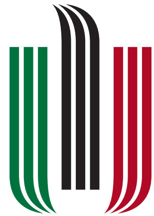


Akademia Górniczo-Hutnicza im. Stanisława Staszica w Krakowie
Wydział Fizyki i Informatyki Stosowanej

Praca magisterska

Robert Gałat

kierunek studiów: Informatyka Stosowana

specjalność: Grafika komputerowa i przetwarzanie obrazów

Warstwa komunikacji z urządzeniami zewnętrznymi w standardzie CMSIS mikrokontrolerów ARM

Opiekun: dr inż. Krzysztof Świentek

Kraków, 19 listopada 2020

Oświadczenie studenta

Uprzedzony(-a) o odpowiedzialności karnej na podstawie art. 115 ust. 1 i 2 ustawy z dnia 4 lutego 1994 r. o prawie autorskim i prawach pokrewnych (t.j. Dz. U. z 2018 r. poz. 1191 z późn. zm.): „Kto przywłaszcza sobie autorstwo albo wprowadza w błąd co do autorstwa całości lub części cudzego utworu albo artystycznego wykonania, podlega grzywnie, karze ograniczenia wolności albo pozbawienia wolności do lat 3. Tej samej karze podlega, kto rozpowszechnia bez podania nazwiska lub pseudonimu twórcy cudzy utwór w wersji oryginalnej albo w postaci opracowania, artystyczne wykonanie albo publicznie zniekształca taki utwór, artystyczne wykonanie, fonogram, wideogram lub nadanie.", a także uprzedzony(-a) o odpowiedzialności dyscyplinarnej na podstawie art. 307 ust. 1 ustawy z dnia 20 lipca 2018 r. Prawo o szkolnictwie wyższym i nauce (Dz. U. z 2018 r. poz. 1668 z późn. zm.) „Student podlega odpowiedzialności dyscyplinarnej za naruszenie przepisów obowiązujących w uczelni oraz za czyn uchybiający godności studenta.", oświadczam, że niniejszą pracę dyplomową wykonałem(-am) osobiście i samodzielnie i nie korzystałem(-am) ze źródeł innych niż wymienione w pracy.

Jednocześnie Uczelnia informuje, że zgodnie z art. 15a ww. ustawy o prawie autorskim i prawach pokrewnych Uczelni przysługuje pierwszeństwo w opublikowaniu pracy dyplomowej studenta. Jeżeli Uczelnia nie opublikowała pracy dyplomowej w terminie 6 miesięcy od dnia jej obrony, autor może ją opublikować, chyba że praca jest częścią utworu zbiorowego. Ponadto Uczelnia jako podmiot, o którym mowa w art. 7 ust. 1 pkt 1 ustawy z dnia 20 lipca 2018 r. — Prawo o szkolnictwie wyższym i nauce (Dz. U. z 2018 r. poz. 1668 z późn. zm.), może korzystać bez wynagrodzenia i bez konieczności uzyskania zgody autora z utworu stworzonego przez studenta w wyniku wykonywania obowiązków związanych z odbywaniem studiów, udostępniać utwór ministrowi właściwemu do spraw szkolnictwa wyższego i nauki oraz korzystać z utworów znajdujących się w prowadzonych przez niego bazach danych, w celu sprawdzania z wykorzystaniem systemu antyplagiatowego. Minister właściwy do spraw szkolnictwa wyższego i nauki może korzystać z prac dyplomowych znajdujących się w prowadzonych przez niego bazach danych w zakresie niezbędnym do zapewnienia prawidłowego utrzymania i rozwoju tych baz oraz współpracujących z nimi systemów informatycznych.

....

(czytelny podpis)

Kraków, listopad 2020

Tematyka pracy magisterskiej i praktyki dyplomowej Roberta Gałata, studenta drugiego roku studiów drugiego stopnia na kierunku informatyka stosowana, specjalności grafika komputerowa i przetwarzanie obrazów

Temat pracy magisterskiej:
Warstwa komunikacji z urządzeniami zewnętrznymi w standardzie CMSIS mikrokontrolerów ARM

Opiekun pracy:	dr inż. Krzysztof Świentek
Recenzent pracy:	dr inż. Jakub Moroń
Miejsce praktyki dyplomowej:	WFiIS AGH, Kraków

Program pracy magisterskiej i praktyki dyplomowej

1. Omówienie realizacji pracy magisterskiej z opiekunem.
2. Zebranie i opracowanie literatury dotyczącej tematu pracy.
3. Praktyka dyplomowa:
   - analiza dokumentacji CMSIS-Driver,
   - zapoznanie się z literaturą dotyczącą systemów wbudowanych,
   - wybór mikrokontrolera do testów,
   - przygotowanie testu wydajności interfejsu CMSIS-Driver,
   - sporządzenie sprawozdania z praktyki.
4. Zebranie i opracowanie wyników.
5. Przygotowanie aplikacji opartej o CMSIS-Driver.
6. Przeniesienie przygotowanej aplikacji na inny mikrokontroler.
7. Opracowanie redakcyjne pracy.

Termin oddania w dziekanacie:

....
(podpis kierownika katedry) (podpis opiekuna)

Spis treści

Wstęp 1

1 Systemy wbudowane 3

1.1 Peryferia w systemach wbudowanych . . . 5

1.2 Funkcjonalna warstwa abstrakcji nad peryferiami . . . 10

2 Koszt sprzętowej warstwy abstrakcji 15

2.1 Założenia testów . . . 15

2.2 Realizacja testu . . . 21

2.2.1 Implementacja oparta o rejestry . . . 21

2.2.2 Implementacja oparta o STM-HAL . . . 28

2.2.3 Implementacja oparta o CMSIS-Driver . . . 33

2.3 Porównanie wyników . . . 41

3 Przykładowa aplikacji oparta o CMSIS-Driver 43

3.1 Struktura oraz funkcjonalności projektu . . . 43

3.2 Implementacja komunikacji z wykorzystaniem CMSIS-Driver . . . 46

3.3 Przenoszenie implementacji z STM na LPC . . . 56

3.4 Wnioski . . . 56

Podsumowanie 59

Literatura 60

Wstęp

Systemy wbudowane stały się integralną częścią naszej codzienności. Znajdziemy je niemal wszędzie w samochodach, domach czy miejscach pracy. Coraz większą popularnością cieszą się inteligentne domy, które są niemal wypełnione systemami wbudowanymi. Zarządzają one oświetleniem, ogrzewaniem, sterują zasłonami, multimediami, a nawet otwierają drzwi. Każdy z tych podsystemów (komputerów wbudowanych) został stworzony do realizacji jednego jasno zdefiniowanego celu np. wspomnianego sterowania zasłonami. Jest to cecha systemów wbudowanych odróżniające je od komputerów osobistych, które mogą mieć wiele zastosowań.

Programowanie systemów wbudowanych jest zadaniem bardzo trudnym ze względu na znaczną ilość trudności. Podstawowym ograniczeniem jest dostępna ilość zasobów takich jak pamięć oraz moc obliczeniowa. Ponadto wprowadzanie aktualizacji na systemach wbudowanych bywa uciążliwe, szczególnie jeśli projekt nie korzysta z sieci, a aktualizacja wymaga fizycznego dostępu do produktu. Oznacza to, że błąd wprowadzony w trakcie procesu tworzenia oprogramowania może nigdy nie zostać naprawiony. Fakt ten motywuje programistów do dokładnego testowania oraz tworzenia kodu jak najwyższej jakości.

Rozwój oprogramowania na systemy wbudowane jest zatem długim procesem, który musi być powtarzany dla każdego nowego produktu. Oczywiście, jeśli nowy produkt korzysta z tego samego mikrokontrolera, możliwe jest przeniesienie części kodu, co skraca czas tworzenia produktu, oraz zwalnia z testowania niskopoziomowych elementów kodu, które były testowane w innym projekcie.

Proces przenoszenia kodu znacznie ułatwia jednolita warstwa abstrakcji nad niskopoziomowymi elementami mikrokontrolera. Programiści korzystają z różnych rozwiązań w zależności od wymagań projektu, nad którym pracują. W systemach medycznych każdy element systemu musi być bardzo dokładnie sprawdzony, a potencjalna perspektywa poprawiania kodu dostarczonego przez zewnętrzną jednostkę, sprawia, że w tak krytycznych obszarach programiści zazwyczaj decydują się na implementację własnych warstw abstrakcji, aby zapewnić odpowiedni poziom jakości kodu. Inne, mniej krytyczne projekty, mogą mieć inne priorytety. Nie trudno sobie wyobrazić scenariusz, kiedy szybkie dostarczenie działającego produktu będzie znacznie ważniejsze niż dostarczenie produktu idealnego. W takich projektach akceptuje się ryzyko wystąpienia drobnych błędów, które mogą być poprawiane w kolejnych wersjach urządzenia.

Standard CMSIS-Driver obiecuje wprowadzenie wygodnej warstwy abstrakcji, celując w projekty, gdzie możliwość szybkiego rozwoju oprogramowania jest kluczowa. Standard ten obiecuje również możliwość przeniesienia kodu opartego o interfejs CMSIS-Driver na inną platformę. W trakcie procesu prototypownia jest to niezwykle przydatna funkcjonalność. Zazwyczaj tego typu projekty rosną z czasem, co może spowodować, że mikrokontroler wybrany początkowo do danego projektu, stanowi ograniczenie w dalszym rozwoju. CMSIS-Driver obiecuje w takiej sytuacji względnie prosty transfer istniejącego rozwiązania na dowolny inny mikrokontroler, co w przypadku korzystania z HAL (*ang. Hardware Abstraction Layer*) dostarczonego przez pro-

ducenta mikrokontrolera, lub własnego autorstwa, jest niezwykle trudne a przede wszystkim czasochłonne.

Celem tej pracy jest przetestowanie interfejsu CMSIS-Driver jako sprzętowej warstwy abstrakcji, przeanalizowanie wydajności dostarczonej implementacji oraz praktyczne sprawdzenie możliwości przenoszenia projektu na nową platformę.

Pierwszy rozdział stanowi krótki wstęp do systemów wbudowanych, gdzie opisana jest schematyczna budowa mikrokontrolera oraz działanie peryferiów. Rozdział ten wprowadza również czytelnika w koncepcję warstwy abstrakcji nad sprzętem. Drugi rozdział przedstawia testy wydajności implementacji wykorzystujących rozwiązania dostarczone przez producenta mikrokontrolera, niskopoziomowe podejście oraz interfejs CMSIS-Driver. Trzeci rozdział prezentuje praktyczne wykorzystanie CMSIS-Driver. Projekt stworzony na bazie CMSIS-Driver zostaje przeniesiony na mikrokontroler wyprodukowany przez innego producenta, co miało za zadanie sprawdzić jedną z kluczowych zalet tego interfejsu.

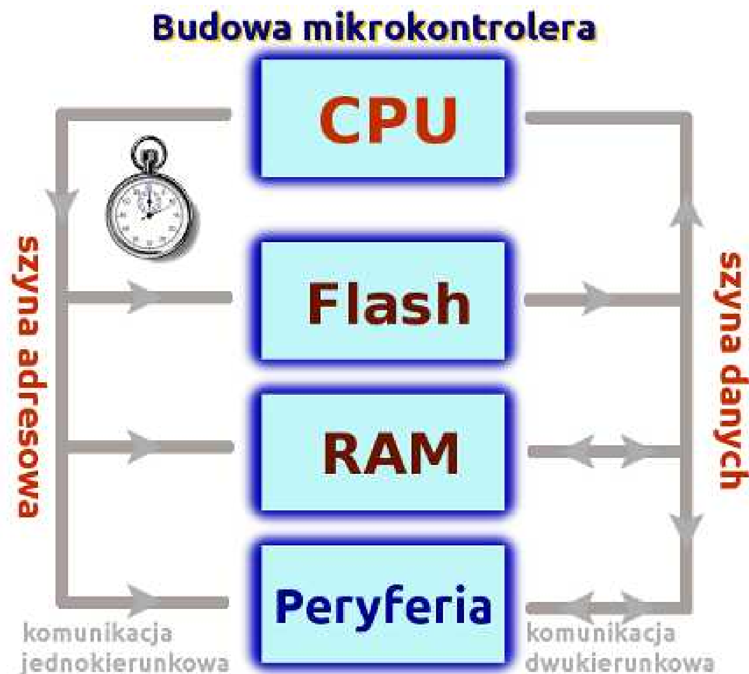
1 Systemy wbudowane

Systemem wbudowanym nazywamy połączenie oprogramowania oraz fizycznego urządzenia składającego się z mikrokontrolera i peryferiów z nim połączonych. System wbudowany możemy znaleźć nawet w najbardziej niespodziewanych miejscach. Dobrym przykładem takiego systemu jest automatyczny ekspres do kawy. Urządzenie takie musi monitorować poziom wody oraz ziaren kawy w zasobniku, ponadto cierpliwie czeka na wybór rodzaju kawy przez użytkownika. Do zmielenia ziaren wykorzysta silnik, podgrzania wody grzałkę a do zaparzenia kawy pompę. Wszystkie te elementy muszą być kontrolowane przez centralny układ, który w zależności od wybranego przez użytkownika napoju dostosuje odpowiednią ilość ziaren, wody i przeprowadzi ekstrakcję przy właściwym ciśnieniu. Układ ten, nazywamy mikrokontrolerem, ponieważ pozwala on na wykonanie jedynie zdefiniowanego zestawu operacji w przeciwieństwie do komputera, który zaprojektowany jest do pełnienia wielu różnych funkcji.[6].

Schematyczną budowę mikrokontrolera przedstawia rysunek 1. Widać na nim podstawowe elementy znajdujące się w każdym mikrokontrolerze[1]. CPU (*ang. central processing unit*) czyli procesor jest to urządzenie, posiadające zdefiniowany i ograniczony zestaw operacji, przypisanych do konkretnych instrukcji. Najprostszymi instrukcjami może być np. załadowanie wartości do wewnętrznego rejestru, wykonanie operacji arytmetycznej na wartościach zapisanych w rejestrach lub wczytanie kolejnej instrukcji. Oczywiście zestaw instrukcji uzależniony jest od zastosowania danej jednostki. Na prezentowanym schemacie widzimy także dwa rodzaje pamięci. Flash to nieulotna pamięć zawierająca kod programu oraz dane tylko do odczytu. Pamięć RAM (*ang. Random Access Memory*) służy do przechowywania wartości w czasie działania programu. To tam tworzony jest stos, na który odkładane są zmienne i wywołania funkcji, jak również sterta, do której programista może uzyskać dostęp, alokując pamięć dynamicznie za pomocą funkcji `malloc`. Niżej znajduje się blok reprezentujący peryferia. Wszystkie te elementy są połączone za pomocą szyny adresowej oraz szyny danych. Służą one do komunikacji pomiędzy tymi elementami. Warto podkreślić, że wszystkie wspomniane elementy znajdują się w jednym układzie scalonym.

