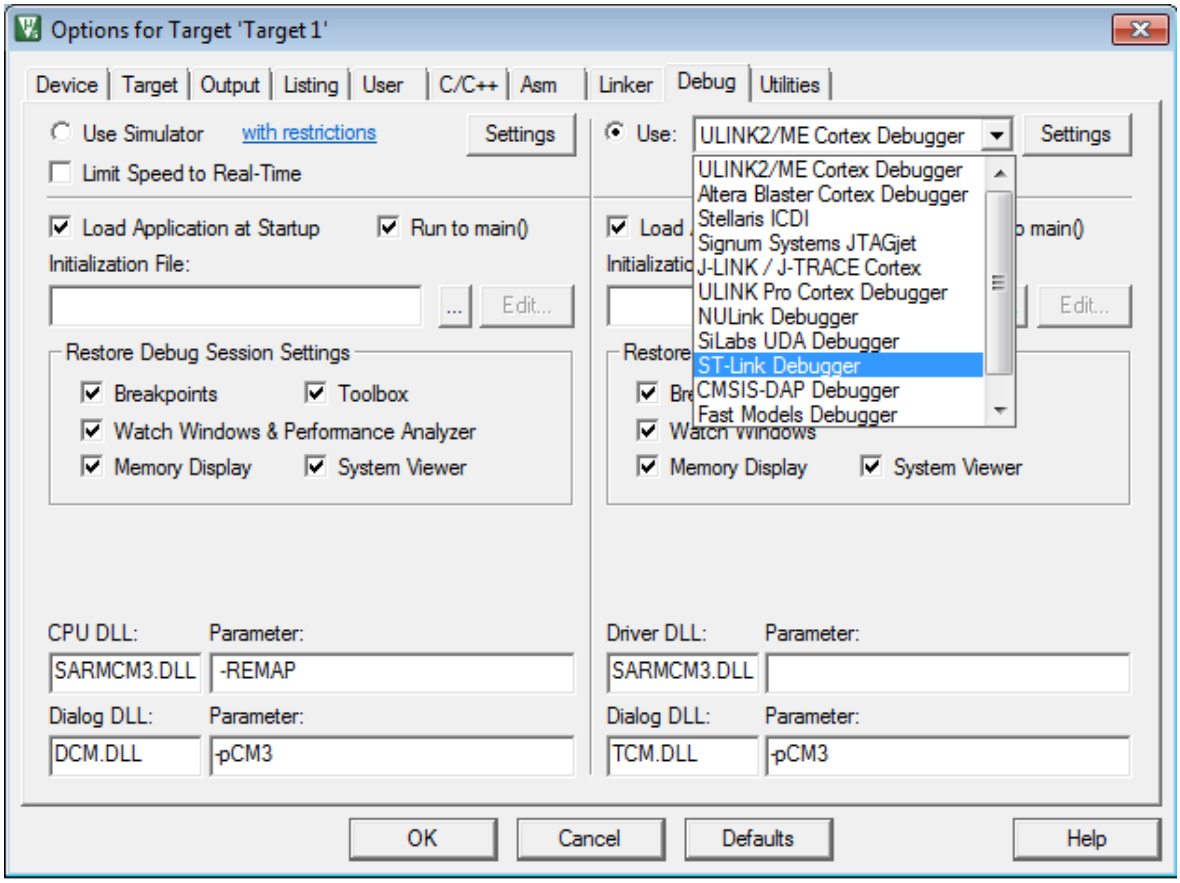
Po naciśnięciu przycisku Add drzewo projektu po lewej stronie okna będzie zawierało nowo utworzony plik main.s w podkatalogu Source Group 1, a część środkowa okna Keil uVison będzie zawierała otwarty edytor pliku main.s, w którym to będzie następowało pisanie programu. Przed tym jednak należy jeszcze skonfigurować programator w celu umożliwienia wgrywania i debugowania napisanych programów do systemu DSM-51.



Rys. 7.5. Wejście w tryb konfiguracji projektu

W tym celu należy prawym przyciskiem myszy kliknąć w katalog Target 1 w drzewie projektu po lewej stronie i z menu kontekstowego wybrać opcję Options for Target ‘Target 1’…, co zostało przedstawione na rys. 7.5. W menu konfiguracyjnym należy wybrać

zakładkę Debug i z listy dostępnych programatorów po prawej stronie wybrać ST-Link Debugger. Widok zakładki z wyborem programatora został przedstawiony na rys. 7.6.



Rys. 7.6. Wybór programatora ST-Link w opcjach konfiguracji projektu

Po wybraniu z listy programatora ST-Link Debugger jeszcze należy wejść w jego ustawienia klikając na przycisk Settings znajdujący się obok listy z programatorami. Tam w zakładce Flash Download należy zaznaczyć opcję Reset and Run. Następnie kliknąć OK i jeszcze raz OK w oknie z konfiguracją projektu. W tak skonfigurowanym środowisku można przystąpić do pisania, asemblacji i wgrywania programu do systemu DSM-51 oraz jego debugowania.

Najprostszy program napisany w języku asembler na mikrokontroler STM32F100RB został przedstawiony na listingu 7.1. W programie tym występują elementy, która będą się znajdować w każdym kolejnym programie pisanym na omawiany mikrokontroler. Program w języku asembler jest to jak widać pewnego rodzaju tekst zapisany według zasad obowiązujących dla danego mikrokontrolera oraz środowiska programistycznego – w tym wypadku ARM Keil uVision. Język asembler jest swego rodzaju odpowiednikiem programu zawartego w mikrokontrolerze ale w formie czytelnej dla człowieka. W celu przetłumaczenia wersji czytelnej dla człowieka na wersję zrozumiałą przez mikrokontroler korzysta się ze specjalnego programu zwanego również asemblerem, który dokonuje tak zwanej asemblacji programu. Wynikiem asemblacji jest właśnie kod wynikowy, który zostaje przesłany do pamięci wewnętrznej mikrokontrolera, tak aby mikrokontroler po sygnale reset mógł zacząć realizować zawarty w tej pamięci kod. Język asembler to przede wszystkim rozkazy (inaczej instrukcje) podawane w formie tekstu, które to procesor będzie wykonywał oraz zapisane wartości stałe w pamięci mikrokontrolera. Różne procesory obsługują różne instrukcje. Lista rozkazów omawianego mikrokontrolera znajduje się w rozdziale 5 niniejszej pracy magisterskiej.

Listing 7.1. Najprostszy program na mikrokontroler STM32F100RB

```
; DYREKTYWY ASEMBLERA
      PRESERVE8
      THUMB                              ; lista instrukcji Thumb-2

; TABLICA WEKTOROW PRZERWAN
      AREA    RESET, DATA, READONLY   ; przestrzen DATA czyli danych
      EXPORT  __Vectors               ; w pamieci programu

__Vectors
      DCD  0x20001000                 ; adres szczytu stosu do SP
      DCD  Reset_Handler              ; adres poczatku programu

      ALIGN                           ; dopełnienie do 4 bajtow

; PROGRAM
      AREA    MYCODE, CODE, READONLY   ; przestrzen CODE czyli rozkazow
                                       ; w pamięci programu
      ENTRY                            ; dyrektywa poczatku programu
      EXPORT Reset_Handler             ; dla linkera

Reset_Handler                          ; etykieta dla linkera

; PROGRAM UZYTKOWNIKA
      B Reset_Handler                  ; petla glowna programu

      END   ; KONIEC PROGRAMU
```

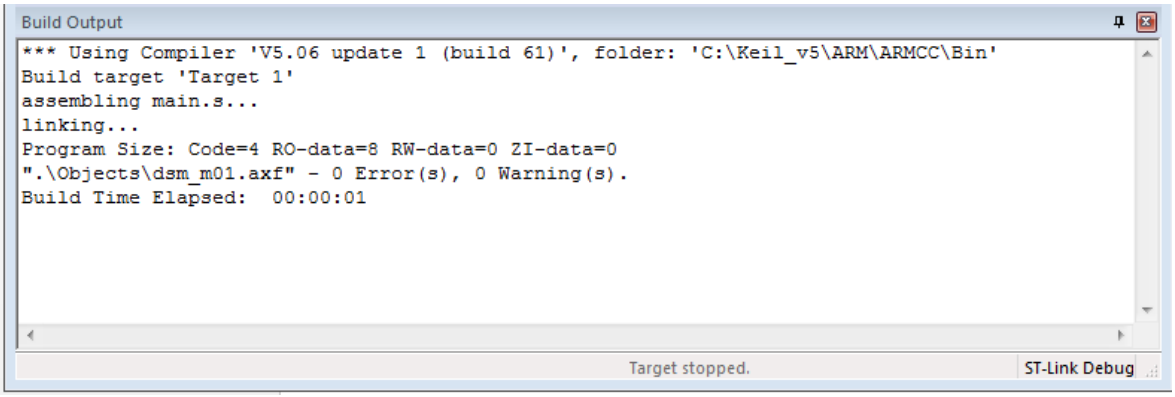
W programie zapisanym w języku asembler oprócz samych rozkazów i stałych, znajdują się także inne elementy służące na przykład to wzbogaceniu czytelności kodu czy informujące program asembler o dodatkowych ustawieniach. Jednym z takich elementów są tak zwane komentarze. Są to teksty rozpoczynające się w kodzie od symbolu średnika „;”, które zawierają informacje napisane przez programistę w celu ułatwienia późniejszej modyfikacji czy też debugowania kodu, a nie są one uwzględniane przez kompilator. Mogą więc zawierać dowolną treść na przykład „Ponizsza czesc programu obsluguje klawiaturę”. Komentarze nie mogą jednak zawierać polskich znaków. Oprócz komentarzy znajdują się między innymi dyrektywy asemblera. W omawianym przykładzie podane są dwie dyrektywy: PRESERVE8 oraz THUMB. Pierwsza z nich informuje linker o zachowaniu w projekcie 8-bajtowego wyrównania na stosie, druga informuje program asembler o rodzaju listy instrukcji procesora. Omawiany mikrokontroler obsługuje tylko instrukcje z listy Thumb-2. Dalej, kod zawarty w programie podzielony jest na pewnego rodzaju fragmenty dyrektywami AREA. Podział wynika z tego, że inną magistralą rdzeń Cortex-M3 odczytuje stałe zapisane w pamięci programu (magistralą DCode bus), a inną rozkazy do wykonania (magistralą ICode bus). W związku z tym napisany kod dzielony jest na obszary DATA (tu umieszczane są stałe) i CODE (tu umieszczane są instrukcje). Po słowie AREA następuje podanie jej nazwy, która nie ma większego znaczenia, następnie po przecinku wskazanie obszaru – obszar danych (DATA) albo instrukcji (CODE) oraz flag informujących na przykład czy jest to przestrzeń tylko do odczytu (READONLY) albo do odczytu i zapisu (READWRITE) oraz inne tutaj nieużywane. Na listingu 7.1. pierwsza dyrektywa AREA definiuje przestrzeń danych tylko do odczytu o nazwie RESET. W niej zawarta będzie tablica wektorów przerwań. Linker oprogramowania Keil wymaga, żeby tablica wektorów przerwań oznaczona była etykietą __Vectors. Etykieta to swego rodzaju nazwa pojawiająca się przed danym fragmentem programu, którą można wykorzystać w dalszej części programu. Można zauważyć, że pomijając komentarze, tylko etykiety zaczynają się od początku linii w kodzie programu przedstawionym na listingu 7.1.