Peryferia wbudowane w mikrokontroler dobierane są przez producenta w zależności od planowanego zastosowania danej jednostki. Peryferiami mogą więc być bardzo różne bloki funkcjonalne takie jak np. moduł komunikacji USART (*eng. Universal Synchronous Asynchronous Receive Transmit*), czyli proste urządzenie służące do transmisji danych synchronicznie lub asynchronicznie w zależności od konfiguracji. Urządzenia niewymagające wyrafinowanych peryferiów mogą ograniczyć się do kilku sprzętowych układów licznikowych, GPIO (*ang. General Purpose Input Output*) oraz kilku prostych protokołów komunikacyjnych takich jak SPI (*eng. Serial Peripheral Interface*), czy I2C (*eng. Inter-Integrated Circuit*). Zaawansowane układy

[1] Ben Eater na swoich filmach buduje prosty mikrokontroler na płytkach prototypowych z dyskretnych komponentów, gdzie bardzo dobrze wyjaśnia działanie poszczególnych elementów, z których składa się mikrokontroler `https://www.youtube.com/watch?v=HyznrdDSSGM&list=PLowKtXNTBypGqImE405J2565dvjafglHU`



Rysunek 1: Schemat budowy mikrokontrolera.

poza bardziej wydajną jednostką obliczeniową będą najprawdopodobniej posiadać sprzętowe wsparcie dla skomplikowanych interfejsów takich jak USB lub Ethernet. Oczywiście istnieją różne rozwiązania na dostarczenie tych protokołów do słabszych jednostek, jednak wymagają one specjalnych zewnętrznych modułów, które pełnią funkcję mostu pomiędzy interfejsem takim jak np. Wi-Fi a mikrokontrolerem. Wykorzystują one zazwyczaj SPI lub I2C do transferu danych, co niesie za sobą między innymi ograniczenia w przepustowości.

Jednym z ważniejszych peryferiów w systemach wbudowanych jest DMA (*eng. Direct Memory Access*). Służy ono do transferu danych z pominięciem CPU. Zazwyczaj każda operacja dostępu do danych przechodzi przez CPU i zajmuje cenne zasoby, w przypadku dużych transferów można skorzystać z DMA, które po odpowiednim skonfigurowaniu automatycznie dokona transferu danych, zwalniając CPU z niepotrzebnej pracy. Istnieją 3 możliwe konfiguracje DMA: pamięć – peryferium, peryferium – peryferium oraz pamięć – pamięć. Pierwszy z nich to klasyczne wysyłanie lub odbieranie danych z urządzeń komunikacyjnych takich jak UART, I2C, czy Ethernet. Drugi rodzaj transferu występuje, gdy dane należy przesłać z jednego peryferium do drugiego. Takie zastosowanie jest przydatne w sytuacji, kiedy aplikacja pozyskuje dane z czujnika podpiętego do jednego peryferium, a dane te ma udostępniać na innym. W takiej konfiguracji najciekawsze jest to, że CPU w trakcie transferu może być całkowicie uśpione, a transfer zostanie zrealizowany pomiędzy działającymi peryferiami. Ostatni rodzaj transferu kopiuje dane z jednego obszaru pamięci do drugiego i ma dość wąskie zastosowanie, ponieważ kopiowanie znacznych ilości danych to antywzorzec, którego należałoby unikać w projekcie aplikacji. Jednak kiedy taki transfer jest konieczny, DMA może wykonać tę operację względnie tanio, nie obciążając CPU [2].

Wiele projektów ma szereg zadań do spełnienia, które bardzo ciężko byłoby wydajnie odzwierciedlić w kodzie. Między innymi w tym celu powstał RTOS (*ang. real-time operating system*). Jest to rodzaj oprogramowania pozwalający wygodnie definiować zadania, które mikrokontroler będzie wykonywał w tym samym czasie. Oczywiście mówiąc o systemach opartych o jeden rdzeń obliczeniowy, równoczesność tych zadań jest symulowana poprzez przełączanie kontekstu. Istnieje wiele implementacji systemów operacyjnych czasu rzeczywistego, dzielą się one na dwie kategorie: sterowane wydarzeniami oraz sterowane upływem czasu [4]. Pierwsza kategoria uzależnia przełączenie zadania od wystąpienia wydarzenia. Może nim być np. wybudzenie się zadania o wyższym priorytecie lub uśpienie aktualnego zadania, które pozwala uruchomić się zadaniu o niższym priorytecie. Podobnie, kiedy aktualne zadanie dotrze do `mutex`, czyli specjalnego obiektu, który uprawnia jego posiadacza do dostępu do zasobu. Jeśli inne zadanie wcześniej zawłaszczyło `mutex`, aktualne zadanie może jedynie czekać na jego zwolnienie. Oczywiście oczekiwanie to polega na zmianie kontekstu do innego zadania. Może się zdarzyć, że wszystkie zadania oczekują na `mutex`. W takiej sytuacji system operacyjny może podjąć decyzję o uśpieniu mikrokontrolera w celu zredukowania poboru energii, ponieważ zadania czekają na zewnętrzne zdarzenie. Jest to standardowa praktyka w projektowaniu systemów opartych o zdarzenia. Po inicjalizacji zadań te zatrzymują się na `mutex` który jest zwalniany dopiero przez fizyczne przerwanie, co powoduje wybudzenie odpowiedniego zadania i wznowienie pracy. Taka aplikacja wykonuje więc minimalną konieczną ilość pracy i nie marnuje energii w trakcie oczekiwania na zdarzenia. Druga kategoria opiera się na idei, według której każde zadanie ma przydzieloną określoną ilość czasu. Po jego upływie kontrola przekazywana jest do innego zadania. Oznacza to, że zadania muszą być tak konstruowane, aby nie przekroczyły swojego budżetu czasu procesora. Takie zachowanie spowodowałoby błąd systemu, ponieważ kontekst przełączyłby się do innego zadania niezależnie od stanu obecnie wykonywanego.

Systemy czasu rzeczywistego różnią się więc od znanych nam systemów operacyjnych z komputerów osobistych[2]. Korzystając z RTOS, jako programiści dostajemy bibliotekę funkcji pozwalających symulować wielowątkowość, która automatycznie zarządza tworzonymi wątkami oraz pamięciom do nich przypisaną. Ponadto RTOS dostarcza narzędzia pozwalające rozwiązywać problemy, jakie wynikają z wielowątkowości, takie jak prioretyzacja zadań lub blokowanie zasobów.

1.1 Peryferia w systemach wbudowanych

Istnieje kilka różnych metod dostępu do peryferiów stosowanych na przestrzeni lat od specjalnych instrukcji procesora, poprzez dodatkową przestrzeń adresową aż do umieszczenia peryferium w jednolitej przestrzeni adresowej wraz z pamięcią operacyjną. To ostatnie podejście

[2]Istnieją kompilacje systemów opartych o jądro Linux na platformy ARM, jednak wymagają one mikrokontrolerów z klasy Cortex-A, co sprawia, że bliżej tym jednostkom do komputerów osobistych niż do mikrokontrolerów.

```c
#define PERIPH_BASE            0x40000000UL
#define AHB1PERIPH_BASE        (PERIPH_BASE + 0x00020000UL)
#define RCC_BASE               (AHB1PERIPH_BASE + 0x3800UL)
#define RCC_AHB1ENR            ((uint32_t *)RCC_BASE) + 12

#define GPIOA_BASE             (AHB1PERIPH_BASE + 0x0000UL)
#define GPIOA_BSSR             ((uint32_t *)GPIOA_BASE) + 6

int main() {
  *RCC_AHB1ENR |= 1;      // uruchom zegar do peryferium
  *GPIOA_BSSR = 1 << 16; // zapal diodę podpiętą pod GPIOA0
  *GPIOA_BSSR = 1;       // zgaś diodę podpiętą pod GPIOA0

  while (1)
    ;
}
```

Listing 1: Zapalenie diody na mikrokontrolerze STM32F407 z wykorzystaniem adresów peryferium GPIO

dobrze współgra ze współczesnymi architekturami procesorów (takimi jak ARM) i jest powszechnie wykorzystywane w nowoczesnych rozwiązaniach. Z powyższego wynika, że każde peryferium w nowoczesnym mikrokontrolerze z punktu widzenia kodu programu widoczne jest jako kawałek pamięci. Konkretny adres, pod którym znajdziemy rejestry interesującego nas peryferium, znaleźć można w dokumentacji danego mikrokontrolera.

Każdy producent może wybrać dowolny zestaw peryferiów, jaki umieści wewnątrz mikrokontrolera, co oznacza, że w różnych urządzeniach podobne funkcjonalnie peryferium może znajdować się pod innym adresem i posiadać inny zestaw rejestrów. Fakt ten stanowi ogromny problem z punktu widzenia przenaszalności kodu.

Rozważmy prosty przykład widoczny na listingach 1 oraz 2, zapalający diodę na przykładowych mikrokontrolerach różnych producentów. Taki kod zadziała jedynie na konkretnym mikrokontrolerze, ponieważ wszystkie adresy zostały zdefiniowane w kodzie programu. Zachowanie poszczególnych rejestrów jest charakterystyczne dla konkretnego peryferium zaprojektowanego przez twórcę mikrokontrolera. Oczywiście poza problemem przenaszalności kodu, takie podejście jest dość niepraktyczne i podatne na błędy. Jeśli każda interakcja z peryferium wymagałaby wyszukania w dokumentacji konkretnego adresu, pod który należy zapisać wartość, to nawet najprostsze projekty byłyby niezwykle skomplikowane.

Z przedstawionych rozwiązań widać, że podstawową potrzebą programisty są pliki zawierające definicje adresów wszystkich układów wewnętrznych. Naprzeciw tym potrzebom wychodzi standard CMSIS-Core [1]. Zgodnie z tym standardem to producent mikrokontrolera dostarcza pliki implementujące podstawowe funkcjonalności mikrokontrolera. Jest to zbiór komponentów

```c
#define FIO0DIR 0x2009C000
#define FIO0SET 0x2009C018
#define FIO0CLR 0x2009C01C

int main(void)
{
  unsigned int volatile * setDIR  = (unsigned int *)FIO0DIR;
  unsigned int volatile * setHIGH = (unsigned int *)FIO0SET;
  unsigned int volatile * setLOW  = (unsigned int *)FIO0CLR;

  *setDIR  |=  1<<0; // ustaw pin p0.0 jako wyjściowy
  *setLOW  |=  1<<0; // zapal diodę
  *setHIGH |=  1<<0; // zgaś diodę

  while (1)
    ;
}
```

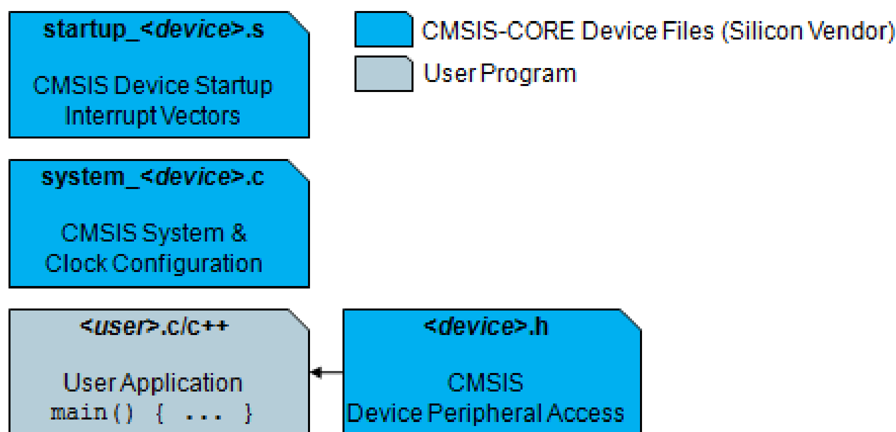
Listing 2: Zapalenie diody na mikrokontrolerze LPC1768 z wykorzystaniem adresów peryferium GPIO

tworzących podstawowe środowisko uruchomieniowe. Standard ten ułatwia również dostęp do peryferiów oraz procesora poprzez implementację między innymi sprzętowej warstwy abstrakcji, systemu wyjątków oraz funkcji definiowanych przez kompilator (ang. *intrinsic functions*).

Elementy CMSIS-Core przedstawia rysunek 2, gdzie za symbol `<device>` należy wstawić nazwę rodziny mikrokontrolerów np. STM32f4xx lub LPC17xx. Plik `startup_<device>.s` zawiera definicję stosu i sterty oraz wektory wszystkich przerwań wraz z domyślną implementacją funkcji wywoływanych w przypadku ich wystąpienia. W tym zbiorze najważniejsza jest funkcja `Reset_Handler` obsługująca reset, która jest pierwszą funkcją wywoływaną po uruchomieniu systemu. Plik `system_<device>.c` zawiera podstawową konfigurację mikrokontrolera. Standard przewiduje definicję zmiennej `SystemCoreClock` i dwóch funkcji w tym pliku:

- `void SystemInit (void)` – odpowiada za poprawną inicjalizację mikrokontrolera, m.in. układ generowania zegara i jest wywoływana z wnętrza `Reset_Handler` przed funkcją `main`.
- `void SystemCoreClockUpdate()` – odpowiada za aktualizację globalnej zmiennej `SystemCoreClock` przechowującej aktualną częstotliwość zegara systemowego,

Ostatni plik dostarczany przez producenta to plik `<device>.h`. Plik ten umożliwia dostęp do peryferiów procesora, poprzez zdefiniowanie struktur danych odpowiadających rejestrom peryferiów obecnych w systemie. Każde peryferium zostało w tym pliku przedstawione w sposób analogiczny do opisu z dokumentacji. Dzięki specjalnym makrom preprocesora zostały przygotowane odpowiednie wskaźniki mapujące adresy urządzeń na struktury je opisujące. Dostępne



Rysunek 2: Organizacja aplikacji korzystającej ze standardu CMSIS-Core; pliki dostarczane przez producenta mikrokontrolera zaznaczono kolorem niebieskim.

są również odpowiednie makra odpowiadające wartościom, jakie chcielibyśmy wpisywać do rejestrów danego peryferium. Dzięki temu korzystanie z peryferiów jest stosunkowo proste.

Na listingach 3 oraz 4 przedstawiono przykład wykorzystania CMSIS-Core dla kodu wcześniej użytego w listingach 1 i 2. Jak widać po dołączeniu pliku `<device>.h`, do dyspozycji programisty dostępne są wszystkie rejestry mikrokontrolera w postaci struktur. W przypadku listingu 3 widać, że plik `stm32f407xx.h` udostępnił programiście makro z wartością umożliwiającą ustawienie rejestru `RCC`. Jego nazwa jednoznacznie wskazuje, dla jakiego rejestru została ona przygotowana, oraz co reprezentuje. W prezentowanym przypadku ustawia ono flagę `GPIOAEN`, której działanie opisane jest w dokumentacji. Analogiczne makra ułatwiające konfigurację rejestrów stanowią znaczną część pliku `stm32f407xx.h`. Nie jest to jednak standardem, ponieważ w przypadku mikrokontrolera `LPC1768`, producent nie zdecydował się na udostępnienie tego typu makr. Ograniczył się jedynie do zdefiniowania struktur rejestrów oraz przygotowania wskaźników mapujących je na odpowiadające im adresy w pamięci mikrokontrolera.