Pozostałe elementy (dyrektywy, instrukcje, itp.) podane są po odsunięciu symbolem tabulacji. Taka jest obowiązująca zasada pisania programów w języku asembler w środowisku Keil. Wyrażenie EXPORT __Vectors powoduje udostępnienie etykiety __Vectors na zewnątrz pliku. Dzięki czemu można ją wykorzystać podając w innym pliku albo może ją też wykorzystać linker. Instrukcje DCD pod etykietą __Vectors powodują zapis podanych wartości stałych w pamięci programu. Linker znajdując etykietę __Vectors umieszcza znajdujące się tam dane w pamięci Flash zaczynając od adresu 0x0800 0000. Mikrokontroler po otrzymaniu sygnału reset przy takiej konfiguracji linii BOOT0 i BOOT1 jak jest w systemie DSM-51 odczytuje pierwszą 32-bitową wartość spod adresu 0x0800 0000 i wpisuje do rejestru procesora SP traktując jako wskaźnik szczytu stosu. W podanym programie będzie to adres 0x2000 1000. Następnie odczytuje daną spod adresu 0x0800 0004 i traktuje podaną tam wartości jako adres od którego ma nastąpić rozpoczęcie wykonywania programu po otrzymaniu sygnału reset (Reset_Handler). W podanym programie nie jest on jednak podany wprost. Występuje tam po słowie DCD nie wartość, lecz etykieta Reset_Handler. Jest to znów wymóg linkera środowiska Keil. Asembler wykonuje się podczas kompilacji dwa razy. Za pierwszym razem asembluje kod i po asemblacji dopiero stwierdza pod jaki adres trafiły instrukcje, nad którymi znajduje się etykieta Reset_Handler. Wówczas wykonuje się drugi raz, w celu wpisania pod etykietę Reset_Handler w instrukcji DCD wyznaczonego wcześniej adresu. W ten sposób, po zaprogramowaniu mikrokontrolera pod adresem 0x0800 0004 znajdzie się już adres pierwszej instrukcji oznaczonej etykietą Reset_Handler. Następujące po wyrażeniu DCD Reset_Handler słowo ALIGN zapewnia wyrównania w kodzie programu, żeby kolejne instrukcje spod etykiety Reset_Handler na pewno tam występowały. Kolejną przestrzenią jest MYCODE również utworzoną dyrektywą AREA. Nazwa MYCODE nie ma tu znaczenia. Ta przestrzeń jest już typu CODE, to znaczy, że w niej znajdować się będą instrukcje procesora. Podane poniżej tej deklaracji słowo ENTRY oznacza miejsce od którego zaczyna się program. Każdy napisany program musi mieć takie miejsce. Następnie następuje wyeksportowanie etykiety Reset_Handler (wymóg linkera Keil), dzięki czemu linker będzie mógł odnaleźć miejsce w kodzie programu, od którego następuje rozpoczęcie pracy. Pod etykietą Reset_Handler zaczyna się dopiero kod użytkownika, który będzie wykonywany. W omawianym programie kod użytkownika składa się z jednej instrukcji B Reset_Handler co oznacza „skocz do etykiety Reset_Handler”. Inaczej mówią program w tym miejscu będzie wykonywał nieskończoną pętlę, gdyż pod etykietą Reset_Handler znajduje się instrukcja skoku do etykiety Reset_Handler więc procesor cały czas będzie wykonywał tą jedną instrukcję. Każdy program mikrokontrolera musi posiadać nieskończoną pętlę pisany czy to w języku C czy asembler. Ostanim wyrażeniem występującym w programie jest słowo END kończące program. Mikrokontroler nigdy nie osiągnie tego miejsca ale środowisko Keil wymaga aby takie słowo występowało na końcu każdego pliku.

Każdy z omawiany na kolejnych zajęciach programów o ile nie wskazano inaczej będzie miał właśnie taką strukturę pliku z programem. W celu oszczędności miejsca często na listingach będzie prezentowana jedynie część programu użytkownika, która znajduje się pomiędzy etykietą Reset_Handler, a dyrektywą END. Należy jednak pamiętać, że w celu poprawnej kompilacji należy wpisać jeszcze całą część pliku występującą nad etykietą Reset_Handler.

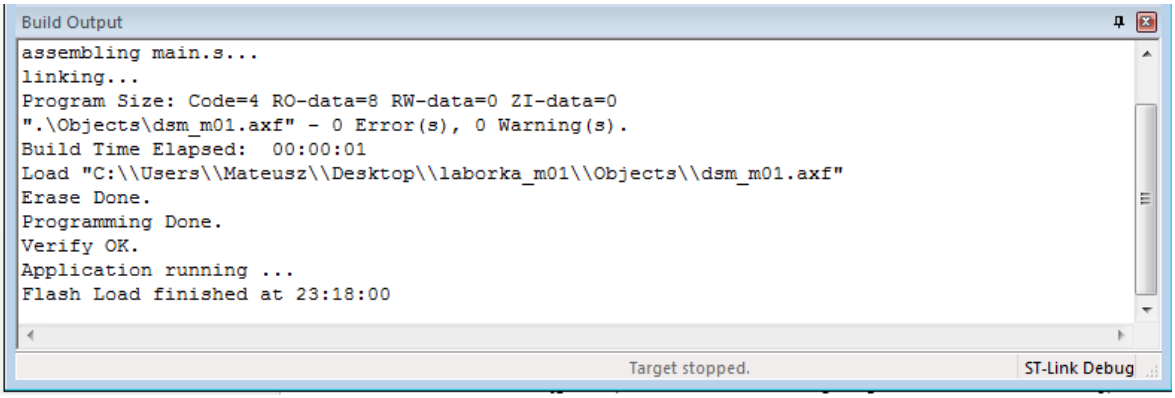
Po zapoznaniu się ze strukturą pliku można przystąpić do pierwszego wygenerowania kodu maszynowego zrozumiałego dla procesora z omawianego programu i zaprogramowania mikrokontrolera. W tym celu należy przepisać/przekopiować program zawarty w listingu 7.1. do otwartego w edytorze Keil wcześniej utworzonego pliku ze źródłem programu (w przykładzie jest to plik main.s). Po przekopiowaniu należy zapisać

zawartość pliku (np. CTRL+s) a następnie dokonać zbudowania kodu wynikowego. Budowania dokonuje się przyciskiem F7. Po poprawnie przeprowadzonym procesie w oknie komunikatów znajdującym się u dołu programu uVision powinien znajdować się komunikat podobny do tego z rys. 7.7.



Rys. 7.7. Okno komunikatów środowiska Keil uVision po poprawnie przeprowadzonym generowaniu kodu wynikowego

Następnie (o ile nie zostało to przeprowadzone wcześniej) należy podłączyć system DSM-51 kablem USB do komputera, na którym uruchomione jest środowisku Keil uVision i będąc w programie uVision nacisnąć przycisk F8 w celu zaprogramowania mikrokontrolera. Okno komunikatów po poprawnym zaprogramowaniu zostało przedstawione na rys. 7.8.



Rys. 7.8. Okno komunikatów środowiska Keil uVision po poprawnym zaprogramowaniu mikrokontrolera

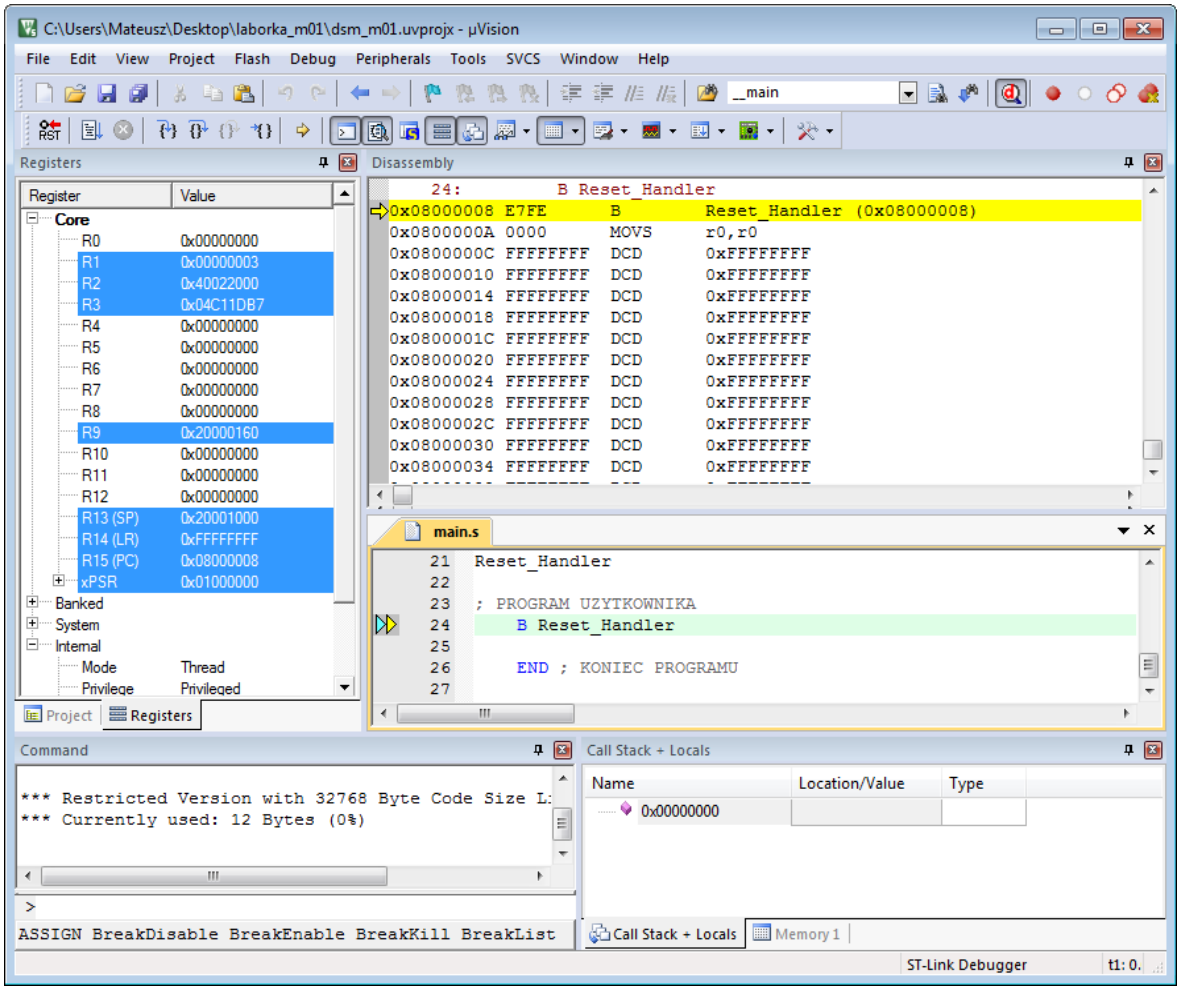
Przykładowy najprostszy program nie robi nic po za ciągłym skokiem do rozkazu właśnie z tym skokiem w związku z czym w systemie DSM-51 nie widać oznak pracy mikrokontrolera. Używanie diody LED TEST, buczka BUZZER, klawiatury czy wyświetlaczy będzie przeprowadzane podczas kolejny zajęć laboratoryjnych.

7.1.1. Debugowanie działania programu

Program napisany, poprawnie skompilowany i wgrany do mikrokontrolera może czasami nie działać tak jak zaplanował to programista. Problemem najczęściej leży w nieprawidłowym przetworzeniu opracowanej koncepcji działania programu na polecenia języka asembler. Po stwierdzeniu nieprawidłowego działania programu rozpoczyna się proces analizy takiego stanu, najczęściej analizując krok po kroku jak działa mikrokontroler. Nieocenionym wręcz narzędziem jest w takim przypadku wspomniany debuger. Mikrokontrolery z rdzeniami Cortex posiadają w swych strukturach elementy umożliwiające wgląd do pracującego mikrokontrolera. Umożliwiają zatrzymanie wykonywanego programu w dowolnej chwili czy analizowanie działania jego pracy krok

po kroku, czyli pauzując wykonywanie programu po każdym rozkazie. Taki proces nazwany jest debugowaniem, a oprogramowanie umożliwiające ten proces – debugerem. Środowisko Keil uVison zawiera wbudowany debuger umożliwiający opisane wyżej czynności.

Po poprawnym zbudowaniu i wgraniu programu z poprzedniego rozdziału do pamięci mikrokontrolera należy użyć debugera w celu podglądu działania programu. W tym celu mając cały czas przyłączony poprzez USB zestaw DSM-51 należy w programie uVision wybrać z menu Debug opcję Start/Stop Debug Sesion lub wcisnąć kombinację przycisków CTRL+F5. Zostanie wówczas wyświetlony komunikat, że w wersji ewaluacyjnej oprogramowania występuje ograniczenie ilości kodu do 32 kB, które należy potwierdzić przyciskiem OK. Po zatwierdzeniu wygląd środowiska Keil uVision ulegnie pewnej zmianie na taki przedstawiony na rys. 7.9.