CMSIS-Core rozwiązuje jedynie problem wygodnego dostępu do peryferium. Porównując listingi 3 oraz 4 można zauważyć, że przedstawiony kod nadal jest silnie zależny od mikrokontrolera.

```c
#include <stm32f407xx.h>

int main() {
  RCC->AHB1ENR |= RCC_AHB1ENR_GPIOAEN; // ustaw zegar dla gpio A
  GPIOA->BSRR = 1 << 16;               // zapal diodę podpiętą pod GPIOA0
  GPIOA->BSRR = 1;                     // zgaś diodę podpiętą pod GPIOA0

  while (1)
    ;
}
```

Listing 3: Zapalenie diody na STM32F407 w CMSIS-Core

```c
#include <lpc17xx.h>

int main(void)
{
  LPC_GPIO0->FIODIR |= (1 << 0); // ustaw pin p0.0 jako wyjściowy
  LPC_GPIO0->FIOCLR |= (1 << 0); // zapal diodę
  LPC_GPIO0->FIOSET |= (1 << 0); // zgaś diodę

  while (1)
    ;
}
```

Listing 4: Zapalenie diody na LPC1768 w CMSIS-Core

1.2 Funkcjonalna warstwa abstrakcji nad peryferiami

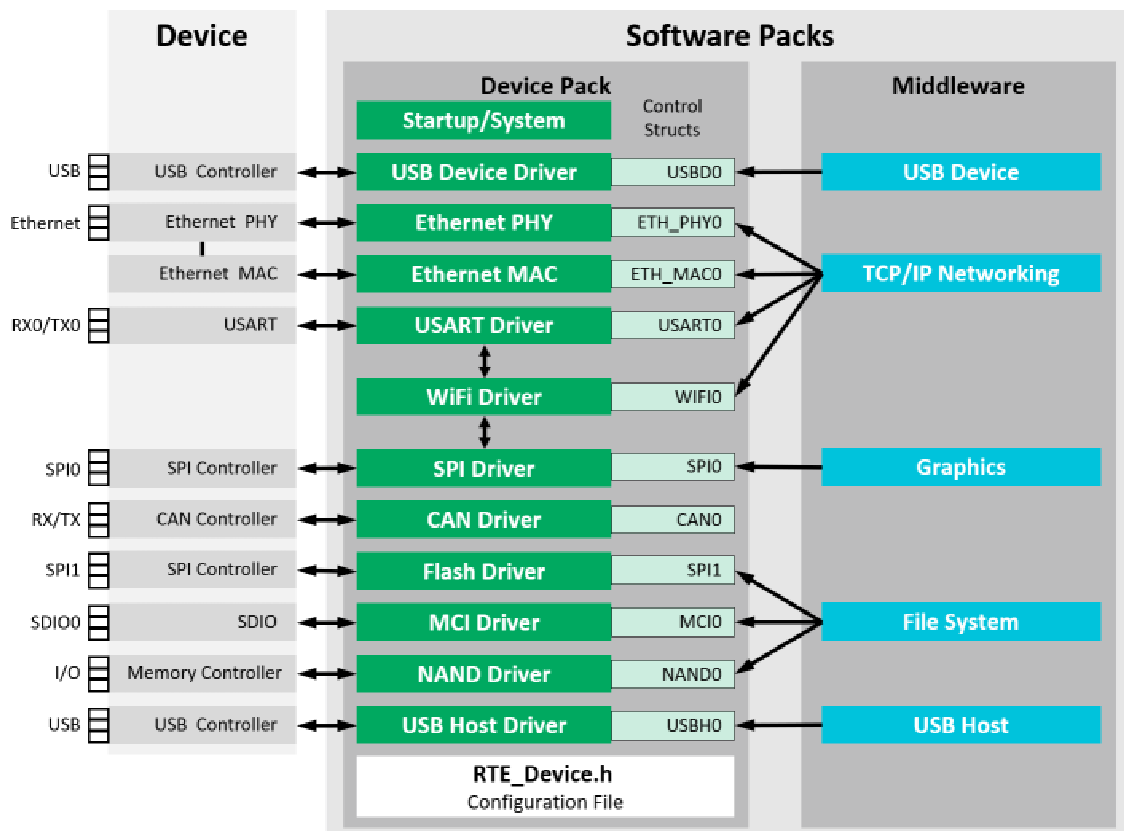
CMSIS-Core stanowi doskonałą podstawę do tworzenia kolejnej warstwy abstrakcji, która skupia się bardziej na tym, co chcemy osiągnąć, a nie w jaki sposób. HAL (ang. *Hardware Abstraction Layer*) jest zestawem funkcji, protokołów oraz narzędzi mających za zadanie wchodzić w interakcję z urządzeniem bez konieczności wnikania w wewnętrzne zawiłości jego obsługi [5]. Dzięki temu tworzenie nawet skomplikowanych projektów jest znacznie prostsze. Programiści mogą skupiać się na logice realizowanej przez projektowaną aplikację, zakładając poprawną implementację komunikacji z urządzeniem przez HAL.

Należy jednak pamiętać, że istnieje wiele równoległych standardów spełniających założenia wspomnianej warstwy. Wszystkie te standardy są tworzone i wspierane z różnych powodów. Producenci mikrokontrolerów zachęcają do korzystania ze swoich rozwiązań, obiecując możliwość szybkiego prototypownia projektów na ich podstawie. Często zapewniają one również przenośność kodu pomiędzy różnymi wersjami mikrokontrolerów znajdujących się w ofercie producenta. Natomiast, niektóre firmy wykorzystujące mikrokontrolery tworzą własne interfejsy, aby spełnić wewnętrzne wymogi wydajności i jakości. Skupiają się zazwyczaj na wąskiej grupie urządzeń wykorzystywanych w projektach.

W ramach standardu CMSIS rolę HAL'a pełni komponent CMSIS-Driver, którego elementy można zobaczyć w centralnej części rysunku 3. Obszar opisany jako Device Pack zawiera elementy CMSIS-Driver, które z lewej połączone są z fizycznymi urządzeniami (obszar Device) z pomocą CMSIS-Core, a z prawej z bibliotekami wysokopoziomowymi implementującymi np. stos TCP/IP, system plików itp.

Peryferia komunikacyjne w mikrokontrolerze muszą zostać podłączone do jego wyprowadzeń, zanim zostaną użyte. Typowo jeden pin może być podłączony do kilku różnych peryferiów, co wyklucza ich równoczesne użycie. Z tego powodu niektóre peryferia mają kilka możliwych konfiguracji wyprowadzeń na wypadek konfliktu. W pliku `RTE_Device.h` zapisana jest konfiguracja, która definiuje wykorzystane urządzenia w projekcie, oraz określa, do których wyprowadzeń mikrokontrolera (tzw. pinów) są one podłączone.

Wszystkie drivery zaprojektowane są w podobny sposób. Podstawą każdego modułu jest globalna struktura, której stworzenie uzależnione jest od konfiguracji zapisanej w `RTE_Device.h` (wykorzystuje on do tego kompilację warunkową). Struktura ta zawiera zestaw wskaźników na funkcje, realizujące funkcjonalność modułu, oraz funkcję konfiguracyjną. Wykorzystanie wskaźników na funkcje w strukturze jest sprytnym podejściem udającym, w języku C, programowanie obiektowe. Listing 5 przedstawia strukturę definiującą zachowanie sterownika protokołu I2C. Funkcje, na które wskazują kolejne pola struktury, wykonują proste i stosunkowo atomowe operacje z punktu widzenia programisty, takie jak: konfiguracja, inicjalizacja, kontrola stanu oraz transfer danych. Te podstawowe operacje są uniwersalne dla każdego modułu I2C, a ich implementacje dostępne są na większość popularnych platform z wykorzystaniem *Software Packs*.



Rysunek 3: Interfejs peryferiów

Są to przygotowane zestawy oprogramowania publikowane przez dostawców na stronie ARM, zawierające implementację do interfejsów z pakietu CMSIS.

W przypadku stosunkowo popularnych mikrokontrolerów produkowanych przez STMicroelectronics implementacja CMSIS-Driver dostarczana jest przez ARM Keil i stanowi alternatywę dla STM-HAL dostarczonej przez STMicroelectronics. Rozwiązanie STM, nie wykorzystuje jednego pliku konfiguracyjnego. Konfiguracja w tym przypadku odbywa się w kodzie programu, razem z inicjalizacją. Listingi 6 i 7 pokazują podstawową konfigurację i prostą operację wysłania danych poprzez I2C, wykorzystując odpowiednio STM-HAL oraz CMSIS-Driver.

Funkcja `void configI2C1(void)` z listingu 6 zawiera inicjalizację wszystkich parametrów I2C. Na początku, udostępniany jest sygnał zegarowy dla peryferium, następnie w dość czytelny sposób ustalane są parametry transmisji, po których obiekt jest inicjalizowany. Następnie należy połączyć to urządzenie do wyprowadzeń mikrokontrolera. Ta operacja wykonywana jest w liniach 30 – 37, gdzie uruchamiane jest GPIO, konfigurowane są piny PB6 oraz PB7, na których ustawiana jest funkcja 4. Jak można znaleźć w dokumentacji, funkcja ta oznacza uruchomienie modułu I2C1 na wspomnianych wyprowadzeniach. Przedstawiony kod może początkowo wydawać się zawiły, jednak jest on dość opisowy, a nazwy pól i wartości, w jasny sposób wskazują jaki element konfiguracji jest ustalany. Zazwyczaj zawartość opisanej funkcji konfiguracyjnej jest wygenerowana z pomocą narzędzia CubeMX udostępnionego przez STMicroelectronics. Dzięki graficznemu interfejsowi w łatwy sposób można wybrać wymagane peryferia, skonfigu-

```
typedef struct _ARM_DRIVER_I2C {
  ARM_DRIVER_VERSION   (*GetVersion)     (void);
  ARM_I2C_CAPABILITIES (*GetCapabilities)(void);
  int32_t              (*Initialize)     (ARM_I2C_SignalEvent_t cb_event);
  int32_t              (*Uninitialize)   (void);
  int32_t              (*PowerControl)   (ARM_POWER_STATE state);
  int32_t              (*MasterTransmit) (uint32_t addr, const uint8_t *data, uint32_t num, bool xfer_pending);
  int32_t              (*MasterReceive)  (uint32_t addr,       uint8_t *data, uint32_t num, bool xfer_pending);
  int32_t              (*SlaveTransmit)  (               const uint8_t *data, uint32_t num);
  int32_t              (*SlaveReceive)   (                     uint8_t *data, uint32_t num);
  int32_t              (*GetDataCount)   (void);
  int32_t              (*Control)        (uint32_t control, uint32_t arg);
  ARM_I2C_STATUS       (*GetStatus)      (void);
} const ARM_DRIVER_I2C;
```

Listing 5: Deklaracja obiektu CMSIS-Driver I2C

rować zegar systemowy i w efekcie wygenerować poprawny[3] szablon projektu.

Drugie podejście do uruchomienia I2C jest przedstawione na listingu 7. W tym przypadku wykorzystany został interfejs CMSIS-Driver. Konfiguracja ograniczona jest jedynie do kilku pierwszych linii. Pozostałe parametry, które ustawione były w poprzednim listingu, mają wartości domyślne, więc z poziomu kodu użytkownika nie musimy ich ustawiać, jeśli nie zamierzamy ich modyfikować. Połączenie modułu I2C1 do konkretnych wyprowadzeń mikrokontrolera odbywa się z pomocą pliku konfiguracyjnego, którego fragment przedstawiony jest na listingu 8. Makro `RTE_I2C1` uruchamia moduł I2C, natomiast modyfikacja makr zawierających `_PORT_ID` konfiguruje, do jakiego wyprowadzenia podłączona zostanie dana funkcja peryferium. Takie podejście gwarantuje, że peryferium będzie poprawnie podpięte do wyprowadzenia, ponieważ wybierana jest jedna ze zdefiniowanych opcji przygotowana pod konkretny mikrokontroler.

Porównując zawartość funkcji `void main()` z listingów 6 oraz 7, można zauważyć, że zapewniają one podobny poziom abstrakcji i wygody użytkowania dla programisty, ponieważ podstawowa operacja wysłania jednego bajtu do urządzenia o adresie 125, w obu przypadkach wymaga wywołania zaledwie jednej funkcji.

Oba standardy znajdują się na podobnym poziomie abstrakcji, umożliwiając wygodny dostęp do peryferiów, jednak sprawdzają się one w różnych zastosowaniach. Rozwiązanie dostarczane przez STMicroelectronics jest dostosowane pod korzystanie z narzędzi od STM, takich jak generator projektów CubeMX, oraz przeznaczone jedynie do mikrokontrolerów produkowanych przez STM. CMSIS-Driver z założenia ma być uniwersalny dla różnych mikrokontrolerów. Główna różnica leży w konfiguracji, która w przypadku STM znajduje się w kodzie, a dla CMSIS-Driver, generowana jest na podstawie warunkowej kompilacji ustawianej w specjalnym pliku. Oczywiście istnieją narzędzia, które pozwalają wygodnie konfigurować CMSIS-Driver, ta-

[3]CubeMX zabezpiecza przed niepoprawną konfiguracją, sygnalizując użytkownikowi problemy i błędy w trakcie konfiguracji projektu.

kie jak Keil μVision, jednak rozwiązanie to nie zapewnia tak kompleksowej obsługi programisty, jak narzędzia dostarczone przez STM.

```
#include "stm32f4xx_hal.h"
I2C_HandleTypeDef i2cInitStruct;
void configI2C1 (void);

int main () {
    uint8_t txData = 0xAA;
    HAL_Init();
    configI2C1();

    HAL_I2C_Master_Transmit(&i2cInitStruct, (125U << 1), &txData, 1, 5);
    while(1)
        ;
}

void configI2C1 (void) {
    GPIO_InitTypeDef GPIO_InitStruct;
    __HAL_RCC_I2C1_CLK_ENABLE();

    i2cInitStruct.Instance = I2C1;
    i2cInitStruct.Init.ClockSpeed = 100000;
    i2cInitStruct.Init.DutyCycle = I2C_DUTYCYCLE_2;
    i2cInitStruct.Init.OwnAddress1 = 242; /* Own Address 121 */
    i2cInitStruct.Init.AddressingMode = I2C_ADDRESSINGMODE_7BIT;
    i2cInitStruct.Init.DualAddressMode = I2C_DUALADDRESS_DISABLE;
    i2cInitStruct.Init.OwnAddress2 = 0;
    i2cInitStruct.Init.GeneralCallMode = I2C_GENERALCALL_DISABLE;
    i2cInitStruct.Init.NoStretchMode = I2C_NOSTRETCH_DISABLE;
    HAL_I2C_Init(&i2cInitStruct);

    __HAL_RCC_GPIOB_CLK_ENABLE();

    GPIO_InitStruct.Pin = GPIO_PIN_6|GPIO_PIN_7;
    GPIO_InitStruct.Mode = GPIO_MODE_AF_OD;
    GPIO_InitStruct.Pull = GPIO_PULLUP;
    GPIO_InitStruct.Speed = GPIO_SPEED_FREQ_VERY_HIGH;
    GPIO_InitStruct.Alternate = GPIO_AF4_I2C1;
    HAL_GPIO_Init(GPIOB, &GPIO_InitStruct);
}
```

Listing 6: Przykład użycia STM-HAL

```c
#include "Driver_I2C.h"

extern ARM_DRIVER_I2C            Driver_I2C1;
int main(){

    Driver_I2C1.Initialize(NULL);
    Driver_I2C1.PowerControl(ARM_POWER_FULL);
    Driver_I2C1.Control(ARM_I2C_BUS_SPEED, ARM_I2C_BUS_SPEED_FAST);
    Driver_I2C1.Control(ARM_I2C_BUS_CLEAR, 0);

    uint8_t txData = 0xAA;
    I2Cdrv->MasterTransmit((125U << 1), &txData, 1, true);
    while(1)
        ;
}
```

Listing 7: Przykład użycia CMSIS-Driver

```c
// <e> I2C1 (Inter-integrated Circuit Interface 1) [Driver_I2C1]
// <i> Configuration settings for Driver_I2C1 in component ::CMSIS Driver:I2C
#define RTE_I2C1                        1

//   <o> I2C1_SCL Pin <0=>PB6 <1=>PB8
#define RTE_I2C1_SCL_PORT_ID            0
#if    (RTE_I2C1_SCL_PORT_ID == 0)
#define RTE_I2C1_SCL_PORT               GPIOB
#define RTE_I2C1_SCL_BIT                6
#elif  (RTE_I2C1_SCL_PORT_ID == 1)
#define RTE_I2C1_SCL_PORT               GPIOB
#define RTE_I2C1_SCL_BIT                8
#else
#error "Invalid I2C1_SCL Pin Configuration!"
#endif

//   <o> I2C1_SDA Pin <0=>PB7 <1=>PB9
#define RTE_I2C1_SDA_PORT_ID            0
#if    (RTE_I2C1_SDA_PORT_ID == 0)
#define RTE_I2C1_SDA_PORT               GPIOB
#define RTE_I2C1_SDA_BIT                7
#elif  (RTE_I2C1_SDA_PORT_ID == 1)
#define RTE_I2C1_SDA_PORT               GPIOB
#define RTE_I2C1_SDA_BIT                9
#else
#error "Invalid I2C1_SDA Pin Configuration!"
#endif
```

Listing 8: Fragment pliku `RTE_Device.h` odpowiedzialnego za konfigurację wyprowadzeń oraz wybór peryferiów.

2 Koszt sprzętowej warstwy abstrakcji

Tworzenie kodu jedynie na podstawie dokumentacji może wydawać się kuszące, ponieważ w ten sposób możemy w pełni wykorzystać możliwości sprzętu. Podejście to jednak niesie za sobą sporo konsekwencji. Kod pisany od podstaw na tym poziomie abstrakcji wymaga ogromnej ilości testów na fizycznym urządzeniu. Jeśli tworzymy interfejs kompatybilny z kilkoma urządzeniami, każdą zmianę wypadałoby przetestować na wszystkich konfiguracjach. Aby ograniczyć ogromną ilość kodu, stosuje się metodę inkrementalnego tworzenia kodu. Wymaga ona, aby tworzyć tylko funkcjonalności, które są konieczne w danej chwili. W taki sposób maksymalizuje się ilość kodu, który aktywnie przyczynia się do postępu projektu. Każdy kod jest jednak podatny na błędy. Zaletą ogólnie dostępnych rozwiązań jest fakt, że skupiają one wokół siebie społeczność, która może weryfikować działanie oprogramowania, na wielu platformach. Dzięki temu, podczas korzystania z tych rozwiązań możemy mieć nadzieję, że nie znajdują się tam poważne błędy. Oczywiście każdy projekt wymaga dokładnego testowania przed wprowadzeniem na rynek, jednak tworząc produkt od podstaw, należy liczyć się z faktem, że projekt będzie powstawał znacznie dłużej, i będzie wymagał znacznie więcej testów, co może wpłynąć na ostateczny koszt tworzonego produktu.

Niezależnie od tego, jakie rozwiązanie jest rozważane, należy zastanowić się nad wydajnością kodu. Celem każdego oprogramowania jest wykonanie określonej operacji. Jeśli tę samą, operacje można zrealizować w mniejszej ilości instrukcji, oznacza to, że finalne urządzenie może mieć większe możliwości, lub być bardziej wydajny energetycznie.

Przy masowej produkcji koszt związany z pracą programistów i testerów staje się mniej istotny, jednak jeśli nie ma ku temu zasadnych przesłanek, nie ma potrzeby wymyślać koła na nowo. Zazwyczaj o wiele taniej jest dostosować znane i popularne rozwiązanie do potrzeb projektu niż tworzyć je od podstaw. Ponadto korzystając z gotowych narzędzi, pozostawiamy sobie możliwość migracji kodu na inną platformę, niezależnie od tego, czy potrzebujemy wydajniejszej jednostki, czy z testów wynika, że słabszy model równie dobrze poradzi sobie z obsługą produktu. W przypadku rozwiązania tworzonego od podstaw taka zmiana wymaga miesięcy na stworzenie odpowiedniej implementacji oraz testy, a dla gotowych systemów czas ten można znacznie ograniczyć.

2.1 Założenia testów

Celem pracy jest próba odpowiedzi na pytanie, w jakim stopniu różne podejścia do implementacji komunikacji z peryferium różnią się w aspekcie wydajności implementacji. W związku z tym wykonano serię pomiarów, aby sprawdzić, ile czasu procesor musi poświęcić na obsługę peryferiów w zależności od sposobu realizacji zadania i wybranej biblioteki.

Do eksperymentu wykorzystano płytkę developerską opartą o mikrokontroler STM32F407 oraz analizator logiczny. Wspomniany mikrokontroler w każdym z testów został skonfigurowany

w taki sposób, aby działać z pełną prędkością 168 MHz generowaną z pomocą zewnętrznego oscylatora 8 MHz. Kod był budowany z wykorzystaniem kompilatora *arm-none-eabi-gcc* w wersji 6.3.1 bez optymalizacji (flaga `-O0`), która mogłaby być znacznym utrudnieniem w przyjętej metodzie testowej.

Do testu wybrano implementację modułu USART, ponieważ jest to stosunkowo proste oraz niezwykle popularne peryferium. Moduł ten nie posiada buforu na dane, a jedynie rejestr wyjściowy o rozmiarze $1B$. Oznacza to więc, że każda z implementacji musi uzupełnić ten rejestr kolejnym bajtem danych po zakończeniu transferu poprzedniego.

Test polega na pomiarze, jak dużo kodu można wywołać (a dokładniej, ile cykli zegara upłynie) w czasie oczekiwania na zakończenie transferu porcji danych przy różnych prędkościach interfejsu przedstawionych na listingu 9. Pozwala to określić ilości kodu wykonywanego w ramach obsługi przerwań, dalej interpretowanego jako obciążenie. W trakcie transferu w głównym programie nie może być wywoływany kod z HAL, ponieważ wykorzystane zostały funkcje nieblokujące. Oddają one kontrolę nad głównym programem zaraz po rozpoczęciu transferu, a sam transfer jest obsługiwany przez przerwania. Każdy z testowanych rozwiązań ma inne podejście do realizacji obsługi przerwań, co skutkuję różną ilością kodu, który ostatecznie wpisuje kolejne bajty danych do rejestru peryferium. Kod wywoływany z obsługi przerwań powinien być możliwie krótki, ponieważ wywołują się one w momencie, kiedy peryferium ich żąda, co powoduje wstrzymanie głównego programu. Zbyt duża ilość kodu w przerwaniach może sparaliżować pracę mikrokontrolera poprzez znacznie ograniczenie ilości kodu wywoływanego w ramach głównego programu. Z tego względu preferowane są implementacje, które realizują minimalną niezbędną ilość kodu w obsłudze przerwań.

```
1  uint32_t common_speeds[] = {
2      4800,   9600,       19200,      38400,
3      57600,  115200,     230400,     460800,
4      921600, 1312500,    2625000,    5250000,
5      10500000, /* MAX Suppoprted Speed */
6  };
```

Listing 9: Testowane prędkości USART

Dzięki porównaniu ilości wykonanego kodu z czasem transferu można stwierdzić jak duże obciążenie dla systemu stanowi testowane rozwiązanie. Wiedza ta może okazać się przydatna, szczególnie kiedy wykorzystywany jest system czasu rzeczywistego, gdzie procesor może przełączać się pomiędzy zadaniami. W takim przypadku, zamiast bezczynnie czekać na zakończenie wysyłania danych mikrokontroler może zostać lepiej wykorzystany w innych zadaniach systemu.

Do pomiaru oczekiwania na zakończenie transferu wykorzystany został analizator logiczny, który śledził stan jednego z wyprowadzeń mikrokontrolera, skonfigurowanego, aby sygnalizować oczekiwanie stanem wysokim. Wykorzystanie zewnętrznego urządzenia daje możliwość do-

kładnego wyznaczenia czasu oczekiwania na zakończenie pracy sterownika. Określenie ilości wykonanego kodu w głównej części programu zostało zrealizowane poprzez zwiększanie licznika z każdym obiegiem pętli, której warunek końcowy ustalał sterownik sygnałem zakończenia transferu. Wartość tego licznika jest więc proporcjonalna do czasu spędzonego w tej pętli, jednak nie można porównywać wartości liczników, powstałych w wyniku testów różnych rozwiązań. Pomimo bardzo podobnej implementacji, nie ma pewności, w jaki sposób kompilator ułoży kod pętli dla kolejnych testów. Można jednak założyć, że jeden obieg pętli wykona się w stałej ilości cykli zegara. Poznanie tej wartości pozwoliłoby na normalizację wartości licznika pomiędzy różnymi testami, pozwalając na ich porównanie.

Próba analizy kodu assembler w celu wyznaczenia ilości instrukcji, skazana jest na porażkę, ponieważ w użytym mikrokontrolerze ilość cykli zegara koniecznych do wykonania danej instrukcji jest różna. Niektóre instrukcje wykonują się w jednym cyklu zegara, jednak wynik innej operacji może być dostępny dopiero po 2 cyklach. Oznacza to, że w zależności jak mikrokontroler ułoży instrukcje do uruchomienia, analizowany kod wykona się w innej liczbie cykli zegara. Podejście takie byłoby zasadne dla prostych mikrokontrolerów, gdzie zestaw instrukcji jest znacznie uproszczony, a sam mikrokontroler mniej wyrafinowany.

Kolejnym sposobem na wyznaczenie brakującej stałej jest odpowiednie przygotowanie kodu. Do kalibracji należy użyć tej samej funkcji, która będzie później wykonywała testy, aby zapewnić, że kalibracja będzie dokonana na kodzie testowym.

Przed kalibracją wyłączone zostały przerwania z USART, aby zapewnić, że nic nie wpłynie na rezultat. Do kalibracji wykorzystano timer, którego konfiguracja została przedstawiona na listingu 10. Po dokładnie $1s$ od uruchomienia wystawi on flagę oznaczającą koniec transferu. W trakcie kalibracji wykonanych zostanie 168 milionów cykli zegara, ponieważ z taką częstotliwością pracuje mikrokontroler w każdym z testów. Jeśli podzielimy tę wartość przez uzyskany w trakcie kalibracji licznik, otrzymamy ilość cykli zegara koniecznych do wykonania jednego obiegu pętli w funkcji testowej.

Ostatecznie wartość porównawczą pomiędzy testami wyznacza wzór 1, gdzie c jest ilością cykli zegara konieczną do wykonania jednego obiegu pętli wyznaczaną dla każdego testowanego sterownika z pomocą timera, l_i jest wartością licznika w danym teście, a t_i czasem działania pętli w sekundach. Kolejne wartości indeksu i oznaczają testowane prędkości transferu.

$$k_i = \frac{c * l_i}{t_i} \tag{1}$$

Wartości k_i stanowią wyznacznik, jak dużo kodu udało się wykonać w głównym programie. Wartości zbliżone do zegara systemowego sugerują minimalne obciążenie, natomiast bliskie 0 wskazują, że przez większość oczekiwania mikrokontroler obsługiwał przerwania.

Funkcje do obsługi testu oraz kalibracji są uniwersalne pomiędzy testowanymi rozwiązaniami. W ten sposób jedyną różnicą w analizowanych rozwiązaniach jest dostarczona implementacja. Funkcja testowa została przygotowana w taki sposób, aby nie ograniczać możliwości

```c
void setup_timer(void **internal, uint32_t baudrate) {
  (void)internal;
  (void)baudrate;
  RCC->APB1ENR |= (1 << 0);

  // Timer clock runs at ABP1 * 2
  //   since ABP1 is set to /4 of fCLK
  //   thus 168M/4 * 2 = 84Mhz
  // set prescaler to 0
  TIM2->PSC = 0;

  // Set the auto-reload value to 84000000
  //   which should give 1 second timer interrupts
  TIM2->ARR = 84000000;

  // Update Interrupt Enable
  TIM2->DIER |= (1 << 0);

  NVIC_SetPriority(TIM2_IRQn, 2); // Priority level 2
  // enable TIM2 IRQ from NVIC
  NVIC_EnableIRQ(TIM2_IRQn);
}
```

Listing 10: Konfiguracja timera do kalibracji.

```
struct test_ctx {
  void * internal;
  void (*configure)(void **internal, uint32_t baudrate);
  bool (*transfer)(void *internal, uint8_t *data, uint16_t size);
  bool (*deinit)(void *internal);
};
```

Listing 11: Deklaracja struktury przechowującej kontekst testu.

testowanych rozwiązań. Implementacje podstawowych operacji dostarczane są z pomocą struktury `struct test_ctx` przedstawionej na listingu 11. Struktura ta zawiera wskaźniki na funkcje do konfiguracji peryferium, umożliwiające rozpoczęcie transferu, oraz deinicjalizację peryferium. Poza tymi funkcjami struktura ta zawiera również wskaźnik na typ `void`, który służy jako zasób, który może być wypełniony podczas inicjalizacji, i wykorzystany dalej w innych funkcjach sterownika. Dzięki temu możliwe jest nie tylko łatwe przenoszenie testu pomiędzy projektami testującymi analizowane implementacje, ale również w ten sposób dokonywana jest kalibracja. Funkcje do obsługi timera, które służą do kalibracji, są kompatybilne z narzuconym interfejsem, aby można było wstrzyknąć je do funkcji testowej bez rekompilacji projektu, co mogłoby wpłynąć na uzyskane wyniki.

Kod odpowiedzialny za test widoczny jest na listingu 12. Wartość zwracana z funkcji `test_performance` sygnalizuje czy test wykonał się poprawnie. Parametry tej funkcji oznaczają kolejno kontekst testu zawierający wskaźniki na funkcje konieczne do konfiguracji i uruchomienia testu, prędkość USART, oraz wskaźnik, pod którym będzie zapisana wartość licznika. Funkcja sprawdza wszystkie wskaźniki przed pierwszym użyciem, czy nie mają wartości `NULL`, ponieważ w takim przypadku próba ich dereferencji spowoduje błąd i niezdefiniowane zachowanie procesora[4]. Pierwszą operacją (po sprawdzeniu odpowiednich wskaźników) w funkcji testowej jest skonfigurowanie peryferium, służy temu funkcja `configure` ze struktury `ctx`. Następnie resetowana jest flaga oznaczająca koniec transferu, a dane do wysłania są randomizowane. Operacja ta zapewnia, że ani mikrokontroler, ani peryferium nie wykona żadnej optymalizacji przy transferze. Następnie rozpoczyna się transfer dzięki kolejnej funkcji ze struktury `ctx` i rozpoczyna się pętla oczekująca. Po zakończeniu transferu konieczna jest deinicjalizacja peryferium, aby zapewnić prawidłowe warunki dla kolejnego testu. W przypadku kalibracji na tym etapie wyłączane jest przerwanie do timera, aby nie wpływał on na wynik testu. Po prawidłowej deinicjalizacji pozostaje jedynie przypisać uzyskany wynik pod wskaźnik `counter`, aby zakończyć przypadek testowy.

[4]Istnieją rozwiązania, które wymuszą wywołanie `Hard_Fault` przy próbie dereferencji wskaźnika o wartości `NULL` i wymagają one wykorzystania MPU procesora. Domyślnie taka operacja ma niezdefiniowane zachowanie.

```
bool test_performance(struct test_ctx *ctx, uint32_t baud, uint32_t *counter) {
  static uint8_t data[500];
  uint32_t cnt = 0;

  if (ctx == NULL)
    return false;
  if (ctx->configure == NULL)
    return false;

  ctx->configure(&ctx->internal, baud);

  UART_TransferComplete = false;
  randomize_payload(data, sizeof(data));
  if (ctx->transfer(ctx->internal, data, sizeof(data)) == false)
    return false;

  GPIOA->ODR &= (uint32_t) ~(1 << 4);
  while (!UART_TransferComplete) {
    cnt++;
  }
  GPIOA->ODR &= (uint32_t) ~(1 << 4);

  if (ctx->deinit == NULL)
    return false;
  ctx->deinit(ctx->internal);

  if (counter == NULL)
    return false;

  *counter = cnt;
  return true;
}
```

Listing 12: Funkcja testowa wykorzystywana w każdym teście.