Rys. 7.9. Okno debugera środowiska Keil uVision

Lewa strona zawiera aktualne wartości rejestrów procesora – rejestrów ogólnego przeznaczenia R0…R12, SP, LR, PC oraz PSR. Ukazane są również rejestry specjalne oraz tryby pracy rdzenia, a także numer kroku i czas procesora od otrzymania sygnału reset. Można zauważyć, iż pomimo nieużywania w programie rejestrów ogólnego przeznaczenia R0…R12 mogą się w nich znajdować niezerowe wartości. Wartości po resecie znajdujące się w tych rejestrach są bowiem niezdefiniowane. Prawa część okna podzielona jest na dwa obszary. W dolnym widoczny jest kod programu napisany przez użytkownika, górna część pokazuje zawartość pamięci mikrokontrolera i rozkazy w niej zawarte. W jednym i drugim oknie aktualnie wykonywana instrukcja jest zaznaczona strzałką z lewej strony. Po wejściu w tryb debugowania, praca mikrokontrolera jest wstrzymywana. Sterowanie pracą obywa

się przyciskami znajdującymi się na belce nad rejestrami. Belka ta przedstawiona jest na rys. 7.10.

Rys. 7.10. Belka przycisków sterujących wykonywaniem programu w trybie debugowania

Patrząc od lewej – pierwszy przycisk służy do resetowania mikrokontrolera, drugi do włączenia pracy ciągłej (tak jakby nie było przyłączonego debugera), kolejny do zatrzymania pracy ciągłej. Następny przycisk lub też klawisz F11, to wykonanie kolejnego tylko jednego rozkazu zapisanego w pamięci programu. Naciskając więc przycisk F11 można uzyskać podgląd na pracę mikrokontrolera krok po kroku w tempie na przykład 1 rozkaz na sekundę. Kolejny przycisk powoduje ominięcie kolejnej tylko jednej linii z rozkazem – czyli jej niewykonanie. Jeszcze następny wyjście z funkcji jeżeli się w niej znajdujemy. Ostatni natomiast umożliwia wykonanie fragmentu kodu do miejsca w którym obecnie znajduje się kursor.

Oprócz powyższych, debuger umożliwia wprowadzenie również tak zwanych punktów przerwań programu (ang. *Break Point*), po których osiągnieciu program zostaje wstrzymany oraz wiele, wiele innych w tym podgląd rejestrów konfiguracyjnych urządzeń specjalnych czy podgląd wartości w pamięci SRAM. W zależności od potrzeb należy korzystać z wybranych funkcjonalności.

Wyjście z trybu debugowania odbywa się w ten sam sposób co wejście, to znaczy poprzez naciśnięcie przycisków CTRL+F5 lub wybrania z menu Debug opcji Start/Stop Debug Sesion.

7.1.2. Zadania do samodzielnego wykonania

Korzystając z szablonu programu przedstawionego na listingu 7.1. należy dokonać modyfikacji znajdującego się tam programu użytkownika, na taki znajdujący się na listingu 7.2, a następnie zbudować i wgrać tak przygotowany program do mikrokontrolera.

Listing 7.2. Program wykorzystujący polecenia MOV, ADD, MUL

```
Reset_Handler
; PROGRAM UZYTKOWNIKA
      MOV R0, #0        ; wpisanie wartości 0 do rejestru R0
      MOV R1, #0        ; wpisanie wartości 0 do rejestru R1
      MOV R2, #0        ; wpisanie wartości 0 do rejestru R2
      MOV R3, #0        ; wpisanie wartości 0 do rejestru R3
      MOV R4, #0        ; wpisanie wartości 0 do rejestru R4

      MOV R0, #20       ; wpisanie wartości 20 do rejestru R0
      MOV R1, #0x20     ; wpisanie wartosci 0x20 do rejestru R0
      ADD R2, R1, R0    ; rownanie R2 = R1 + R0
      MOV R3, R2        ; wpisanie wartosci rejestru R2 do R3
      ADD R3, R3, R3    ; rownanie R3(new) = R3(old) + R3(old)
      MUL R4, R3, R1    ; rownanie R4 = R3 x R1
      B .

      END   ; KONIEC PROGRAMU
```

Następnie korzystając z debugera przeanalizować zawartości rejestrów ogólnego przeznaczenia R0…R4 w pracy krokowej po wykonaniu każdego rozkazu. W oknie debugera należy również przeanalizować ile cykli zegara i jaki czas potrzebny jest na wykonanie instrukcji: MOV, ADD oraz MUL. W listingu 7.2. ostatnim rozkazem przez dyrektywą END jest rozkaz „B .” . Zapis ten oznacza wykonywanie skoku do miejsca, w którym się znajduje ten rozkaz, czyli inaczej mówiąc jest to wspomniana wcześniej pętla nieskończona.

Kolejny program do przeanalizowania z użyciem debugera został przedstawiony na listingu 7.3.

Listing 7.3. Program wykorzystujący pętle

```
Reset_Handler
; PROGRAM UZYTKOWNIKA
      MOV R0, #6        ; wpisanie wartości 6 do rejestru R0
      MOV R1, #1        ; wpisanie wartości 1 do rejestru R1
      MOV R2, #0        ; wpisanie wartości 0 do rejestru R2

Skocz_tu                ; etykieta dla poniższego fragmentu programu
      ADD R2, R1        ; dodaj do rejestru R2 wartosc z R1 i zapisz w R2
      SUBS R0, #1       ; odejmij od R0 jeden i zapisz w R0
      BNE   Skocz_tu    ; jeżeli R0 > 0 to skocz do etykiety Skocz_tu

      B .               ; petla glowna programu

      END   ; KONIEC PROGRAMU
```

W programie tym wykorzystana została pętla, która wykona się tyle razy jaka została wpisana liczba do rejestru R0 przy starcie programu. Należy następnie zmodyfikować wartość początkową rejestru R1 i przeanalizować efekty. Używając debugera należy zwrócić uwagę na stan poszczególnych flag w rejestrze xPSR w trakcie wykonywania kolejnych kroków programu oraz zaobserwować ich wpływ na wykonanie instrukcji skoku warunkowego BNE Skocz_tu.

Zadania do wykonania

1. Na podstawie podanych przykładów napisać i przeanalizować debugerem program wyliczający silnię liczby podanej w rejestrze R0 i zapisujący wynik do rejestru R1.
2. Na podstawie podanych przykładów napisać i przeanalizować debugerem program wyliczający kwadrat sumy liczb zawartych w rejestrach R0 i R1 i zapisujący wynik do rejestru R3.
3. Na podstawie podanych przykładów napisać możliwie najprostszy program wyliczający podaną zależność: `R5 = R4 - (R2 * 8)`

7.2. Linie wejść/wyjść mikrokontrolera – asembler

Porty GPIO są urządzeniami peryferyjnymi z punktu widzenia procesora takimi samymi jak każde inne urządzenie peryferyjne w mikrokontrolerze, to znaczy posiada pewne rejestry, które są widoczne pod konkretnymi adresami z przestrzeni adresowej urządzeń peryferyjnych. Procesor zawarty w omawianym mikrokontrolerze umożliwia wykonywanie operacji jedynie na rejestrach ogólnego przeznaczenia, ewentualnie rejestrach specjalnych procesora. Natomiast w przypadku zamiaru przeprowadzania operacji na danych znajdujących się poza rejestrami ogólnego przeznaczenia (np. danych z pamięci SRAM, czy urządzeń peryferyjnych) należy korzystać ze specjalnych instrukcji LDR i STR służących do przesyłania danych z pamięci do rejestrów ogólnego przeznaczenia i w drugą stronę. Rozkazy te omówione zostały w skrypcie do zajęć wprowadzających pt.: „Lista instrukcji mikrokontrolera STM32F100RBT6B z rdzeniem ARM Cortex-M3”.

Najprostszy program, którego funkcjonalność sprowadza się do zaświecenia diody LED TEST w systemie DSM-51 został przedstawiony na listingu 7.4. Jak należy rozumieć przedstawiony na nim zapis. Przede wszystkim należy posiadać wiedzę, do którego wyprowadzenia mikrokontrolera przyłączona jest dioda LED TEST.

Listing 7.4. Program zaświecający diodę LED TEST

```
Reset_Handler
; PROGRAM UZYTKOWNIKA
      LDR R0, = 0x40021018    ; rejestr RCC_APB2ENR
      LDR R1, = 0x00000008    ; flaga zalaczenie zegara do portu B
      STR R1, [R0]            ; wpisanie wartosci z R1 do adresu z R0
      LDR R0, = 0x40010C04    ; rejestr GPIOB_CRH
      LDR R1, = 0x70000000    ; bity ustawiające linie 15 w otwarty dren
      STR R1, [R0]            ; wpisanie wartosci z R1 do adresu z R0
      LDR R0, = 0x40010C14    ; rejestr GPIOB_BRR
      LDR R1, = 0x8000        ; bit ustawiajacy linie nr 15 w stan niski
      STR R1, [R0]            ; wpisanie wartosci z R1 do adresu z R0
      B .                     ; glowna petla programu

      END   ; KONIEC PROGRAMU
```

Jest nim wyprowadzenie nr 15 portu B, inaczej mówiąc B15. Oznacza to, że należy zająć się rejestrami konfiguracyjnymi portu GPIOB. Na samym jednak początku należy zwrócić uwagę na fakt, że domyślnie sygnał zegara do wszystkich urządzeń peryferyjnych jest wyłączony, należy najpierw załączyć sygnał zegara dla wybranych urządzeń peryferyjnych, a dopiero w kolejnym kroku przystąpić do konfiguracji jego rejestrów. Jak wiadomo ze skryptu do zajęć wprowadzających, załączania i wyłączania sygnału zegarowego dokonuje się poprzez układ kontroli sygnałów zegarowych i resetu RCC. Załączenie sygnału zegarowego do portu GPIOB odbywa się poprzez ustawienie flagi IOPBEN w rejestrze RCC_APB2ENR. Należy więc po pierwsze, obliczyć adres rejestru RCC_APB2ENR wykorzystując informację o adresie pierwszego rejestru układu RCC oraz offsecie dla rejestru RCC_APB2ENR, a następnie pod ten adres wpisać taką wartość, żeby odpowiadała ustawionej fladze IOPBEN. Korzystając z poprzednich skryptów wiadomo, że pierwszy rejestr urządzenia RCC widoczny jest pod adresem 0x4002 1000 natomiast rejestr RCC_APB2ENR posiada offset 0x18 co po zsumowaniu daje adres 0x4002 1018. Jest to adres rejestru RCC_APB2ENR. Następnie w rejestrze tym należy odnaleźć flagę IOPBEN. Flaga znajduje się na trzecim bicie (licząc od 0) w tym rejestrze. Binarnie więc można byłoby ten rejestr z ustawioną tylko flagą IOPBEN zapisać w postaci:

```
      0000 0000 0000 0000 0000 0000 0000 1000
```

Posługiwanie się jednak tego typu ciągami jest bardzo niewygodne, dlatego przetwarza się takie wartości na postać heksadecymalną (szesnastkową) i podobnie jak adres zapisuje w postaci: 0x….. . Dla podanego przypadku będzie to 0x08.
Zatem w celu załączenia sygnału zegarowego do portu GPIOB należy pod adres 0x4002 1018 wpisać wartość 0x08. Można w tym celu użyć rozkazu: STR Rd, [Ra], gdzie Rd – to rejestr ogólnego przeznaczenia, zawierający daną do zapisu, a Ra – rejestr ogólnego przeznaczenia zawierający adres, pod który należy zapisać daną z rejestru Rd. Należy jednak wcześniej do tych rejestrów niejako „włożyć" obliczone wcześniej wartości, to znaczy do rejestru przechowującego adres liczbę 0x4002 1018, a do adresu przechowującego daną liczbę 0x08. Z tym drugim przypadkiem, nie ma dużego problemu, gdyż można użyć tu rozkazu MOV, natomiast użycie tego rozkazu w przypadku liczby z adresem spowoduje błąd asemblera, gdyż rozkaz MOV nie jest w stanie przekazać tak dużej wartości stałej do rejestru. Wygodnie jest w tym przypadku użyć adresowania w postaci pseudoinstrukcji. W takim przypadku zapisuje się rozkaz LDR, R =wartość. Wówczas asembler sam dobierze najodpowiedniejszy sposób przekazania wartości do rejestru R. Dla prostych przypadków będzie to np. instrukcja MOV, dla większych liczb np. adresowanie pośrednie z wykorzystaniem etykiety. W omawianym przykładzie jako

rejestr przechowujący adres został wybrany rejestr R0, a rejestr przechowujący daną R1. Można zauważyć, że dwa pierwsze rozkazy w omawianym programie to wpisanie do rejestrów R0 i R1 wyliczonych wcześniej wartości. Następnie wykonywania jest instrukcja STR, która wpisuje wartość z rejestru R1 pod adres zawarty w R0. Po wykonaniu tej instrukcji uruchomione zostało doprowadzanie sygnału zegarowego do port GPIOB.