2.2 Realizacja testu

Zdecydowano się przeprowadzić testy dla trzech różnych podejść do programowania peryferiów w mikrokontrolerach: bezpośrednich odwołań do rejestrów, wykorzystując bibliotekę HAL-STM i w oparciu o CMSIS-Driver. W każdym z tych przypadków sprawdzono dwie implementacje bazujące tylko na przerwaniach o wykorzystującą DMA.

2.2.1 Implementacja oparta o rejestry[5]

Implementacja oparta o rejestry, nie jest oczywiście przenaszalna, jednak stanowi dobre porównanie górnej granicy wydajności, jakiej można się spodziewać.

Aby lepiej zrozumieć działanie tego prostego sterownika, na listingach od 13 do 18 przedstawiono najważniejsze elementy jego implementacji.

Głównym zadaniem sterownika poza odpowiednią konfiguracją parametrów transferu jest uzupełnianie rejestru odpowiedzialnego za wysyłanie danych. W tym celu może również zostać wykorzystane DMA. Jest to peryferium, które zajmuje się obsługą transferu danych. Niektóre transfery DMA mogą odbywać się, nawet gdy procesor jest w trybie uśpienia, a dane mają przepływać pomiędzy dwoma peryferiami. Dzięki oddelegowaniu zadania wysyłania danych do osobnego peryferium możliwe jest lepsze wykorzystanie mikrokontrolera.

Listing 13 odpowiada za inicjalizację urządzenia. Wiele parametrów peryferium ustalonych jest na stałe wartości, ponieważ nie są one konieczne do testu. Należy jednak zwrócić uwagę, że prawdziwy sterownik, powinien uwzględnić możliwość, ustawienia tych parametrów, tak aby nie ograniczać funkcjonalności peryferium. Przedstawione podejście zakłada implementację jedynie koniecznych elementów do wykonania testu, więc większość parametrów transferu jest stała. Funkcja `get_bbr_for_speed()` jest funkcją pomocniczą. Ma ona za zadanie wyznaczyć wartość rejestru `BBR`. Rejestr ten jak wskazuje dokumentacja, odpowiada za wyznaczenie prędkości transmisji. Implementacja tej funkcji jest odzwierciedleniem instrukcji opisanych w dokumentacji [9, str. 978]. Przechodząc dalej do zasadniczej funkcji konfiguracyjnej, pierwszą operacją, jaką należy wykonać, to uruchomić peryferium. Odbywa się to poprzez udostępnienie zegara dla peryferium, co realizuje linia 15 listingu. Następnie należy upewnić się, że GPIO, na którego wyprowadzeniach spodziewamy się konfigurowanego modułu, jest uruchomione. Kolejną operacją, jest skonfigurowanie funkcji USART na pinach, których używamy[6]. W liniach 22 – 27 ustawiany jest tryb działania wyprowadzenia oraz wybierana jest funkcja dla danego wyprowadzenia. Wspomniane linie kodu mają za zadanie skonfigurować moduł GPIO, tak aby na wyprowadzeniach $PA9$ oraz $PA10$ podłączony został moduł USART1. Kolejnym krokiem po podłączeniu peryferium do wyprowadzeń, jest skonfigurowanie parametrów transferu. Na

[5]Implementacja testu dostępna jest w publicznym repozytorium `https://github.com/robgal519/performance_test_registers`.

[6]Ze względu na ograniczoną liczbę wyprowadzeń obudowy mikrokontrolera oraz wygodę przy projektowaniu płytki drukowanej, każde z wyprowadzeń ma kilka alternatywnych zastosowań.

początku włączona zostaje funkcjonalność do wysyłania danych, następnie *oversampling*[7] zostaje zredukowany do 8. Domyślnie *oversampling* jest dwukrotnie większy, jednak jak wskazuje dokumentacja, aby osiągnąć maksymalną prędkość, należy ustawić ten parametr tak jak na listingu. Kolejnym krokiem jest ustawienie dzielnika zegara, określając w ten sposób prędkość transmisji. Warte uwagi jest to, że jako prędkość zegara wpisana została tutaj wartość, która stanowi połowę zegara systemowego. Jest to wartość poprawna, ponieważ choć główny zegar systemowy działa z prędkością $168Mhz$, to zegar peryferium może wynieść maksymalnie połowę zegara systemowego, i taka wartość została ustalona podczas inicjalizacji zegara. Ostatnim krokiem konfiguracji jest uruchomienie peryferium oraz podpięcie przerwania.

Listing 14 przedstawia funkcję implementującą przerwanie USART mające za zadanie załadowanie kolejnego bajtu danych do rejestru peryferium oraz sygnalizację zakończenia transferu opartego o przerwanie. Po potwierdzeniu, że przerwanie dotyczy zakończenia transmisji bajtu danych, kontrolowana jest ilość danych do wysłania. W przypadku zakończenia transferu ustawiana jest stosowna flaga oraz wyłączone zostaje przerwanie. W sytuacji, kiedy dane nie zostały w pełni wysłane, kolejny bajt przesyłany jest do rejestru peryferium.

Do podstawowej implementacji brakuje jeszcze funkcji umożliwiającej rozpoczęcie transferu. Listing 15 przedstawia jakie kroki należy wykonać, aby wysłać dane, wykorzystując implementację opartą o przerwania USART. Po skopiowaniu wskaźnika na dane oraz rozmiaru do globalnych zmiennych, z których będzie korzystać przerwanie, pozostaje jedynie poinstruować USART, aby rozpoczął transfer. Odpowiedzialna jest za to instrukcja z linii 6.

W przypadku DMA, większość konfiguracji przebiega podobnie, jednak w tym przypadku należy dodatkowo skonfigurować strumień DMA. Listing 16 zawiera konfigurację DMA. W tym przypadku przerwanie USART jest zbędne, ponieważ to moduł DMA będzie odpowiedzialny za transfer kolejnych bajtów. Na końcu wspomnianego listingu konfigurowane jest jednak przerwanie do obsługi transferu DMA. Wywoła się ono po wysłaniu ostatniego bajtu danych, sygnalizując tym samym zakończenie pracy. Analizując przedstawiony listing, ponownie widzimy znajomy schemat działania. Pierwszym krokiem w próbie komunikacji z peryferium jest upewnienie się, że peryferium to otrzymuje sygnał zegarowy. W liniach $10 - 12$ ustalany jest stan urządzenia, tak aby zapewnić pełną kontrolę. Następnie wybierany jest kanał, na którym DMA będzie pracować. W linii 19 DMA zostaje poinstruowane, aby wystawiało przerwanie po zakończeniu transferu. W kolejnej instrukcji ustalany jest rozmiar pojedynczego transferu na $1B$, ponieważ przesyłamy dane bezpośrednio do rejestru DR peryferium USART1, które przyjmuje jedynie $1B$. Kolejno ustalany jest priorytet oraz tryb transferu. W tym przypadku transfer odbywa się z pamięci do peryferium, ponieważ źródło danych znajduje się w pamięci RAM, a celem jest rejestr USART. Ostatecznie uruchamiane jest przerwanie DMA.

Dla testu w wersji obsługującej transfer DMA przerwanie od USART jest zbędne, konieczne

[7] Co prawda istnieje polski odpowiednik tego terminu, czyli nadpróbkowanie jednak zdecydowano się pozostawić termin w oryginale dla zachowania zgodności z dokumentacją mikrokontrolera.

```
static uint16_t get_bbr_for_speed(uint32_t speed, uint32_t fck){
    // baud rate = fCK / (8 * (2 - OVER8) * USARTDIV)

    static const float over8 = 1.0f;
    float usartDiv = ((float)fck/(float)speed)/(8.0f *(2.0f-over8));

    uint32_t mantisa = (uint32_t)usartDiv;
    uint16_t fraction = (uint16_t)(16*(usartDiv-(float)mantisa));
    uint16_t BRR = (uint16_t)((mantisa << 4U ) + fraction);
    return BRR;
}
void configure_usart1(uint32_t baudrate){

// enable USART1 clock, bit 17 on APB1ENR
RCC->APB2ENR |= RCC_APB2ENR_USART1EN;

// enable GPIOA clock, bit 0 on AHB1ENR
RCC->AHB1ENR |= RCC_AHB1ENR_GPIOAEN;

// set pin modes as alternate mode 7
// USART1 TX and RX pins are PA9 and PA10 respectively
GPIOA->MODER &= ~(0xFU << (2*9)); // Reset bits  PA9 and PA10
GPIOA->MODER |=  (0xAU << (2*9)); // Set   bits  PA9 and PA10 to alternate mode (10)

// choose AF7 for USART2 in Alternate Function registers
GPIOA->AFR[1] |= (0x7 << (9-8)*4); // for pin A9
GPIOA->AFR[1] |= (0x7 << (10-8)*4); // for pin A10

// USART1 TX enable, TE bit 3
USART1->CR1 |= USART_CR1_TE;

USART1->CR1 |= USART_CR1_OVER8;

USART1->BRR = get_bbr_for_speed(baudrate,84000000);
// enable usart1 - UE, bit 13
USART1->CR1 |= USART_CR1_UE;

NVIC_SetPriority(USART1_IRQn, 1); // Priority level 1
NVIC_EnableIRQ(USART1_IRQn);

}
```

Listing 13: Implementacja konfiguracji USART

jest jednak skonfigurowanie przerwanie generowane z DMA, które ma za zadanie sygnalizację zakończenia transferu. Listing 17 przedstawia jego implementację. Po potwierdzeniu, że przyczyną wystąpienia przerwanie jest zakończenie transferu, flaga ta zostaje zdjęta, aby poinformować

```c
void USART1_IRQHandler(void)
{
    // check if the source is transmit interrupt
    if (USART1->SR & USART_SR_TXE_Msk) {

        if (next_byte_pos == data_size) {
            UART_TransferComplete = true;
            USART1->CR1 &= ~ USART_CR1_TXEIE;
        }
        else {
            // flush ot the next char in the buffer
            USART1->DR = data[next_byte_pos++];
        }
    }
}
```

Listing 14: Implementacja przerwania do obsługi USART1

```c
void start_transfer_usart1(uint8_t* buffer, uint32_t size){
    data = buffer;
    data_size = size;
    next_byte_pos = 0;
    UART_TransferComplete = false;
    USART1->CR1 |= USART_CR1_TXEIE ;
}
```

Listing 15: Implementacja funkcji do wysyłania danych

peryferium o poprawnym obsłużeniu zdarzenia, a następnie sygnał zakończenia pracy zostaje przekazany głównemu programowi poprzez ustawienie globalnej zmiennej.

Rozpoczęcie transferu danych z wykorzystaniem DMA przedstawione zostało na listingu 18. W tym przypadku wskaźnik na dane, ilość danych, oraz wskaźnik na miejsce docelowe zapisywany jest w rejestrach DMA, a ostatnia linia rozpoczyna transfer.

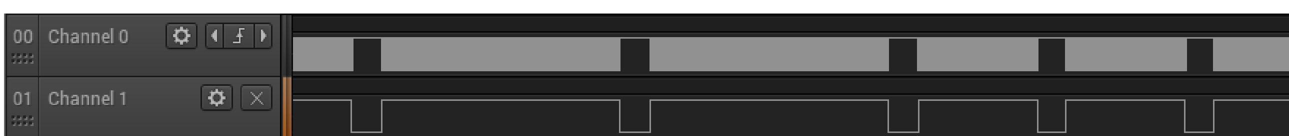
Przedstawiony zestaw funkcji jest wystarczający, aby z powodzeniem wysłać dane, jednak każdy driver powinien posiadać również funkcję, która spowoduje wyłączenie peryferium. W tym przypadku również stworzono taką funkcję. Jej implementacja ogranicza się jedynie do skasowania flagi w rejestrze peryferium odpowiedzialnej za jego uruchomienie. Dodatkowo wyłączone zostają przerwania związane ze skonfigurowanymi peryferiami poprzez wywołanie funkcji `NVIC_DisableIRQ();`.

Rysunek 4 przedstawia fragment sygnału przechwyconego przez analizator logiczny. Na kanale 0 nasłuchiwany był pin odpowiedzialny za transfer USART1, natomiast kanał 1 śledził pin $PA4$. Sygnalizował on czas spędzony na oczekiwaniu na zakończenie transferu. Na rysunku widać, moment zmiany prędkości transferu, który można rozpoznać po skróconym czasie trans-

```
...
USART1->BRR = get_bbr_for_speed(baudrate,84000000);
// enable usart1 - UE, bit 13
USART1->CR1 |= USART_CR1_UE;

USART1->SR &= ~USART_SR_TC;

RCC->AHB1ENR |= RCC_AHB1ENR_DMA2EN;
// clear control register
DMA2_Stream7->CR = 0;
// wait until DMA is disabled
while(DMA2_Stream7->CR & (1 << 0));


// channel select
DMA2_Stream7->CR |= (0x4 << 25); // channel4

// enable transfer complete interrupt
DMA2_Stream7->CR |= DMA_SxCR_TCIE; // (1 << 4);

// peripheral data size already 00 for byte
// memory increment mode
DMA2_Stream7->CR |= DMA_SxCR_MINC; // (1 << 10);

// Priority level
DMA2_Stream7->CR |= (0x2 << 16); // high - 10

// DIR bits should be 01 for memory-to-peripheral
//   source is SxM0AR, dest is SxPAR
DMA2_Stream7->CR |= (0x1 << 6);


NVIC_SetPriority(DMA2_Stream7_IRQn, 3); // Priority level 3
NVIC_EnableIRQ(DMA2_Stream7_IRQn);
```

Listing 16: Modyfikacja konfiguracji do wersji obsługującej DMA

feru. W każdym teście wysyłana jest taka sama porcja danych w wielkości $500B$. Można również zauważyć, że dana prędkość jest testowana kilkukrotnie, aby uśrednić błąd, który może powstać w trakcie badania. Wszystkie testy zostały powtórzone pięć razy.



Rysunek 4: Fragment testu zarejestrowany analizatorem logicznym

```c
void DMA2_Stream7_IRQHandler(void)
{
    // clear stream transfer complete interrupt
    if (DMA2->HISR & DMA_HISR_TCIF7) {
        // clear interrupt
        DMA2->HIFCR |= DMA_HISR_TCIF7;
        UART_TransferComplete = true;
    }
}
```

Listing 17: Przerwanie związane z zakończeniem transferu DMA

```c
void transfer_usart1_dma(uint8_t* data, uint32_t size){
    UART_TransferComplete = false;
    // source memory address
    DMA2_Stream7->M0AR = (uint32_t)data;
    // destination memory address
    DMA2_Stream7->PAR = (uint32_t)&(USART1->DR);
    // number of items to be transferred
    DMA2_Stream7->NDTR = size;
    // enable DMA
    DMA2_Stream7->CR |= DMA_SxCR_EN; // (1 << 0);
}
```

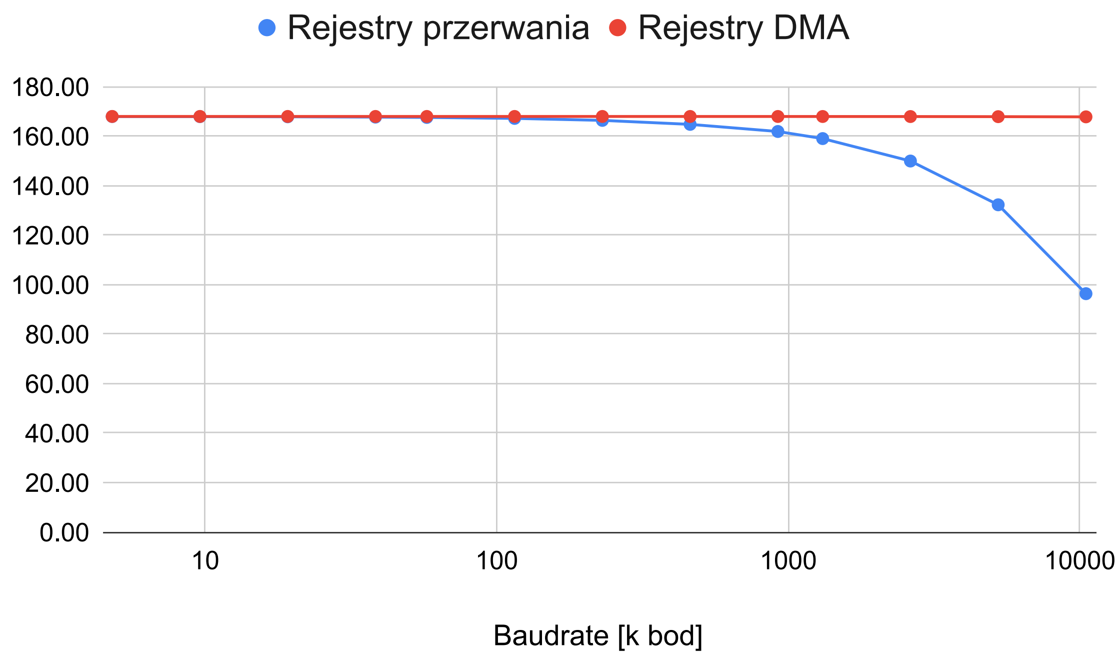
Listing 18: Funkcja rozpoczynająca transfer W wersji wykorzystującej DMA

Dla omawianej implementacji wartość współczynnika c ze wzoru 1 wyniosła 18. Tabela 1 oraz Rysunek 5 przedstawiają wyniki uzyskane w obu testach opartych o rejestry mikrokontrolera. Jak wyraźnie widać implementacja wykorzystująca DMA, nie wpływa na wydajność głównej pętli, co jest oczekiwanym wynikiem, ponieważ po skonfigurowaniu transferu, ten odbywa się automatycznie bez udziału mikrokontrolera, który dostaje jedynie sygnał o zakończeniu transmisji.

Implementacja oparta o przerwania pokazuje, w jaki sposób rosnąca ilość wywołań przerwania wpływa na ilość czasu procesora dostępną w czasie działania, i zgodnie z oczekiwaniami spada ona wraz ze zwiększaniem się prędkości transferu. W przypadku użytego mikrokontrolera prędkość $10.5 Mbps$ jest maksymalną dostępną wartością i przy tej prędkości testowana implementacja oparta o przerwania nadal pozostawiła głównemu programowi ok. 57% z wszystkich dostępnych cykli zegara.

Baudrate [k bod]	Rejestry przerwania	Rejestry DMA
$4,8$	$167,913$	$167,946$
$9,6$	$167,880$	$167,946$
$19,2$	$167,814$	$167,946$
$38,4$	$167,685$	$167,945$
$57,6$	$167,552$	$167,945$
$115,2$	$167,162$	$167,944$
$230,4$	$166,360$	$167,943$
$460,8$	$164,746$	$167,939$
$921,6$	$161,867$	$167,933$
$1312,5$	$159,000$	$167,926$
$2625,0$	$149,929$	$167,908$
$5250,0$	$132,239$	$167,863$
$10500,0$	$96,318$	$167,786$

Tablica 1: Estymowana liczba cykli zegara w milionach dostępna w głównym programie w $1s$



Rysunek 5: Estymowana ilość cykli zegara w milionach dostępna w głównym programie w $1s$

2.2.2 Implementacja oparta o STM-HAL[8]

STM-HAL, czyli sterownik dostarczony przez producenta pozwala stosunkowo wygodnie skonfigurować każdy parametr obsługi peryferium. Dzięki dodatkowemu oprogramowaniu do tworzenia projektów większość kodu służącego do konfiguracji jest generowana automatycznie na podstawie prostych wytycznych ustawionych przez programistę. W odróżnieniu od przedstawionej wcześniej implementacji opartej o rejestry, interfejs dostarczony przez producenta jest kompleksowym rozwiązaniem, które przewiduje możliwość dowolnej zmiany parametrów transferu.

Do zainicjalizowania peryferium STM-HAL dostarcza funkcję `HAL_UART_Init()`, która wymaga przygotowanej struktury `UART_HandleTypeDef` przedstawionej na listingu 19 jako argument. Struktura ta zawiera wskaźnik, pod którym z mapowane jest peryferium opisane jako `Instance`. Bufory na dane wejściowe oraz kolejka wyjściowa również zostały zebrane w strukturze wraz ze zmiennymi określającymi ich rozmiar, oraz licznik dotychczas wysłanych danych. Do działania sterownika przydatne są również obiekty zawierające konfigurację DMA oraz zmienne zapisujące aktualny stan sterownika. Wewnątrz bloku warunkowej kompilacji znajdują się wskaźniki na funkcje, które zostaną wywołana w odpowiedzi na wystąpienie danej akcji, takiej jak np. zakończenie transferu, wystąpienie błędu i inne. W trakcie testu makro `USE_HAL_UART_REGISTER_CALLBBACKS` zostało ustalone na wartość 0, a więc kod pod kompilacją warunkową nie był częścią struktury konfiguracyjnej. Możliwość zarejestrowania niskopoziomowych funkcji pomocniczych jest ukłonem ze strony twórców interfejsu do programistów, dla których możliwość reagowania na sygnały generowane przez sterownik poprzez zarejestrowanie odpowiedniej funkcji może być bardzo wygodnym rozwiązaniem. Do określania parametrów transferu wykorzystywane jest pole `Init`. Struktura definiująca to pole przedstawiona została na listingu 20. Zawiera ona kilka pól definiujących wszystkie istotne parametry transferu takie jak prędkość transmisji, długość słowa, wykorzystanie bitów stopu, bitu parzystości, trybu transferu, kontrola przepływu czy *oversampling*. Parametry te są przechowywane w postaci typu `uint32_t`, a dostępne wartości dostarczone są w postaci makr preprocesora, których kilka przykładów dołączono do listingu 20. Rozwiązanie to jest stosunkowo proste, a kod definiujący parametry transferu jest dość jasny i czytelny dzięki wykorzystaniu makr. Oczywiście, gdyby zastosowano typowane pola danych, pisanie kodu byłoby prostsze ze względu na możliwości współczesnych narzędzi developerskich, które potrafią dokładnie podpowiadać możliwe wartości, jednak w tym przypadku autor sterownika zdecydował się przechowywać wartości, jakie należy wpisać w rejestry peryferium, co z pewnością upraszcza implementację niskopoziomowych warstw.

Funkcja służąca do wysyłania wykorzystująca przerwania przedstawiona została na listingu 21. Implementacja dostarczona przez STM jest podobnie rozwiązana do analogicznej funk-

[8] Implementacja testu dostępna jest w publicznym repozytorium `https://github.com/robgal519/MT_uart1_HAL`.

```c
typedef struct __UART_HandleTypeDef
{
  USART_TypeDef               *Instance;        /*!< UART registers base address        */
  UART_InitTypeDef            Init;             /*!< UART communication parameters      */
  uint8_t                     *pTxBuffPtr;      /*!< Pointer to UART Tx transfer Buffer */
  uint16_t                    TxXferSize;       /*!< UART Tx Transfer size              */
  __IO uint16_t               TxXferCount;      /*!< UART Tx Transfer Counter           */
  uint8_t                     *pRxBuffPtr;      /*!< Pointer to UART Rx transfer Buffer */
  uint16_t                    RxXferSize;       /*!< UART Rx Transfer size              */
  __IO uint16_t               RxXferCount;      /*!< UART Rx Transfer Counter           */
  DMA_HandleTypeDef           *hdmatx;          /*!< UART Tx DMA Handle parameters      */
  DMA_HandleTypeDef           *hdmarx;          /*!< UART Rx DMA Handle parameters      */
  HAL_LockTypeDef             Lock;             /*!< Locking object                     */
  __IO HAL_UART_StateTypeDef  gState;           /*!< UART state Tx operations.
                                                     value of @ref HAL_UART_StateTypeDef */
  __IO HAL_UART_StateTypeDef  RxState;          /*!< UART state Rx operations.
                                                     value of @ref HAL_UART_StateTypeDef */
  __IO uint32_t               ErrorCode;        /*!< UART Error code                    */
#if (USE_HAL_UART_REGISTER_CALLBACKS == 1)
  void (* TxHalfCpltCallback)(struct __UART_HandleTypeDef *huart);
  void (* TxCpltCallback)(struct __UART_HandleTypeDef *huart);
  void (* RxHalfCpltCallback)(struct __UART_HandleTypeDef *huart);
  void (* RxCpltCallback)(struct __UART_HandleTypeDef *huart);
  void (* ErrorCallback)(struct __UART_HandleTypeDef *huart);
  void (* AbortCpltCallback)(struct __UART_HandleTypeDef *huart);
  void (* AbortTransmitCpltCallback)(struct __UART_HandleTypeDef *huart);
  void (* AbortReceiveCpltCallback)(struct __UART_HandleTypeDef *huart);
  void (* WakeupCallback)(struct __UART_HandleTypeDef *huart);
  void (* MspInitCallback)(struct __UART_HandleTypeDef *huart);
  void (* MspDeInitCallback)(struct __UART_HandleTypeDef *huart);
#endif  /* USE_HAL_UART_REGISTER_CALLBACKS */
} UART_HandleTypeDef;
```

Listing 19: Struktura do konfiguracji USART

cji przedstawionej w przypadku implementacji opartej o rejestry. Również w tym przypadku widzimy przypisanie wskaźnika na dane, zapisanie jego rozmiaru oraz licznika danych do wysłania. Następnie z pomocą makra ustawiana jest flaga w rejestrze peryferium rozpoczynająca transfer. Autorzy tej funkcji nie zapomnieli również o odpowiednich zabezpieczeniach poprzez wprowadzenie sekcji krytycznej na instrukcje modyfikujące peryferium. Implementacja funkcji wykorzystującej DMA do transferu również jest analogiczna i ogranicza się do przepisania przekazanych parametrów do wewnętrznych struktur oraz rozpoczęcia transferu.

Analizując kod dostarczony przez twórców sterownika, można odnieść wrażenie, że miejscami implementacje są wyjątkowo długie lub wręcz rozwlekłe. Jednak po chwili pracy, bardzo łatwo można dostrzec, że implementacje są logicznie rozłożone, a ilość kodu w funkcjach wynika

```c
typedef struct
{
  uint32_t BaudRate;
  uint32_t WordLength;
  uint32_t StopBits;
  uint32_t Parity;
  uint32_t Mode;
  uint32_t HwFlowCtl;
  uint32_t OverSampling;
} UART_InitTypeDef;

#define UART_WORDLENGTH_8B                    0x00000000U
#define UART_WORDLENGTH_9B                    ((uint32_t)USART_CR1_M)

#define UART_STOPBITS_1                       0x00000000U
#define UART_STOPBITS_2                       ((uint32_t)USART_CR2_STOP_1)

#define UART_PARITY_NONE                      0x00000000U
#define UART_PARITY_EVEN                      ((uint32_t)USART_CR1_PCE)
#define UART_PARITY_ODD                       ((uint32_t)(USART_CR1_PCE | USART_CR1_PS))

#define UART_HWCONTROL_NONE                   0x00000000U
#define UART_HWCONTROL_RTS                    ((uint32_t)USART_CR3_RTSE)
#define UART_HWCONTROL_CTS                    ((uint32_t)USART_CR3_CTSE)
#define UART_HWCONTROL_RTS_CTS                ((uint32_t)(USART_CR3_RTSE | USART_CR3_CTSE))

#define UART_MODE_RX                          ((uint32_t)USART_CR1_RE)
#define UART_MODE_TX                          ((uint32_t)USART_CR1_TE)
#define UART_MODE_TX_RX                       ((uint32_t)(USART_CR1_TE | USART_CR1_RE))

#define UART_STATE_DISABLE                    0x00000000U
#define UART_STATE_ENABLE                     ((uint32_t)USART_CR1_UE)

#define UART_OVERSAMPLING_16                  0x00000000U
#define UART_OVERSAMPLING_8                   ((uint32_t)USART_CR1_OVER8)
```

Listing 20: Struktura definiująca parametry transmisji UART

głównie z konieczności obsłużenia wielu możliwych konfiguracji. Należy więc zaznaczyć, że choć na pierwszy rzut oka kod dostarczony od STM jest skomplikowany, to po chwili pracy z nim można odnaleźć oczywiste schematy w jego budowie i standardowe praktyki w dostarczonych rozwiązaniach takie jak funkcje `callback` do sygnalizowania zdarzeń.

Przeglądając kod, znaczną uwagę zwraca implementacja przerwań, która wydaje się wyjątkowo długa, jednak wynika to z konieczności obsłużenia wszystkich sygnałów dostarczonych przez fizyczne urządzenie. Skutkuje to oczywiście funkcją o znacznej objętości kodu, jednak każda ze ścieżek w tej funkcji jest niezwykle krótka i ogranicza się jedynie do określenia

```c
HAL_StatusTypeDef HAL_UART_Transmit_IT(UART_HandleTypeDef *huart, uint8_t *pData, uint16_t Size)
{
  /* Check that a Tx process is not already ongoing */
  if (huart->gState == HAL_UART_STATE_READY)
  {
    if ((pData == NULL) || (Size == 0U))
    {
      return HAL_ERROR;
    }

    /* Process Locked */
    __HAL_LOCK(huart);

    huart->pTxBuffPtr = pData;
    huart->TxXferSize = Size;
    huart->TxXferCount = Size;

    huart->ErrorCode = HAL_UART_ERROR_NONE;
    huart->gState = HAL_UART_STATE_BUSY_TX;

    /* Process Unlocked */
    __HAL_UNLOCK(huart);

    /* Enable the UART Transmit data register empty Interrupt */
    __HAL_UART_ENABLE_IT(huart, UART_IT_TXE);

    return HAL_OK;
  }
  else
  {
    return HAL_BUSY;
  }
}
```

Listing 21: Implementacja funkcji do wysyłania danych wykorzystująca przerwania

przyczyny wystąpienia przerwania, następnie zawołania odpowiedniej funkcji, która ustali wewnętrzny stan sterownika, lub podejmie konkretną akcję. Na końcu ścieżki wywołania w przerwaniu można znaleźć również zawołanie do zarejestrowanej funkcji `callback` której implementacji sterownik będzie oczekiwał od programisty.

Ostatecznie ilość linii kodu przygotowana do obsługi peryferium jest duża, jednak kod ten jest dość dobrze rozdzielony na mniejsze funkcje obsługujące konkretne przypadki, a większość implementacji zajmuje określenie stanu peryferium oraz przyczyny wystąpienia przerwania.