Kolejnym krokiem jest ustalenie trybu pracy linii nr 15 portu B. Linia ma służyć do zaświecania lub nie diody LED TEST. Musi zatem to być linia skonfigurowana w tryb wyjścia. Trybów wyjściowych są dwa: albo otwarty dren, albo wyjście typu push-pull. W przypadku systemu DSM-51 ze względu na to, iż elementy w oryginalnej części systemu pracują w standardzie TTL, a mikrokontroler zasilany jest z napięcia 3,3 V zostały zastosowane zewnętrzne rezystory podciągające na wszystkich liniach mikrokontrolera do napięcia +5 V względem masy, które wymuszają stan wysoki. Oznacza to, że mikrokontroler powinien jedynie zwierać wyjście do masy w przypadku wystawiania zera logicznego i pozostawiać linię wolną w przypadku wystawiania jedynki logicznej. Taką funkcjonalność ma wyjście typu otwarty dren. Ostatnią czynnością jest wybór maksymalnej częstotliwości pracy. Niech w omawianym przypadku będzie to 50 MHz. Teraz kierując się tab. 5.10. należy wyznaczyć dla określonych wymagań stan flag CNF i MODE. Dla tych wymagań powinny być to ustawienia CNF1 = 0, CNF0 = 1, MODE1 = 1, MODE0 = 1. Należy teraz sprawdzić, w którym rejestrze znajdują się ustawienia linii nr 15 portu B. Jest to rejestr GPIOB_CRH widoczny pod adresem 0x4001 0C04. Adres został wyliczony analogicznie do RCC_APB2ENR. Bity CNF i MODE odpowiedzialne za linię nr 15 stanowią 4 najbardziej znaczące bity omawianym rejestrze. Przeprowadzając analogiczną operację zamiany wartości bitowej na szesnastkową można wyznaczyć, że wymagane ustawienie bitów odpowiada wartości 0x7000 0000. Inaczej mówiąc, pod adres 0x4001 0C04 należy wpisać wartość 0x7000 0000. Trzy kolejne linie w programie, po ustawieniu rejestru RCC_APB2ENR temu właśnie służą.

Ostatnią czynnością jest wymuszenie stanu wysokiego lub niskiego na skonfigurowanej linii. Z konstrukcji systemu DSM-51 wynika, że dioda LED TEST zaświeci się, gdy na linii nr 15 portu B będzie panował stan niski. Można to uzyskać wpisując w odpowiedni bit rejestru GPIOB_BRR wartość 1. Służą temu 3 kolejne linie przedstawionego kodu.

Program przedstawiony na listingu 7.4. należy zbudować i wgrać do systemu DSM-51, a następnie zaobserwować zaświecenie się diody LED TEST. Patrząc się na podany program, można odnieść wrażenie, że jest on mało zrozumiały i czytelny. Bez komentarzy wpisanych przez programistę, po upływie nawet kilku dni trudno jest od razu powiedzieć jaki cel ma wpisywanie takich a nie innych wartości pod wskazane adresy. Sytuację można poprawić tworząc pewnego rodzaju wyrażenia czytelne dla człowieka, które dla asemblera widoczne są jako liczby. Należy w tym celu użyć dyrektyw EQU. Dyrektywa taka tworzy odwzorowanie wartości liczbowej określonemu wyrażeniu. Można ją wykorzystać w przypadku korzystania ze stałych matematycznych, fizycznych (np. liczba π) albo omawianym przypadku do przechowywania adresów rejestrów. Wyrażenie z dyrektywą EQU ma następującą postać:

```
RCC_APB2ENR       EQU   0x40021018
```

Takie wyrażenie powinno znaleźć się na samym początku pliku z programem asembler. Następnie w kodzie programu można wykorzystywać nazwę RCC_APB2ENR, która odwzorowuje adres tego rejestru. Stąd w programie zamiast zapisu:

```
LDR R0, = 0x40021018
LDR R1, = 0x00000008
STR R1, [R0]
```

można korzystać z następującego:

```
      LDR R0, = RCC_APB2ENR
      LDR R1, = 0x00000008
      STR R1, [R0]
```

Pogram o funkcjonalności tej samej co na listingu 7.4., ale wykorzystujący dyrektywy EQU został przedstawiony na listingu 7.5.

Listing 7.5. Program zaświecający diodę LED TEST z wykorzystaniem WAIT_10e5_CYCLES

```
; REJESTRY MIKROKONTROLERA
RCC_APB2ENR       EQU   0x40021018
GPIOB_CRH         EQU   0x40010C04
GPIOB_BRR         EQU   0x40010C14

; DYREKTYWY ASEMBLERA
      PRESERVE8
      THUMB

; TABLICA WEKTOROW PRZERWAN
      AREA    RESET, DATA, READONLY
      EXPORT  __Vectors

__Vectors
      DCD  0x20001000
      DCD  Reset_Handler

      ALIGN

; PROGRAM
      AREA    MYCODE, CODE, READONLY

      ENTRY
      EXPORT Reset_Handler

Reset_Handler
; PROGRAM UZYTKOWNIKA
      LDR R0, = RCC_APB2ENR
      LDR R1, = 0x00000008
      STR R1, [R0]
      LDR R0, = GPIOB_CRH
      LDR R1, = 0x70000000
      STR R1, [R0]
      LDR R0, = GPIOB_BRR
      LDR R1, = 0x8000
      STR R1, [R0]
      B .

      END   ; KONIEC PROGRAMU
```

Można zauważyć, że stosowanie dyrektyw EQU zwiększa czytelność kodu, a także umożliwia wielokrotne wykorzystywanie wprowadzonych nazw bez konieczności późniejszego pamiętania jakie adresy symbolizują.
Należy zbudować i wgrać do systemu DSM program z listingu 7.5.

Kolejnym krokiem w prezentowaniu możliwości wyjść GPIO niech będzie miganie, czyli naprzemienne załączanie i wyłączanie diody LED TEST. Korzystając z poprzednich dyrektyw EQU znajdujących się na początku pliku oraz po dopisaniu jeszcze jednej:

```
      GPIOB_BSRR  EQU   0x40010C10
```

został napisany program przedstawiony na listingu 7.6, który realizuje miganie diody LED TEST.

Listing 7.6. Program migający diodą LED TEST

```
Reset_Handler
; PROGRAM UZYTKOWNIKA
      LDR R0, = RCC_APB2ENR
      LDR R1, = 0x00000008    ; flaga wlaczajaca signal zegara do portu B
      STR R1, [R0]
      LDR R0, = GPIOB_CRH
      LDR R1, = 0x70000000    ; linia 15 portu B w tryb otwarty dren
      STR R1, [R0]
PETLA
      LDR R0, = GPIOB_BRR
      LDR R1, = 0x8000        ; stan niski na linii 15 portu B
      STR R1, [R0]
      LDR R0, = GPIOB_BSRR    ; stan wysoki na lini 15 portu B
      LDR R1, = 0x8000
      STR R1, [R0]
      B PETLA                 ; skok do etykiety PETLA

      END   ; KONIEC PROGRAMU
```

W programie tym został wykorzystany kolejny rejestr GPIOB_BSRR. Rejestr ten działa analogicznie do GPIOB_BRR z dokładnością do tego, że ustawienie bitów w tym rejestrze powoduje ustawienie jedynki logicznej na danych wyprowadzeniach, a nie jak w przypadku rejestru GPIOB_BRR zera logicznego. Drugim innym elementem jest wykorzystanie pętli. Na końcu programu nie występuje już skok do samego siebie (B .) lecz skok do miejsca oznaczonego etykietą PETLA, to znaczy, że instrukcje zawarte pomiędzy etykietą PETLA, a rozkazem skoku B PETLA będą wykonywane w pętli cały czas. W pętli tej najpierw wykonywane jest zaświecanie diody LED TEST, a następnie z wykorzystaniem rejestru GPIOB_BSRR jej gaszenie. Należy wgrać przedstawiony na listingu 7.6 program i zaobserwować miganie diody.

Jak można zauważyć, po wgraniu programu miganie diody nie jest widoczne. Nie jest to jednak oznaka wadliwie działającego programu. Mikrokontroler przetwarza rozkazy z taką szybkością, że miganie diody wykonywane jest z częstotliwością kilkuset tysięcy razy na sekundę. Oko nie jest w stanie zaobserwować tak szybko zmieniających się obrazów, a jedynie je uśrednia, stąd wydaje się, że dioda LED TEST świeci światłem ciągłym ale nieco słabiej niż przy poprzednich programach (fakt ten jest wykorzystywany w niektórych regulatorach jasności czy też mocy – tzw. sterowanie PWM). Chcąc móc zaobserwować miganie diody należy spowodować, aby przełączanie jej stanu następowało nie częściej niż kilka razy na sekundę. Można w tym celu wykorzystać przygotowaną przez autora procedurę WAIT_10e5_CYCLES . Procedurę tą wpisuje się w kod programu jak zwykły rozkaz, jednak procesor będzie ją przetwarzał przez czas określony wzorem 7.1.

$$\text{liczba\_cykli} = \text{wartość\_w\_rejestrze\_R11} \times 100\,000 \qquad (7.1)$$

To znaczy, że jeżeli procedura ta zostanie wywołana, podczas gdy w rejestrze R11 znajduje się liczba 3, to zajmie ona procesorowi 300 000 cykli zegara, zanim procesor przejdzie do wykonania kolejnego rozkazu. Korzystając więc z takiej procedury można spowodować spowolnienie migania diodą LED TEST. Procedura ta nie jest częścią oprogramowania Keil czy też instrukcją samego procesora. Została opracowana przez autora i w celu poprawnego jej wywołania należy w programie załączyć plik dsm_procedures.inc, w którym procedura jest zdefiniowana. Załączenie pliku dokonuje się

dyrektywą INCLUDE na początku pliku. Całość programu wykorzystująca omawianą procedurę została przedstawiona na listingu 7.7. Skoku do procedury WAIT_10e5_CYCLES dokonuje się rozkazem BL, który powoduje, że aktualna wartość rejestru PC, jest zapisywana do rejestru LR, dzięki czemu procesor po zakończeniu podprogramu WAIT_10e5_CYCLES może wrócić do miejsca, z którego odbył się skok.