Do obsługi testu zgodnie z przyjętą konwencją wymagana jest definicja struktury opisującej kontekst testowy. Ponieważ wszystkie testy realizowane są z wykorzystaniem tej samej struk-

tury konfiguracyjnej, konieczne było stworzenie kilku prostych funkcji, które połączą interfejs oczekiwany w teście, oraz dostarczony przez sterownik. W tym celu stworzony został specjalny plik, którego kluczowe elementy zostały przedstawione na listingu 22.

Funkcja `init_Hal_UART` ma za zadanie skonfigurowanie peryferium do pracy przy zadanej prędkości. Widać tutaj prezentowaną wcześniej strukturę, która pozwala na określenie wszystkich parametrów peryferium, a dzięki wykorzystaniu makr, kod staje się stosunkowo prosty do analizy. Po przygotowaniu wspomnianej struktury przekazywana jest ona do funkcji `HAL_UART_Init`, która odpowiada za przygotowanie sterownika do pracy.

Kolejną funkcją w prezentowanym pliku jest implementacja specjalnej funkcji, która będzie zawołana z przerwania, kiedy otrzymany zostanie sygnał zakończenia transferu. W tej sytuacji konieczne jest wstrzymanie pętli oczekującej w głównej funkcji testowej, a to uwarunkowane jest od stanu flagi `UART_TransferComplete`. Dla pewności przed zmianą stanu flagi sprawdzany jest obiekt sterownika, aby potwierdzić, że sygnał ten przyszedł z odpowiedniego urządzenia. Funkcja ta jest ciekawa, ponieważ twórcy sterownika nie mogą zmusić programistów do zaimplementowania tej funkcji, jeśli Ci nie widzą takiej potrzeby. Po przeszukaniu plików źródłowych sterownika można natknąć się na funkcję o tej samej nazwie, jednak o pustej implementacji. Funkcja ta stanowi domyślną implementację a dzięki atrybutowi preprocesora `__weak`, linker weźmie pod uwagę implementacje dostarczoną przez programistę, a zignoruję wydmuszkę dostarczoną wraz ze sterownikiem. Wadą tego rozwiązania jest fakt, że funkcja ta jest wspólna dla wszystkich modułów UART. Jako argument przyjmuje wskaźnik na obiekt UART, dzięki czemu możliwa jest jednoznaczna identyfikacja, który obiekt raportuje zakończenie transferu, jednak w przypadku wykorzystania więcej niż jednego modułu UART w projekcie, dobra implementacja tego elementu jest kluczowa. Jest to potencjalne źródło znacznych opóźnień, ponieważ kod ten wywołuje się w przestrzeni przerwań. Należy więc dołożyć wszelkich starań, aby implementacja tej funkcji była możliwie zwięzła.

Pozostałe funkcje w tym pliku odpowiadają za rozpoczęcie transferu oraz deinicjalizację obiektu. Funkcje te jedynie przekazują parametry do funkcji dostarczonych przez STM, i służą jedynie do połączenia sterownika z interfejsem wymaganym przez test.

W trakcie procesu kalibracji współczynnik został ustalony na wartość 18, a zebrane dane po normalizacji zostały przedstawione w tabeli 2 oraz w postaci wykresu 6

Z prezentowanych danych można odczytać, że tym razem implementacja oparta o transfer DMA wykazuje nieznaczne obciążenie systemu przy bardzo dużych prędkościach, jednak nadal można uznać, że jest to dla większości zastosowań niezauważalne obciążenie. Najbardziej interesujący w tym zestawie danych jest test oparty o przerwania przy maksymalnej prędkości. Podczas tego testu, analizator logiczny zarejestrował poprawny transfer danych, jednak licznik w pętli oczekującej nie zwiększył swojej wartości. Fakt ten wskazuje, że mikrokontroler działał na granicy swoich możliwości, poświęcając cały swój czas na obsługę przerwań i stąd też wartość 0 w ostatnim wierszu tablicy 2.

```c
extern volatile bool UART_TransferComplete;
static UART_HandleTypeDef huart1;

void init_Hal_UART(void **data, uint32_t baudrate) {

  huart1.Instance = USART1;
  huart1.Init.BaudRate = baudrate;
  huart1.Init.WordLength = UART_WORDLENGTH_8B;
  huart1.Init.StopBits = UART_STOPBITS_1;
  huart1.Init.Parity = UART_PARITY_NONE;
  huart1.Init.Mode = UART_MODE_TX_RX;
  huart1.Init.HwFlowCtl = UART_HWCONTROL_NONE;
  huart1.Init.OverSampling = UART_OVERSAMPLING_8;
  HAL_UART_Init(&huart1);
  *data = &huart1;
}

void HAL_UART_TxCpltCallback(UART_HandleTypeDef *huart) {
  if (huart == &huart1)
    UART_TransferComplete = true;
}

bool transfer_IT_Hal_UART(void *internal, uint8_t *data, uint16_t size) {
  return HAL_UART_Transmit_IT(internal, data, size) == HAL_OK;
}

bool transfer_DMA_Hal_UART(void *internal, uint8_t *data, uint16_t size) {
  return HAL_UART_Transmit_DMA(internal, data, size) == HAL_OK;
}
bool deinit_Hal_UART(void *internal) {
  return HAL_UART_DeInit(internal) == HAL_OK;
}
```

Listing 22: Adapter umożliwiający podłączenie implementacji dostarczonej przez STM do testu.

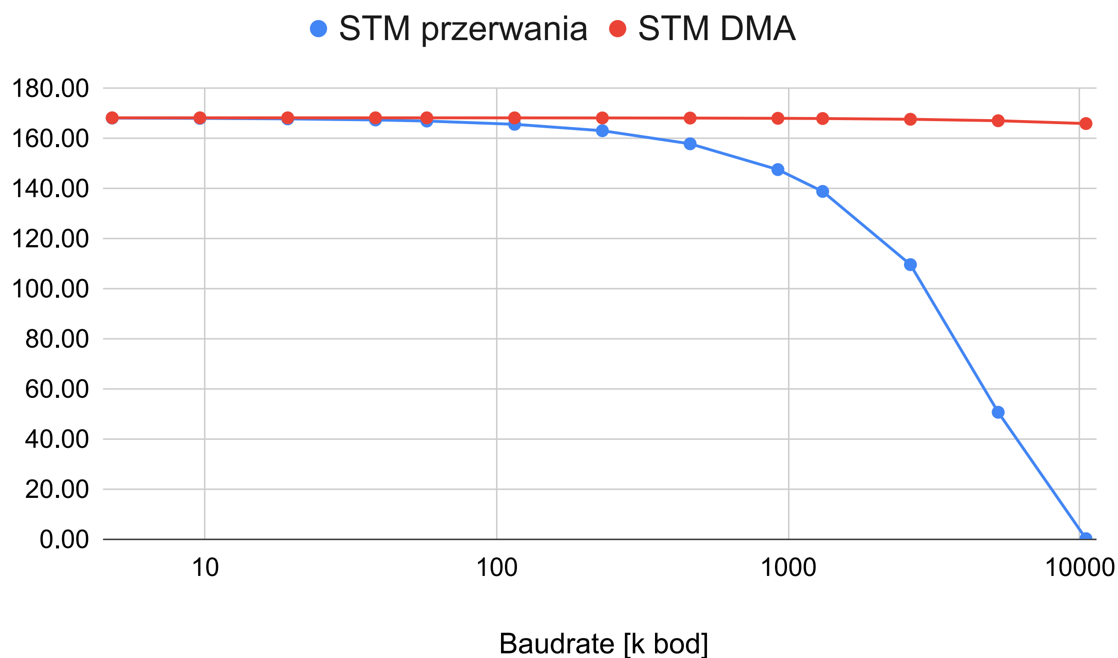
2.2.3 Implementacja oparta o CMSIS-Driver[9]

Twórcy interfejsu CMSIS-Driver w ciekawy sposób podeszli do problemu konfiguracji peryferium. Na początku w pliku `RTE_Device.h`, należy uruchomić peryferium, następnie można wybrać parametry definiujące jego podstawowe funkcje, takie jak numer pinu wyjściowego dla nadajnika i odbiornika czy wykorzystanie DMA do transferu danych. Modyfikacja pliku `RTE_Dvevice.h` wprost jest stosunkowo niewygodna. Pomimo opisowych nazw makr i logicznie rozłożonej struktury pliku, znalezienie odpowiedniego makra do modyfikacji może stanowić wyzwanie. Zdecydowaną wadą jest brak podziału pomiędzy parametrami przeznaczonymi do modyfikacji przez

[9]Implementacja testu dostępna jest w publicznym repozytorium `https://github.com/robgal519/MT_uart1_CMSIS`.

Baudrate [k bod]	STM przerwania	STM DMA
$4,8$	$167,839$	$167,945$
$9,6$	$167,731$	$167,944$
$19,2$	$167,517$	$167,942$
$38,4$	$167,085$	$167,938$
$57,6$	$166,652$	$167,933$
$115,2$	$165,355$	$167,920$
$230,4$	$162,800$	$167,894$
$460,8$	$157,576$	$167,844$
$921,6$	$147,280$	$167,742$
$1312,5$	$138,568$	$167,656$
$2625,0$	$109,350$	$167,362$
$5250,0$	$50,444$	$166,779$
$10500,0$	0	$165,650$

Tablica 2: Estymowana liczba cykli zegara w milionach dostępna w głównym programie w $1s$ dla sterownika od STM



Rysunek 6: Estymowana ilość cykli zegara w milionach dostępna w głównym programie w $1s$ dla sterownika od STM

programistę a flagami, które powstają w wyniku wybranych ustawień, które są konieczne do poprawnego przygotowania sterownika. Gdyby umieścić makra, które są przeznaczone do modyfikacji przez programistę na początku pliku, zdecydowanie zwiększono by jego czytelność.

Fragment pliku RTE_Device.h został zaprezentowany na listingu 23. Twórcy interfejsu przewidują, że programista zmodyfikuje wartość jedynie makr z linii 3 oraz 6, czyli RTE_USART1 odpowiedzialnego za uruchomienie tego peryferium, oraz RTE_USART1_TX_ID, które odpowiada za wybór wyprowadzenia dla pinu wyjściowego peryferium. Linie od 7 do 23 stanowią więc informację nieistotną dla programisty, która swobodnie mogłaby zostać umiejscowiona na końcu pliku. Jak widać, plik ten dość niefortunnie miesza elementy, które powinny być modyfikowane, z elementami, które zostały wygenerowane, i nie powinny być zmieniane przez programistę. Twórcy CMSIS najwyraźniej uznali, że plik ten należy traktować jako czarną skrzynkę, i zalecają obsługę zapisanych tam wartości przez edytor Keil μVision. Edytor ten potrafi przedstawić w przyjaznej formie konfigurację projektu, co redukuje prawdopodobieństwo błędu, oraz ułatwia dokonywanie zmian dzięki prezentacji konfiguracji w postaci graficznej.

```
1  // <e> USART1 (Universal synchronous asynchronous receiver transmitter) [Driver_USART1]
2  // <i> Configuration settings for Driver_USART1 in component ::CMSIS Driver:USART
3  #define RTE_USART1                       1
4  
5  //   <o> USART1_TX Pin <0=>Not Used <1=>PA9 <2=>PA15 <3=>PB6
6  #define RTE_USART1_TX_ID                 1
7  #if     (RTE_USART1_TX_ID == 0)
8  #define RTE_USART1_TX                    0
9  #elif   (RTE_USART1_TX_ID == 1)
10 #define RTE_USART1_TX                    1
11 #define RTE_USART1_TX_PORT               GPIOA
12 #define RTE_USART1_TX_BIT                9
13 #elif   (RTE_USART1_TX_ID == 2)
14 #define RTE_USART1_TX                    1
15 #define RTE_USART1_TX_PORT               GPIOA
16 #define RTE_USART1_TX_BIT                15
17 #elif   (RTE_USART1_TX_ID == 3)
18 #define RTE_USART1_TX                    1
19 #define RTE_USART1_TX_PORT               GPIOB
20 #define RTE_USART1_TX_BIT                6
21 #else
22 #error "Invalid USART1_TX Pin Configuration!"
23 #endif
```

Listing 23: Fragemnt pliku RTE_Device.h przedstawiający konfigurację UART1

Do użycia sterownika CMSIS konieczne jest uzyskanie dostępu do specjalnego statycznego obiektu, który jest stworzony na podstawie konfiguracji z pliku RTE_Device.h. Obiekt ten można osiągnąć jedynie z pomocą słowa kluczowego extern, ponieważ jest on tworzony w pliku z implementacją odpowiedniego sterownika, a autorzy interfejsu nie przewidują żadnej innej metody dostępu do tego elementu. Wydaje się, że takie podejście jest ścieżką na skróty, ze strony autorów interfejsu, ponieważ na podstawie wartości makr w RTE_Device.h można byłoby stworzyć

obiekt o typie enumerowanym (`enum`), który zawierałby jedynie wartości odpowiadające skonfigurowanym peryferiom. Pozwoliłoby to na przygotowanie funkcji, która w elegancki sposób udostępniłaby odpowiedni obiekt na podstawie wartości takiego typu przekazanego jako parametr. Jest to oczywiście jedna z wielu możliwości na eleganckie uzyskanie dostępu do obiektu reprezentującego peryferium.

Obiekt uzyskany ze sterownika ma podobną strukturę dla każdego peryferium. Zawiera on wskaźniki na funkcje, które mają za zadanie jego kontrolę, jak również takie, które odpowiadają logicznym funkcjom wykonywanym przez peryferium.

Pełna inicjalizacja peryferium przed rozpoczęciem pracy jest rozdzielona na kilka elementów, które można zaobserwować na listingu 24. Na początku konieczne jest wywołanie funkcji `Initialize`, która ma za zadanie wstępne uruchomienie sterownika poprzez wyzerowanie wszelkich jego wewnętrznych flag, oraz przypisanie głównego `callback` dostarczonego z argumentem. `Callback` dostarczony przy inicjalizacji, jest standardowym punktem wyjścia dla wszelkich zdarzeń sygnalizowanych przez sterownik. Implementacja tej funkcji nie jest wymagana przez sterownik. Równie poprawne byłoby przekazanie w tym miejscu wskaźnika `NULL`. Nie jest to jednak optymalne rozwiązanie, ponieważ w takim przypadku monitorowanie stanu peryferium ogranicza się jedynie do *pollingu*, czyli regularnego odpytywania sterownika o jego stan.

Kolejną funkcją w procesie konfiguracji jest `PowerControl`. Funkcja ta odpowiada za uruchomienie lub wyłączenie peryferium. Teoretycznie obsługiwane są 3 wartości argumentu dla tej funkcji:

- `ARM_POWER_OFF`,
- `ARM_POWER_FULL`,
- `ARM_POWER_LOW`,

W praktyce, po wczytaniu się w implementację tej funkcji dla przykładowego mikrokontrolera funkcja ta dla wartości `ARM_POWER_LOW`, zwraca status informujący, że nie wspiera tej funkcjonalności. Wartość `ARM_POWER_OFF` ma spowodować wyłączenie peryferium, więc wykonuje odpowiednie kroki, aby przerwać wszelkie transfery, które mogą być w trakcie wykonania, wyrejestrować przerwania od peryferiów i wykonać deinicjalizację peryferium.

W przypadku przedstawionym na listingu przekazano wartość `ARM_POWER_FULL`, która po sprawdzeniu, że urządzenie zostało zainicjalizowane komendą z poprzedniej linii, zeruje wewnętrzne zmienne odpowiedzialne za stan transferu, oraz uruchamia wszystkie konieczne zegary do peryferiów koniecznych do działania USART1.