Listing 7.7. Program zaświecający diodę LED TEST z wykorzystaniem dyrektyw EQU

```
; PLIKI NAGLOWKOWE
      INCLUDE dsm_procedures.inc          ; import pliku z procedurami

; REJESTRY MIKROKONTROLERA
RCC_APB2ENR       EQU   0x40021018
GPIOB_CRH         EQU   0x40010C04
GPIOB_BRR         EQU   0x40010C14
GPIOB_BSRR        EQU   0x40010C10

; DYREKTYWY ASEMBLERA
      PRESERVE8
      THUMB

; TABLICA WEKTOROW PRZERWAN
      AREA    RESET, DATA, READONLY
      EXPORT  __Vectors

__Vectors
      DCD  0x20001000
      DCD  Reset_Handler

      ALIGN

; PROGRAM
      AREA    MYCODE, CODE, READONLY

      ENTRY
      EXPORT Reset_Handler

Reset_Handler

; PROGRAM UZYTKOWNIKA
      LDR R0, = RCC_APB2ENR
      LDR R1, = 0x00000008
      STR R1, [R0]
      LDR R0, = GPIOB_CRH
      LDR R1, = 0x70000000
      STR R1, [R0]
      MOV R11, #5
PETLA
      LDR R0, = GPIOB_BRR
      LDR R1, = 0x8000
      STR R1, [R0]
      BL WAIT_10e5_CYCLES     ; odczekaj 10e5 x R11 cykli zegara
      LDR R0, = GPIOB_BSRR
      LDR R1, = 0x8000
      STR R1, [R0]
      BL WAIT_10e5_CYCLES     ; odczekaj 10e5 x R11 cykli zegara
      B PETLA                 ; skocz do etykiety petla

      END   ; KONIEC PROGRAMU
```

Więcej o procedurach na jednych z przyszłych zajęć.

Program z listingu 7.7 należy wgrać do systemu DSM i zaobserwować miganie diodą LED TEST. Tym razem miganie diodą będzie widoczne.

Zadania do wykonania

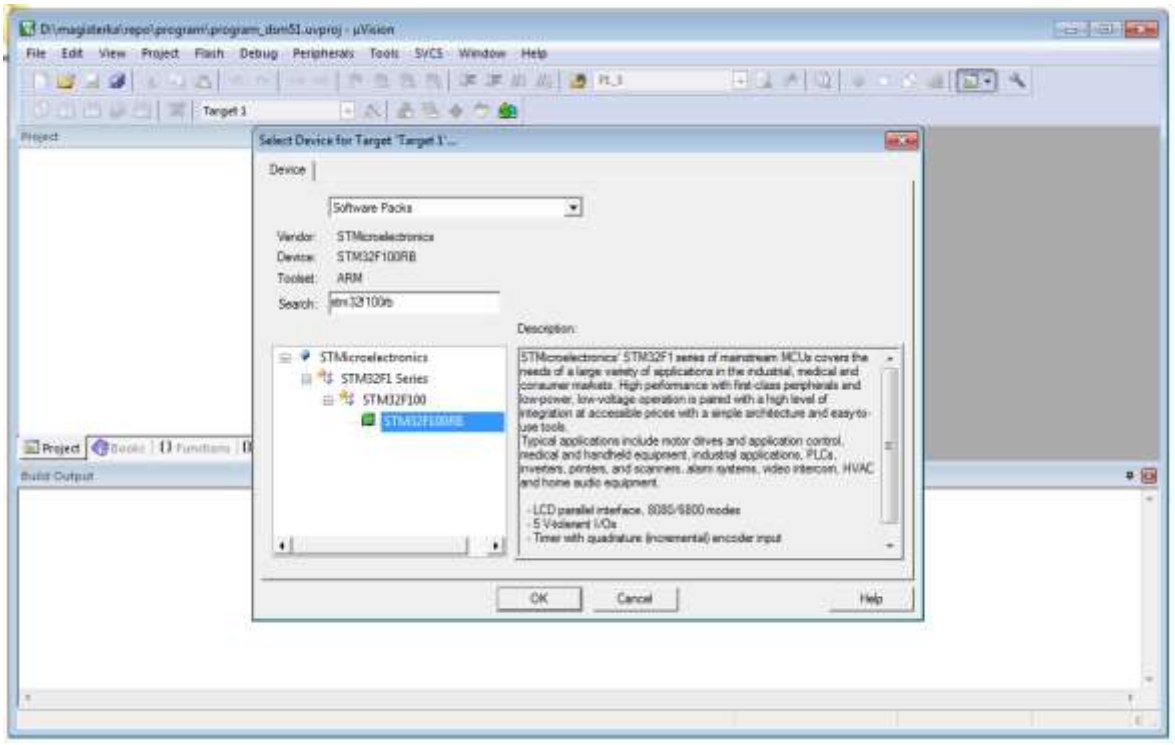
1. Przerobić podane programy tak, aby korzystały z buczka BUZZER przyłączonego do linii 13 portu B mikrokontrolera zamiast diody LED TEST.
2. Przerobić program z listingu 7.7. tak, aby miganie diodą następowało w następującej sekwencji: 2 x szybkie mignięcie, 1 x wolne mignięcie.
3. Na podstawie programu z listingu 7.7. napisać program, który sterowałby naprzemiennie diodą LED TEST i buczkiem BUZZER, to znaczy, gdy dioda jest zaświecona, buczek jest wyciszony, a gdy dioda jest wygaszona buczek jest załączony.

7.3. Skrypt do zajęć wprowadzających – język C

Język C zaliczany jest do języków programowania wysokiego poziomu, w przeciwieństwie do języka asembler, który jest niskopoziomowy. Wysokopoziomowość polega na wprowadzeniu abstrakcyjnych z punktu widzenia mikrokontrolera zapisów, które nie mają bezpośredniego odwzorowania w liście instrukcji procesora, a z drugiej strony są bardziej zwięzłe i czytelne dla programisty. Inaczej mówiąc programowanie w języku C jest szybsze, umożliwiające późniejsze łatwe modyfikacje, zwiększające funkcjonalność w porównaniu z językiem asembler. Nie operuje się w nim już rejestrami ogólnego przeznaczenia czy rejestrami specjalnymi, ani nie używa rozkazów z listy instrukcji danego procesora. Tłumaczeniem programu napisanego w języku C na postać w języku asembler zajmuje się kompilator, a sam proces tłumaczenia nazwany jest kompilowaniem. Kompilatory wykorzystują wysoko zaawansowane algorytmy w celu jak najbardziej optymalnego tłumaczenia jednego języka na drugi, jednak często kod wynikowy w języku asembler jest mniej optymalny niż gdyby napisał go bezpośrednio doświadczony programista asemblera. Stąd z jednej strony język C umożliwia szybsze pisanie programów, ale w przypadku krytycznych aplikacji w dalszym ciągu spotyka się jeszcze programy pisane bezpośrednio w języku asembler. Nie mniej jednak dla zdecydowanej większości aplikacji język C jest szybszy, wygodniejszy i zapewniający odpowiednią wydajność.

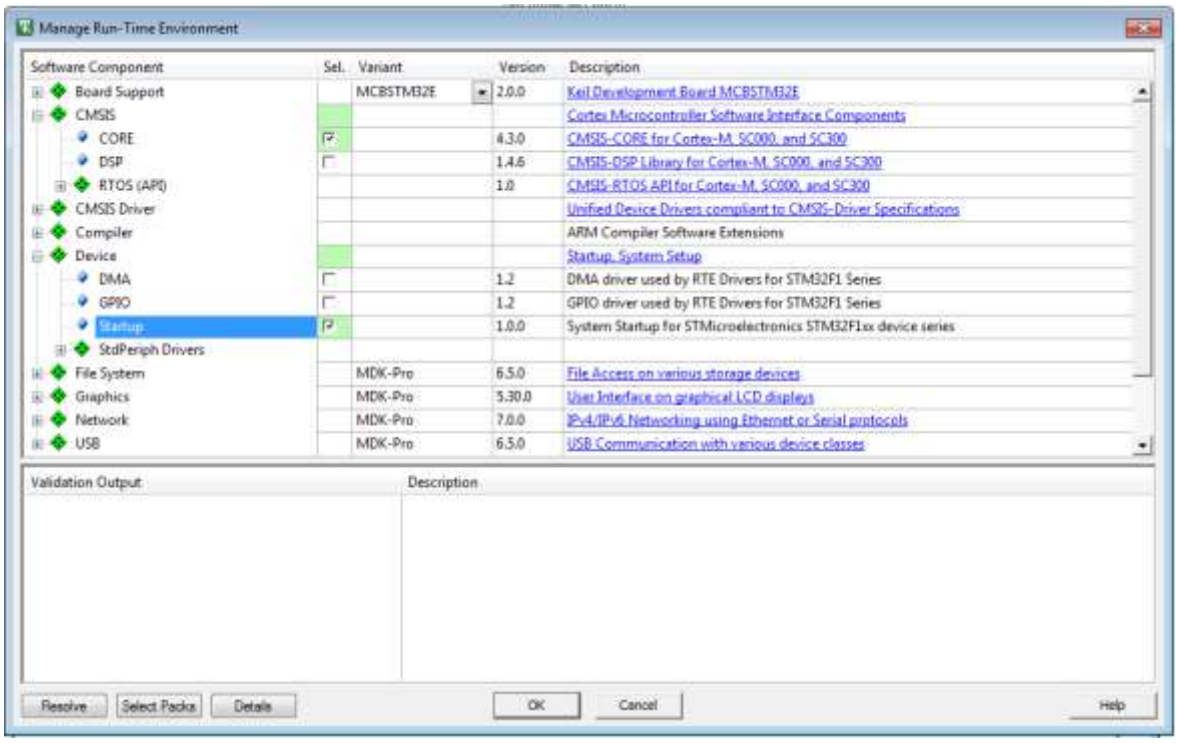
7.3.1. Pierwszy program w języku C

Na komputerze, na który będą przeprowadzane ćwiczenia laboratoryjne należy utworzyć katalog (np. w katalogu domowym użytkownika albo na Pulpicie systemu Windows), w którym później będą znajdowały się pliki źródłowe pisanych programów. Po przygotowaniu katalogu należy uruchomić oprogramowanie Keil uVision5. Domyślnie otwiera się on na projekcie, który był ostatnio uruchamiany. W celu utworzenia nowego należy z menu Project wybrać opcję New µVision Project... . Zostanie wyświetlone okno, w którym należy wskazać wcześniej utworzony katalog oraz podać nazwę nowego projektu, na przykład `dsm_m21.uvprojx` przy czym rozszerzenie pliku może także zostać dodane automatycznie po naciśnięciu przycisku Zapisz. Nazwa nie powinna zawierać polskich znaków, spacji oraz znaków specjalnych. Po naciśnięciu przycisku Zapisz, zostanie przedstawione okno przedstawione na rys. 7.11., w którym to dokonuje się wyboru modelu mikrokontrolera, którego dotyczy projekt. W trakcie wyboru można wykorzystywać okienko Search.



Rys. 7.11. Pierwszy krok tworzenia nowego projektu w Keil MDK-ARM

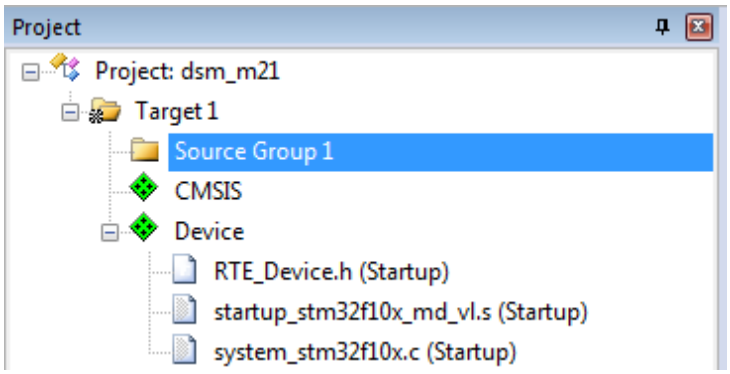
W systemie DSM-51 zastosowany jest mikrokontroler produkcji STMicroelectronics o nazwie STM32F100RB i taki też należy zaznaczyć w wyświetlonym drzewie to lewej stronie. Po naciśnięciu przycisku OK, środowisko prosi użytkownika o określenie, z jakich dostępnych sterowników i bibliotek będzie w danym projekcie korzystał.



Rys. 7.12. Wybór odpowiednich bibliotek dla nowego projektu

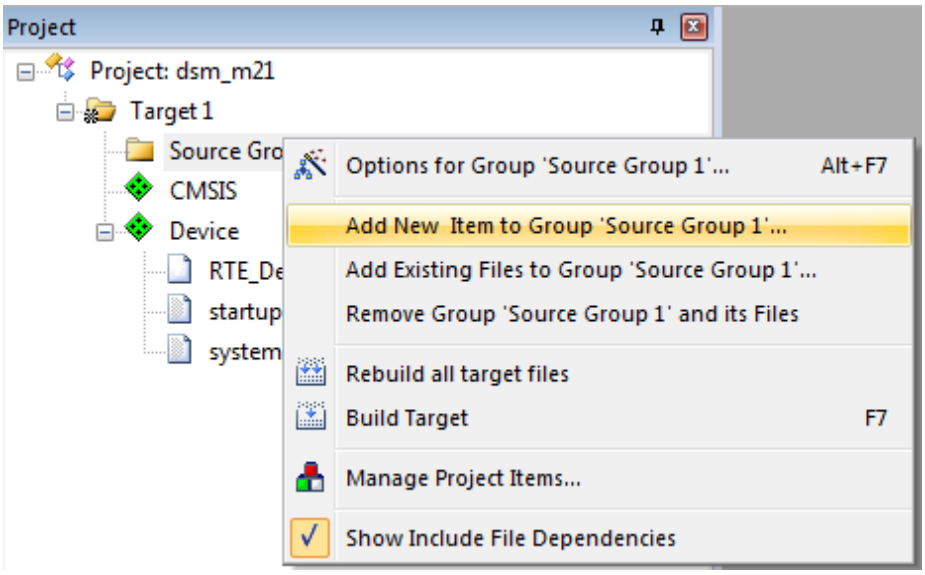
Podczas zajęć laboratoryjnych zostaną wykorzystane dwie biblioteki: CMSIS-Core i System Startup for STMicroelectronics. Taki wybór został przedstawiony na rys. 7.12.

Biblioteka CMSIS-Core zapewnia możliwość odwoływania się do rejestrów mikroprocesora poprzez ich nazwy, a nie operowanie na adresach pamięci. System Startup natomiast pozwala na uruchomienie napisanych, skompilowanych i przesłanych do mikrokontrolera programów. Pozostałe bibliotek w profesjonalnych rozwiązaniach są stosowane, lecz zaciemniają obraz działania mikrokontrolera i pozbawiają napisany program wartości dydaktycznych. Po zatwierdzeniu zostanie ukazany okno programu Keil uVision z drzewem projektu po lewej stronie zawierającym jedynie katalog Target1 z podkatalogiem Source Group 1, przedstawione na rys. 7.13.

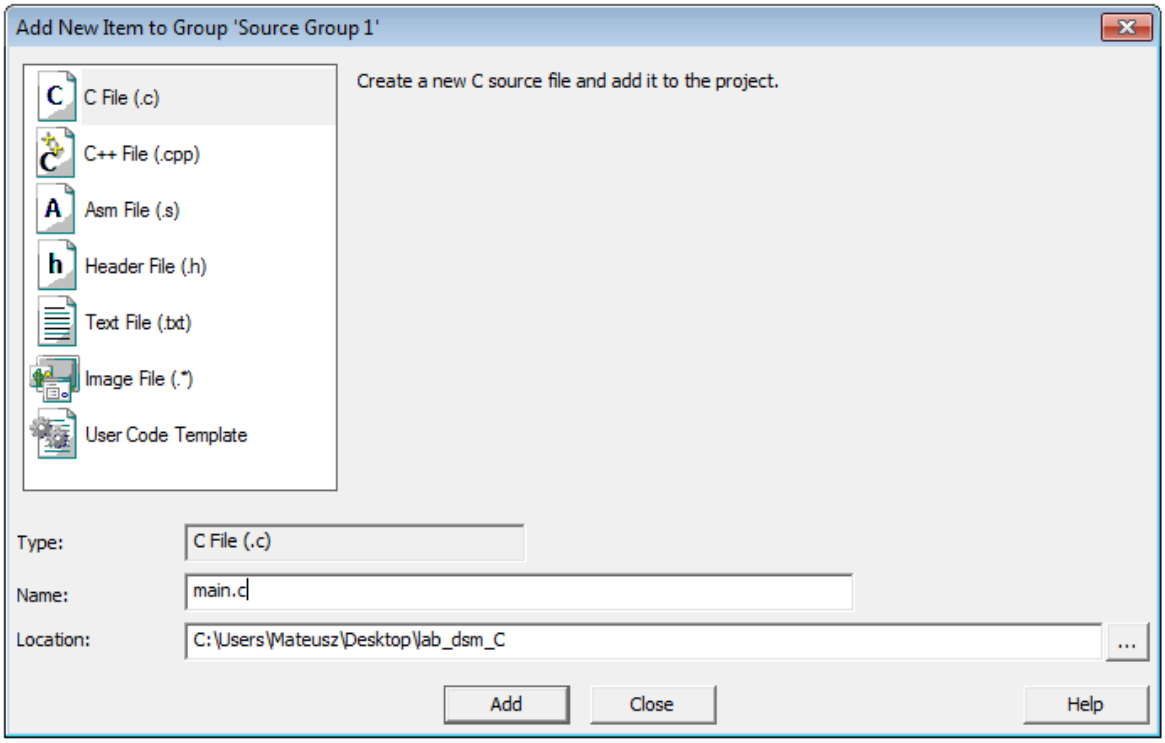


Rys. 7.13. Drzewo nowego projektu bez plików źródłowych

W celu utworzenia i dodania nowego pliku źródłowego do obecnego projektu należy kliknąć prawym przyciskiem myszy na katalog Source Group 1 w drzewie projektu i w nim wybrać opcję Add New Item to Group 'Source Group 1'… co zostało przedstawione na rys. 7.14. Zostanie wówczas wyświetlone okno na rys. 7.15., w którym należy wybrać typ pliku C File (.c) oraz nadać mu nazwę na przykład main.c. Nazwa nie powinna zawierać polskich znaków, spacji oraz znaków specjalnych. Po naciśnięciu przycisku Add drzewo projektu po lewej stronie okna będzie zawierało nowo utworzony plik main.c w podkatalogu Source Group 1, a część środkowa okna Keil uVison będzie zawierała otwarty edytor pliku main.c, w którym to będzie następowało pisanie programu. Przed tym jednak należy jeszcze skonfigurować programator w celu umożliwienia wgrywania i debugowania napisanych programów do systemu DSM-51.

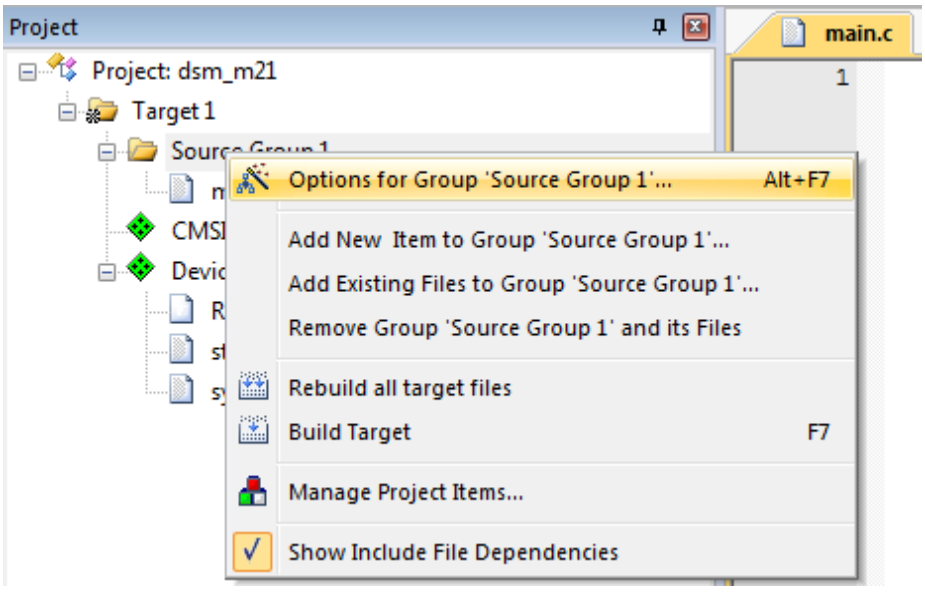


Rys. 7.14. Tworzenie nowego pliku źródłowego i dodawanie go do projektu

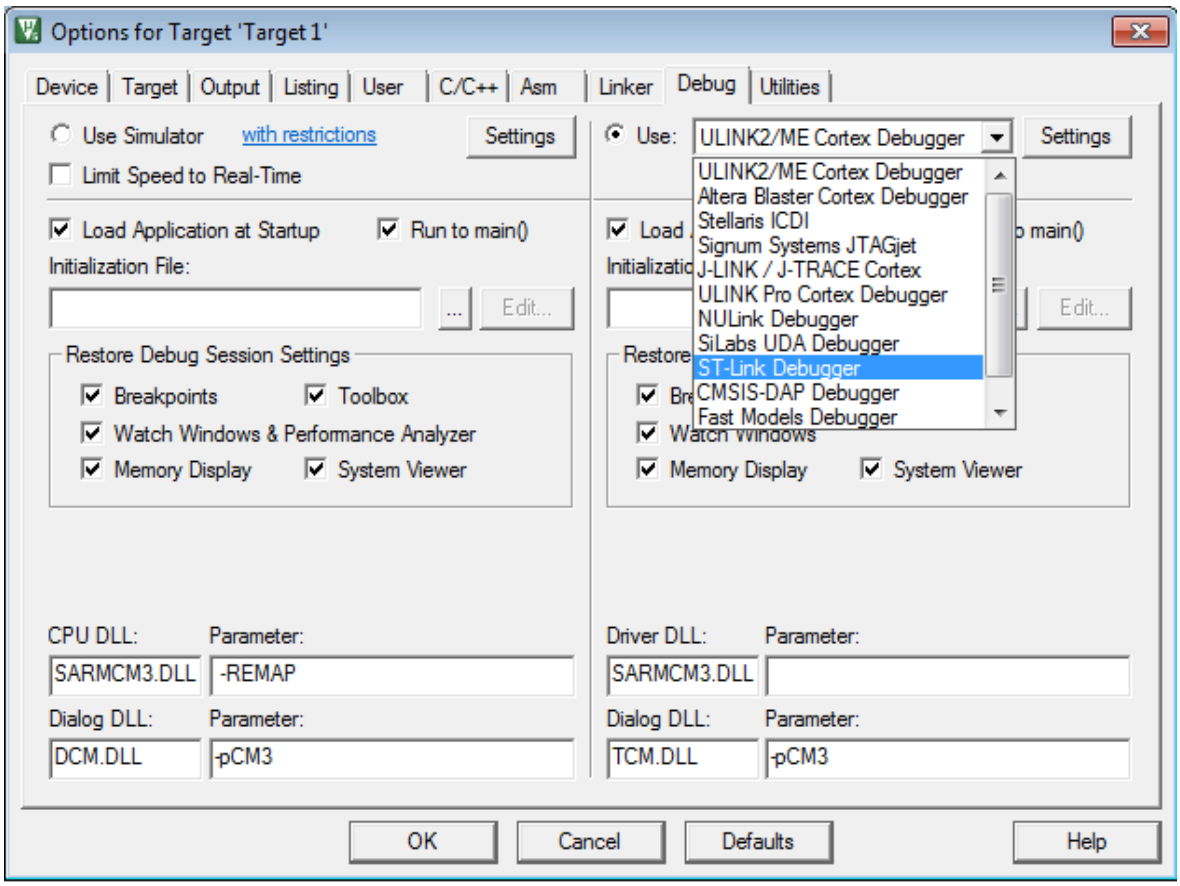


Rys. 7.15. Parametryzowanie nowego pliku źródłowego

W tym celu należy prawym przyciskiem myszy kliknąć w katalog Target 1 w drzewie projektu po lewej stronie i z menu kontekstowego wybrać opcję Options for Target 'Target 1'…, co zostało przedstawione na rys. 7.16. W menu konfiguracyjnym należy wybrać zakładkę Debug i z listy dostępnych programatorów po prawej stronie wybrać ST-Link Debugger. Widok zakładki z wyborem programatora został przedstawiony na rys. 7.17. Po wybraniu z listy programatora ST-Link Debugger jeszcze należy wejść w jego ustawienia klikając na przycisk Settings znajdujący się obok listy z programatorami. Tam w zakładce Flash Download należy zaznaczyć opcję Reset and Run. Następnie kliknąć OK i jeszcze raz OK w oknie z konfiguracją projektu. W tak skonfigurowanym środowisku można przystąpić do pisania, asemblacji i wgrywania programu do systemu DSM-51 oraz jego debugowania.



Rys. 7.16. Wejście w tryb konfiguracji projektu



Rys. 7.17. Wybór programatora ST-Link w opcjach konfiguracji projektu

Najprostszy program napisany w języku C na mikrokontroler STM32F100RB został przedstawiony na listingu 7.8.

Listing 7.8. Najprostszy program w języku C na mikrokontroler STM32F100RB

```
#include "stm32F10x.h"        // import pliku z nazwami rejestrow

/* poczatek program */
int main(void)                // funkcja glowna main()
{
  while(1)                    // petal glowna programu
  {
  } // end while(1)

} // end main()
```

W programie tym występują elementy, które będą się znajdować w każdym kolejnym programie pisanym na omawiany mikrokontroler. Program w języku C jest to jak widać pewnego rodzaju tekst zapisany według określonych zasad, które stanowi norma ISO/IEC 9899:2011. Program napisany w języku C nie składa się rozkazów mikrokontrolera jak to miało miejsce w języku asembler lecz ze sformułowań, które pełnią rolę dyrektyw, funkcji, zmiennych, stałych, pętli. Zapis ten następnie tłumaczony jest w procesie kompilacji na postać języka asembler, a następnie w procesie asemblacji na kod maszynowy procesora.

Pierwszym zapisem przedstawionym na listingu 7.8. jest dyrektywa #include z podaną nazwą pliku stm32F10x.h. Dyrektywa ta powoduje dołączenie zawartości wskazanego pliku na początku programu. We wskazanym pliku znajdują się definicje

z przyporządkowaniem adresów rejestrów urządzeń peryferyjnych z ich nazwami, w celu umożliwienia posługiwaniem się nimi w kodzie programu.

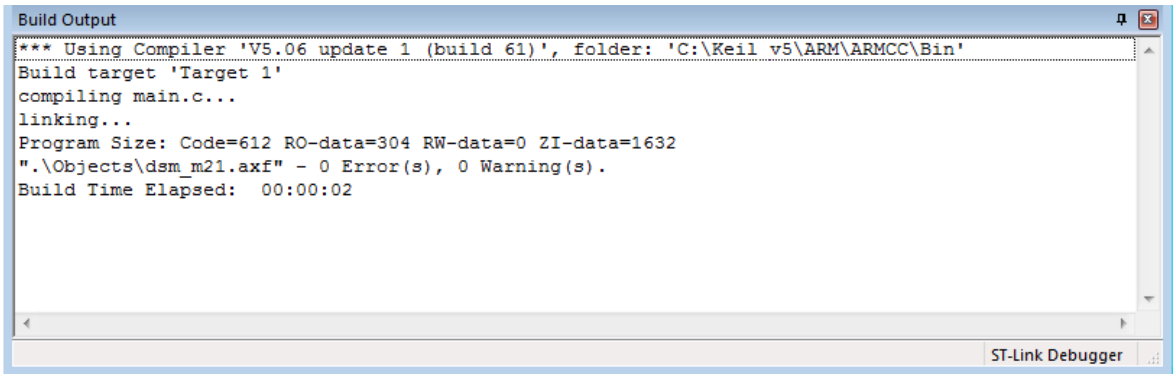
Następnie znajduje się definicja funkcji main(). Funkcja to zbiór poleceń, fragment programu, który przedstawia konkretną funkcjonalność. Można powiedzieć, że funkcja w języku C jest odpowiednikiem podprogramu w języku asembler. Tak zwane ciało funkcji, czyli jej zawartość ograniczone jest klamrami {}. W każdym programie napisanym w języku C musi znajdować się funkcja main(). Do niej to po wstępnym przygotowaniu mikrokontrolera przez bibliotekę System Startup następuje skok w celu rozpoczęcia wykonywania programu użytkownika. Czy w programie znajdą się inne funkcje – zależy to od programisty.

W funkcji main() znajduje się pętla while(1). Pętla ta, jest to pętla nieskończona, która podobnie jak w języku asembler musi występować w programie mikrokontrolera. To co wywoływane jest w pętli zawiera się podobnie jak w przypadku funkcji w klamrach {}.

Język C podobnie jak język asembler przewiduje komentarze w kodzie, które nie są kompilowane, a jedynie stanowią informację, opis dla programisty. Komentarze w języku C zaczyna się od symbolu //, ewentualnie zamyka się je pomiędzy znacznik otwarcia /* i zamknięcia */.

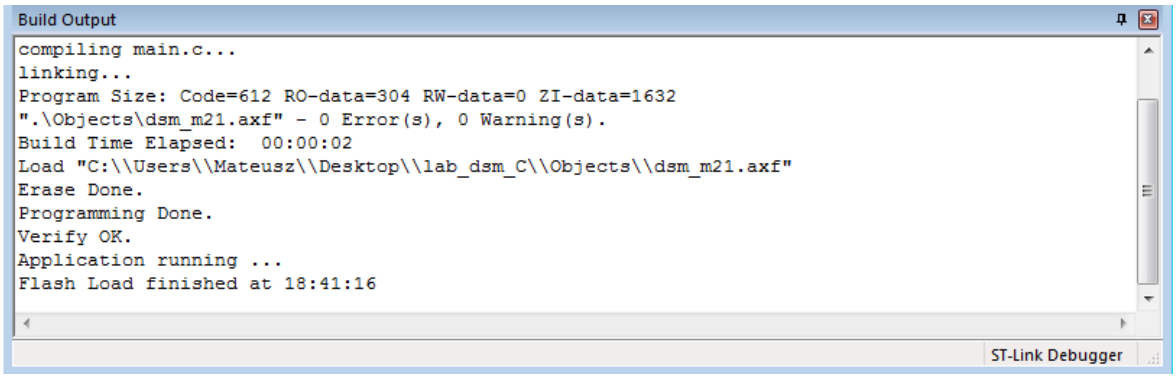
Można zauważyć, że kod najprostszego programu w języku C zawiera znacznie mniej linii niż najprostszy program w języku asembler. Dzieje się tak z dwóch przyczyn, po pierwsze z definicji język C jest bardziej zwięzły, po drugie część kodu znajduje się w plikach bibliotecznych CMSIS, które nie są bezpośrednio widoczne, ale podczas debugowania programu, będzie można zauważyć fragmenty kodu, które z niej pochodzą.

Po zapoznaniu się ze strukturą pliku można przystąpić do pierwszej kompilacji omawianego programu i zaprogramowania mikrokontrolera. W tym celu należy przepisać/przekopiować program zawarty w listingu 7.8. do otwartego w edytorze Keil wcześniej utworzonego pliku ze źródłem programu (w przykładzie jest to plik main.c). Po przekopiowaniu należy zapisać zawartość pliku (np. CTRL+s) a następnie dokonać zbudowania kodu wynikowego. Budowania dokonuje się przyciskiem F7. Po poprawnie przeprowadzonym procesie w oknie komunikatów znajdującym się u dołu programu uVision powinien znajdować się komunikat podobny do tego z rys. 7.18.



Rys. 7.18. Okno komunikatów środowiska Keil uVision po poprawnie przeprowadzonym generowaniu kodu wynikowego

Następnie (o ile nie zostało to przeprowadzone wcześniej) należy podłączyć system DSM-51 kablem USB do komputera, na którym uruchomione jest środowisku Keil uVision i będąc w programie uVision nacisnąć przycisk F8 w celu zaprogramowania mikrokontrolera. Okno komunikatów po poprawnym zaprogramowaniu zostało przedstawione na rys. 7.19.



Rys. 7.19. Okno komunikatów środowiska Keil uVision po poprawnym zaprogramowaniu mikrokontrolera

Przykładowy najprostszy program nie robi nic po za ciągłym wykonywaniem skoku do nieskończonej pętli w związku z czym w systemie DSM-51 nie widać oznak pracy mikrokontrolera. Używanie diody LED TEST, buczka BUZZER, klawiatury czy wyświetlaczy będzie przeprowadzane podczas kolejny zajęć laboratoryjnych.

7.3.2. Zadania do samodzielnego wykonania

Po poprawnym zbudowaniu i wgraniu programu z poprzedniego rozdziału do pamięci mikrokontrolera należy użyć debugera w celu podglądu działania programu. Z debugera dla programów napisanych w języku C korzysta się w ten sam sposób co z debugera dla programów napisanych w języku asembler. Sposób jego wykorzystania został przedstawiony w skrypcie do zajęć wprowadzających pt.: „Lista instrukcji mikrokontrolera STM32F100RBT6B z rdzeniem ARM Cortex-M3". Należy zaobserwować ilość dodatkowych czynności wykonywanych przez procesor po otrzymaniu sygnału reset, które nie zostały zapisane w kodzie programu, a pochodzą z bibliotek CMSIS.

Następnie należy zbudować i wgrać do mikrokontrolera program przedstawiony na listingu 7.9 oraz przeanalizować jego działanie używając debugera.

Listing 7.9. Program wykorzystujący zmienne

```
#include "stm32F10x.h"

int main(void)
{
      int jablka, gruszki, owoce;   // zmienne typu int

      jablka = 5;       // przypisz wartość 5 zmiennej jablka
      gruszki = 7;      // przypisz wartość 7 zmiennej gruszki

      owoce = jablka + gruszki;     // rownanie

      if (owoce > 5)    // jeżeli wartosc zmiennej owoce > 5 to:
            jablka = 2; // przypisz zmiennej jablka wartosc 2
      else              // w przeciwnym wypadku:
            jablka = 8; // przypisz zmiennej jablka wartosc 8

  while(1); // petla glowna programu
}
```

W programie tym zostały wykorzystane zmienne o nazwach: jabłka, gruszki i owoce. Zmienne to tak jakby pojemniki przechowujące pewną wartość. Można dostrzec tu analogię do rejestrów ogólnego przeznaczenia, przy czym liczba zmiennych nie jest ograniczona do dwunastu. Zmienne mogą być różnego typu, to znaczy przechowywać wartości o różnych zakresach, być stało- lub zmiennoprzecinkowe. W programie została

też wykorzystana instrukcja warunkowa if – else, która sprawdza czy jest spełniony podany warunek i w zależności od wyniku porównania wykonuje jedną z dwóch czynności.

Kolejny program do przeanalizowania z użyciem debugera został przedstawiony na listingu 7.10. Program ten zawiera fragment wcześniejszego, z pominięciem instrukcji warunkowej if – else. W trakcie kompilacji użytkownik zostanie poinformowany ostrzeżeniem, że zmienna owoce została zmodyfikowana, ale nie jest w dalszej części programu użyta. Po zaprogramowaniu i uruchomieniu debugera, można zauważyć, że program w języku asembler znaczenie odbiega od tego, który został stworzony poprzez kompilację powyższego programu. W tym obecnym używane są jedynie instrukcje NOP zamiast instrukcji MOV czy ADD tak jak to miało miejsce w poprzednim programie. Dzieje się tak dlatego, że kompilator wykrył, iż zmienne, które zostały utworzone i zmodyfikowane, nie są później wykorzystywane w programie. Na skutek przeprowadzonej optymalizacji usunął więc ten fragment programu w czasie kompilacji w celu zaoszczędzenia czasu procesora, miejsca w pamięci programu oraz pamięci operacyjnej. Gdyby utworzone zmienne zostały później wykorzystane (np. wyświetlone na wyświetlaczu, albo wykorzystane tak jak poprzednio w instrukcji warunkowej) to kompilator skompilowałby cały kod tak jak jest on napisany.

Listing 7.10. Program wykorzystujący zmienne bez instrukcji warunkowej

```
#include "stm32F10x.h"

int main(void)
{
      int jablka, gruszki, owoce;

      jablka = 5;
      gruszki = 7;

      owoce = jablka + gruszki;

      if (owoce > 5)
            jablka = 2;
      else
            jablka = 8;

  while(1);
}
```

Zadania do wykonania

1. Na podstawie podanych przykładów napisać i przeanalizować debugerem program wyliczający silnię liczby podanej w zmiennej „liczba” i zapisujący wynik do zmiennej wynik.
2. Na podstawie podanych przykładów napisać i przeanalizować debugerem program wyliczający kwadrat sumy liczb zawartych w zmiennych a i b i zapisujący wynik do zmiennej C.
3. Na podstawie podanych przykładów napisać możliwie najprostszy program wyliczający podaną zależność: `x = a - (b * 8)`.

7.4. Linie wejść/wyjść mikrokontrolera – język C

Najprostszy program, którego funkcjonalność sprowadza się do zaświecenia diody LED TEST w systemie DSM-51 został przedstawiony na listingu 7.11. Jak należy

rozumieć przedstawiony na nim zapis. Przede wszystkim należy posiadać wiedzę, do którego wyprowadzenia mikrokontrolera przyłączona jest dioda LED TEST. Jest nim wyprowadzenie nr 15 portu B, inaczej mówiąc B15.

Listing 7.11. Program zaświecający diodę LED TEST

```
#include "stm32F10x.h"

int main(void)
{
      RCC->APB2ENR = 0x00000008;    // wlaczenie zegara dla portu B
      GPIOB->CRH = 0x70000000;      // linia 15 w tryb otwarty dren
      GPIOB->BRR = 0x8000;          // stan niski na linii nr 15

  while(1); // petla glowna programu
}
```

Podobnie jak w programie napisanym w języku asembler należy najpierw załączyć sygnał zegara do portu B mikrokontrolera. Załączenie sygnału zegarowego do portu GPIOB odbywa się poprzez ustawienie flagi IOPBEN w rejestrze RCC_APB2ENR. Pisząc w języku C nie należy jednak obliczać adresu rejestru RCC_APB2ENR gdyż dołączone biblioteki umożliwiają programiście odwoływanie się do rejestrów mikrokontrolera z wykorzystaniem ich nazw. Zapis:

```
      RCC->APB2ENR = 0x00000008;
```

oznacza wpisanie wartości 0x0000 0008 do rejestru RCC_APB2ENR. W bibliotece bowiem, zdefiniowane są struktury z nazwami rejestrów i odpowiednimi ich adresami w pamięci mikrokontrolera dla poszczególnych urządzeń. RCC jest wskaźnikiem na strukturę dla kontrolera sygnałów zegarowych, a odwołanie się do konkretniej składowej tej struktury (w tym wypadku rejestru APB2ENR) dokonuje się w języku C poprzez operator strzałki „->”. Podane informacje obowiązują dla wszystkich rejestrów urządzeń peryferyjnych

Kolejnym krokiem jest ustalenie trybu pracy linii nr 15 portu B. Linia będzie skonfigurowana jako wyjściowa, typu otwarty dren, o maksymalnej częstotliwości pracy 50 MHz. W tym celu do rejestru GPIOB_CRH należy wpisać wartość 0x7000 0000. Kolejna linia w programie, po ustawieniu rejestru RCC_APB2ENR temu właśnie służy.

Ostatnią czynnością jest wymuszenie stanu wysokiego lub niskiego na skonfigurowanej linii. Z konstrukcji systemu DSM-51 wynika, że dioda LED TEST zaświeci się, gdy na linii nr 15 portu B będzie panował stan niski. Można to uzyskać wpisując w odpowiedni bit rejestru GPIOB_BRR wartość 1. Służy temu kolejna linia przedstawionego kodu.

Program przedstawiony na listingu 7.11. należy zbudować i wgrać do systemu DSM-51, a następnie zaobserwować zaświecenie się diody LED TEST. Patrząc się na podany program, można odnieść wrażenie, że jest on dużo bardziej zrozumiały i czytelny w porównaniu z programem tworzącym analogiczną funkcjonalność ale zapisanym w języku asembler. Należy przeanalizować działanie programu z użyciem debugera.

Kolejnym krokiem w prezentowaniu możliwości wyjść GPIO niech będzie miganie, czyli naprzemienne załączanie i wyłączanie diody LED TEST. Korzystając z poprzedniego programu został napisany kolejny przedstawiony na listingu 7.12, który realizuje miganie diody LED TEST. W programie tym został wykorzystany kolejny rejestr GPIOB_BSRR. Rejestr ten działa analogicznie do GPIOB_BRR z dokładnością do tego, że ustawienie bitów w tym rejestrze powoduje ustawienie jedynki logicznej na danych wyprowadzeniach, a nie jak w przypadku rejestru GPIOB_BRR zera logicznego. Drugim innym elementem jest wykorzystanie pętli. Na końcu programu nie występuje już skok do

samego siebie (while(1)) lecz wykonywanie w nieskończonej pętli zawartych tam instrukcji. W pętli tej najpierw wykonywane jest zaświecanie diody LED TEST, a następnie z wykorzystaniem rejestru GPIOB_BSRR jej gaszenie.

Listing 7.12. Program migający diodą LED TEST

```
#include "stm32F10x.h"

int main(void)
{
      RCC->APB2ENR = 0x00000008;
      GPIOB->CRH = 0x70000000;

  while(1) {      // wykonywanie w niekonczacej sie petli:
            GPIOB->BRR = 0x8000;    // stan niski na lini 15 portu B
            GPIOB->BSRR = 0x8000;   // stan wysoki na linii 15 portu B
      }
}
```

Należy wgrać przedstawiony na listingu 7.12 program i zaobserwować miganie diody.
Jak można zauważyć, po wgraniu programu miganie diody nie jest widoczne. Nie jest to jednak oznaka wadliwie działającego programu. Mikrokontroler przetwarza rozkazy z taką szybkością, że miganie diody wykonywane jest z częstotliwością kilkuset tysięcy razy na sekundę. Oko nie jest w stanie zaobserwować tak szybko zmieniających się obrazów, a jedynie je uśrednia, stąd wydaje się, że dioda LED TEST świeci światłem ciągłym ale nieco słabiej niż przy poprzednich programach (fakt ten jest wykorzystywany w niektórych regulatorach jasności czy też mocy – tzw. sterowanie PWM). W celu sprawdzenia czy program rzeczywiście realizuje zaświecanie i gaszenie diody należy przeprowadzić inspekcję jego działania debugerem. Chcąc móc zaobserwować miganie diody w trakcie normalnej pracy mikrokontrolera należy spowodować, aby przełączanie jej stanu następowało nie częściej niż kilka razy na sekundę.

Listing 7.13. Program zaświecający diodę LED TEST z funkcją wait()

```
#include "stm32F10x.h"

void wait(int czas) {         // funkcja wait, ktora zajmuje procesor
      long j = 1000000;       // na ilosc cykli = czas x 1000000, gdzie
      while(czas>0) {         // czas to wartosc przekazywana do funkcji
            while(j>0) {
                  __asm("NOP");  // wykonanie asemblerowej instrukcji NOP
                  j--;        // dekrementacja zmiennej j
            }
            czas--;           // dekrementacja zmiennej czas
      }
}

int main(void)
{
      RCC->APB2ENR = 0x00000008;    // zegar dla portu B
      GPIOB->CRH = 0x70000000;      // linia 15 w tryb otwarty dren

  while(1) {      // wykonuj w niekonczacej sie petli:
            GPIOB->BRR = 0x8000;    // stan niski na linii 15 portu B
            wait(10);               // odczekaj 10 x 1000000 cykli
            GPIOB->BSRR = 0x8000;   // stan wysoki na linii 15 portu B
            wait(10);               // odczekaj 10 x 1000000 cykli
      }
}
```

Można w tym celu napisać i wykorzystać funkcję, która zajęłaby procesor na kilkaset cykli zegara w celu spowolnienia zmiany stanu diody. Program wykorzystujący omawianą funkcję został przedstawiony na listingu 7.13. Należy go wgrać do systemu DSM i zaobserwować miganie diodą LED TEST. Tym razem miganie diodą będzie naprawdę widoczne.

Zadania do wykonania

1. Przerobić podane programy tak, aby korzystały z buczka BUZZER przyłączonego do linii 13 portu B mikrokontrolera zamiast diody LED TEST.
2. Przerobić program z listingu 7.13. tak aby miganie diodą następowało w następującej sekwencji: 2 x szybkie mignięcie, 1 x wolne mignięcie.
3. Na podstawie programu z listingu 7.13. napisać program, który sterowałby naprzemiennie diodą LED TEST i buczkiem BUZZER, to znaczy gdy dioda jest zaświecona, buczek jest wyciszony, a gdy dioda jest wygaszona buczek jest załączony.

Literatura

[1] Gałka P., Gałka P.: Podstawy programowania mikrokontrolera 8051, wydanie IV, MIKOM, Warszawa, sierpień 2005,

[2] Kowalik R.: Podstawy techniki mikroprocesorowej dla elektroenergetyków, wykład,

[3] „Microchip największym na świecie dostawcą mikrokontrolerów 8-bitowych w 2014 r.", ElektronikaB2B, czerwiec 2015,

[4] ARM: Licencje udzielone przez ARM Holdings: http://www.arm.com/products/processors/licensees.php,

[5] V.P. Guruprasad: Cortex-M And Classical Series ARM Architecture Comparisons,

[6] Katedra Radiokomunikacji Politechniki Poznańskiej: Treść ćwiczenia „1 Wprowadzenie", Laboratoria dotyczące mikroprocesora ARM Cortex-M4, Poznań, 2015,

[7] STMicroelectronics: "UM0919 User Manual STM32VLDISCOVERY", June 2011,

[8] ARM: "ARM® Debug Interface Architecture Specification ADIv5.0 to ADIv5.2", August 2013,

[9] STMicroelectronics: "PM0056 Programming manual STM32F10xxx/20xxx/21xxx/L1xxxx Cortex-M3 programming manual", May 2013,

[10] Bungo J.: "The ARM Architecture (with focus on Cortex-M3)", ARM University Program, May 2009,

[11] ARM: "AMBA® 3 AHB-Lite Protocol v1.0 Specification", June 2006,

[12] STMicroelectronics: "RM0041 Reference manual STM32F100xx", July 2011,

[13] ARM: "Multi-layer Overview AHB", May 2004,

[14] ARM: "AMBA Design Kit Revision: r3p0 Technical Reference Manual", August 2007,

[15] ARM: "AMBA™ Specification (Rev 2.0)", May 1999,

[16] ARM: "Cortex™-M3 Revision: r1p1 Technical Reference Manual", June 2007,

[17] STMicroelectronics: "STM32F100x4 STM32F100x6 STM32F100x8 STM32F100xB Low & medium-density value line, advanced ARM®-based 32-bit MCU with 16 to 128 KB Flash, 12 timers, ADC, DAC & 8 comm interfaces Datasheet - production data", June 2015,

[18] MicroMade: "DSM-51 rev.2 Dokumentacja techniczna", 1998,

[19] Temic: „80C31/80C51 CMOS 0 to 42 MHz Single-Chip 8 Bit Microcontroller", 1995,

[20] Atmel: "32-Bit Atmel AVR Microcontroller AT32UC3A0512", 2012,

[21] Freescale Semiconductor: "MCF5213 ColdFire Microcontroller", 2007,

[22] Renesas: "RX110 Renesas Group MCUs 32 MHz 32-bit RX MCUs, 50 DMIPS", 2014,

[23] Microchip: „PIC32MX1XX/2XX 28/36/44-PIN 32-bit Microcontrollers (up to 256 KB Flash and 64 KB SRAM)", 2015,

[24] Infineon: Schemat blokowy mikrokontrolerów AURIX http://www.infineon.com/export/sites/default/en/product/promopages/32-bit-microcontroller-for-automotive/images/aurix_architektur_big.jpg,

[25] Bogusz J.: Mikrokontrolery z rdzeniem 32-bitowym (1), Elektronika Praktyczna, czerwiec 2013.