Kolejne linie z listingu 24 są ciekawe, ponieważ przedstawiają wielokrotne wywołania funkcji `Control`. Jest to niewątpliwie ciekawa funkcja, ponieważ przyjmuje dwa argumenty, których interpretacja uzależniona jest od ich wartości. W linii 6 można zauważyć, że pierwszy argument został skonstruowany na podstawie makr, które tworzą maski bitowe reprezentujące poszczególne tryby pracy peryferium, a drugi argument mówi o prędkości transferu. Wykorzystanie

```
extern ARM_DRIVER_USART Driver_USART1;

void init_usart1(void **internal, uint32_t baudrate) {
  Driver_USART1.Initialize(UART_eventHandler);
  Driver_USART1.PowerControl(ARM_POWER_FULL);
  Driver_USART1.Control(ARM_USART_MODE_ASYNCHRONOUS | ARM_USART_DATA_BITS_8 |
                      ARM_USART_PARITY_NONE | ARM_USART_STOP_BITS_1 |
                      ARM_USART_FLOW_CONTROL_NONE,
                baudrate);
  Driver_USART1.Control(ARM_USART_CONTROL_TX, 1);
  Driver_USART1.Control(ARM_USART_CONTROL_RX, 1);
  *internal = (void *)&Driver_USART1;
}
```

Listing 24: Funkcja pomocnicza do inicjalizacji USART.

nazwanych pól bitowych jest czytelnym rozwiązaniem, choć konfiguracja prędkości transferu poprzez drugi argument jest mało intuicyjna. Patrząc dalej na linie 10 oraz 11, widać ponowne wywołanie funkcji `Control`, która z kolei przyjmuje bardziej konwencjonalną formę, gdzie jako pierwszy argument przedstawiony zostaje parametr, który zostanie ustawiony, a drugi argument stanowi wartość, jaką przyjmie wywołany argument. Z punktu widzenia użytkownika jest to zdecydowanie ciekawe rozwiązanie, które można uznać za czytelne i proste w zrozumieniu. Początkowo niezbędna jest jednak dokumentacja i analiza przykładu, ponieważ ustawienie prędkości transmisji, jest trochę nieoczekiwane, jednak można zauważyć, że w wywołaniu z linii 6 konfigurowane są parametry transmisji, natomiast w kolejnych wywołaniach funkcji `Control`, kontrolowane są inne aspekty peryferium.

Jak można się spodziewać, implementacja takiego interfejsu oparta jest o parsowanie pierwszego argumentu i odpowiednią interpretację drugiego. W praktyce odbywa się to z pomocą ogromnej instrukcji `switch`, która miejscami zawiera dodatkowe zagnieżdżone instrukcje `switch`. Tego typu implementacja zdecydowanie zebrałaby ogromną ilość komentarzy podczas *code review*[10], gdyby była częścią komercyjnego produktu. Może się wydawać interesujące, jaką faktycznie funkcję pełnią wywołania z linii $10 - 11$, uruchomienie wyprowadzeń odbywało się przecież z pomocą makr w pliku `RTE_Device.h`. Analiza kodu ukazuje, że w zależności od wartości z pliku `RTE_Device.h` tworzone są statyczne obiekty reprezentujące wyprowadzenia peryferium, które następnie są przekazywane do kolejnych statycznych obiektów opisujących peryferium. Oznacza to więc, że informacje zapisane w `RTE_Device.h` przekazywane są do sterownika, jednak nie powodują one automatycznie ich konfiguracji. Jest to ciekawe rozwiązanie, ponieważ oznacza, że dla danej kompilacji nie można zmieniać wyprowadzeń (ponieważ te ustawiane są przez makra preprocesora), ale można sterować czy dane wyprowadzenie będzie urucho-

[10]Code review to inaczej przegląd kodu stosowany często przy tworzeniu oprogramowania, kiedy to programiści spoza zespołu twórców oceniają jakość kodu i starają się wskazać dogi jego poprawy.

mione. Pozwala to między innymi wybrać odpowiednie wyprowadzenia dla USART, a następnie w czasie działania programu wybrać czy potrzebny jest tylko pin `TX`, czy może konieczne są oba. Ostatecznie więc to funkcja `Configure` odpowiada za uruchomienie odpowiedniej funkcji na wyprowadzeniach `TX` oraz `RX`. Jak widać kompetencje funkcji `Configure`, są niezwykle duże, od wyznaczania parametrów transferu, do uruchomienia wyprowadzeń. Oczywiście na odpowiednim poziomie abstrakcji taka funkcja mogłaby znaleźć swoje miejsce, jednak analizując implementację dla testowanego mikrokontrolera, trzeba stwierdzić, że funkcja ta jest ogromna i przeprowadza programistę, który podejmie się jej analizy przez istny *roller coaster* po różnych abstrakcjach i zawiłościach w implementacji.

Ostatnia ciekawa funkcja konieczna do wykonania testu odpowiada za wysłanie danych. Interfejs CMSIS-Driver przewiduje, że dane będą wysyłane z pomocą funkcji `Send` dostępnej z obiektu `Driver_USART1`. Funkcja ta z punktu widzenia użytkownika nie zawiera żadnych niespodzianek. Klasycznie przyjmuje ona wskaźnik na dane do wysłania oraz zmienną przedstawiającą jak dużo bajtów należy przesłać. Analizując implementację tej funkcji zauważymy jednak, że transfer z wykorzystaniem DMA odbywa się z użyciem dokładnie tej samej implementacji. To która metoda transferu zostanie zrealizowana uzależnione jest od wartości makr `__USART_DMA_TX`, które z kolei ustalane jest w przypadku gdy makro `RTE_USART1_TX_DMA` z pliku `RTE_Device.h` ma wartość 1.

Przeglądając plik `USART_STM32F4xx.c` mozna odnieść wrażenie, że kod ten został napisany w tylko jednym celu, aby spełnił swoją podstawową funkcję. Twórcy z pewnością nie przejmowali się czytelnością tworzonego rozwiązania. Wstawki kompilacji warunkowej sprawiają wrażenie, jakby były dopisywane w miarę potrzeb. Kod ten niewątpliwie zasługuje na fundamentalny refaktoring. Fakt ten sprawia, że rozwiązanie to nie będzie się nadawać do pewnych zastosowań.

Na szczęście z punktu widzenia testu, a co za tym idzie programisty korzystającego z interfejsu, przygotowanie struktury konfiguracyjnej sprowadza się do stworzenia wydmuszek funkcji, które wołają metody z obiektu USART. Implementacje te przedstawiono na listingu 25.

Ogromną zaletą rozwiązania od STM, była możliwość wyboru trybu transferu w czasie działania. Interfejs CMSIS-Driver ogranicza nas do transferu metodą wybraną w pliku `RTE_Device.h`. Oznacza to, że aby zmienić transfer z DMA na transfer obsługiwany przerwaniami, konieczne jest wyłączenie DMA dla USART w tym pliku oraz ponowna kompilacja projektu. Rozwiązanie to powoduje, że nie ulega zmianie implementacja kodu opartego o CMSIS-Driver, jednak w kodzie nie ma żadnej informacji o wykorzystanym trybie. Nie trudno sobie wyobrazić taką aplikację, która w czasie działania zmienia zastosowanie dla danego wyprowadzenia, co może nieść za sobą konieczność transferu znacznych ilości danych. W takim przypadku należy od samego początku zadeklarować czy będzie wykorzystywane DMA i nie można tej decyzji odłożyć do momentu użycia. W niektórych zastosowaniach taki wzorzec stanowi wadę.

W trakcie kalibracji współczynnik do normalizacji ponownie wyniósł 18. Dane uzyskane

```c
extern ARM_DRIVER_USART Driver_USART1;
extern volatile bool UART_TransferComplete;

void UART_eventHandler(uint32_t event) {
  if (event & ARM_USART_EVENT_SEND_COMPLETE)
    UART_TransferComplete = true;
}

void init_usart1(void **internal, uint32_t baudrate) {
  Driver_USART1.Initialize(UART_eventHandler);
  Driver_USART1.PowerControl(ARM_POWER_FULL);
  Driver_USART1.Control(ARM_USART_MODE_ASYNCHRONOUS | ARM_USART_DATA_BITS_8 |
                        ARM_USART_PARITY_NONE | ARM_USART_STOP_BITS_1 |
                        ARM_USART_FLOW_CONTROL_NONE,
                baudrate);
  Driver_USART1.Control(ARM_USART_CONTROL_TX, 1);
  Driver_USART1.Control(ARM_USART_CONTROL_RX, 1);
  *internal = (void *)&Driver_USART1;
}

bool test_send(void *internal, uint8_t *data, uint16_t size) {
  if (internal != &Driver_USART1)
    return false;
  Driver_USART1.Send(data, size);
  return true;
}

bool test_uninitialize(void *internal) {
  if (internal != &Driver_USART1)
    return false;
  Driver_USART1.Uninitialize();
  return true;
}
```

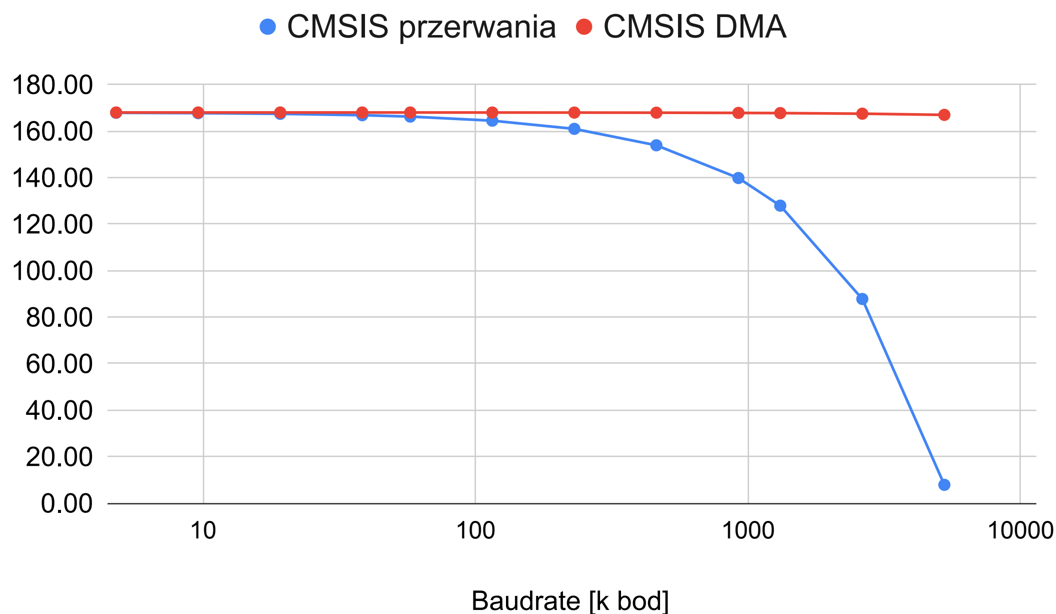
Listing 25: Kontekst testowy dla testu CMSIS-Driver.

w trakcie obu testów zostały zebrane w tabeli 3 oraz wykresie 7.

Uzyskane wyniki przedstawiają podobną charakterystykę do tej uzyskanej w testach poprzednich sterowników. Istotną różnicę stanowi fakt, że nie udało się osiągnąć transferu z maksymalną prędkością wspieraną przez mikrokontroler. Fakt ten jest związany z parametrem *oversampling*, który wymaga modyfikacji, aby osiągnąć maksymalną prędkość pracy, a interfejs CMSIS nie przewiduje jego konfiguracji. Tłumaczy to wartości *ND* w ostatnim wierszu tabeli 3. Analizując wartości w kolumnie reprezentującej przerwania, można zauważyć, że od prędkości ok. 230kbit, obciążenie zaczyna być zauważalne i każda wyższa prędkość powoduje eksponencjalne zwiększenie obciążenia. Należy zauważyć, że podążając widocznym trendem,

Baudrate [k bod]	CMSIS przerwania	CMSIS DMA
4, 8	167, 799	167, 944
9, 6	167, 652	167, 943
19, 2	167, 358	167, 942
38, 4	166, 771	167, 938
57, 6	166, 181	167, 934
115, 2	164, 416	167, 923
230, 4	160, 899	167, 898
460, 8	153, 832	167, 852
921, 6	139, 759	167, 758
1312, 5	127, 875	167, 678
2625, 0	87, 751	167, 419
5250, 0	7, 831	166, 913
10500, 0	ND	ND

Tablica 3: Estymowana liczba cykli zegara w milionach dostępna w głównym programie w 1s dla sterownika CMSIS



Rysunek 7: Estymowana ilość cykli zegara w milionach dostępna w głównym programie w 1s dla drivera CMSIS

najprawdopodobniej dla prędkości 10.5Mbit, obciążenie wyniosłoby ponad 100% czasu procesora, co oznacza, że transfer przy takiej prędkości najprawdopodobniej nie byłby skuteczny.

Test wykonany z wykorzystaniem DMA prezentuje spodziewany wynik, w którym, obciążenie głównego programu jest minimalne i w praktyce niezależne od prędkości.

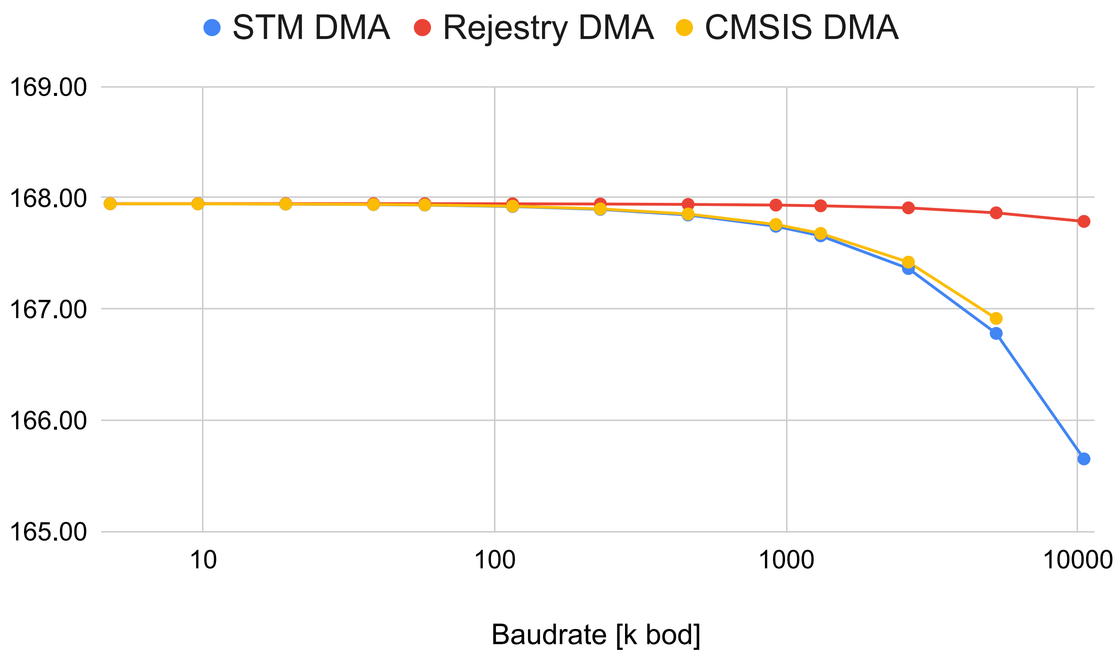
2.3 Porównanie wyników

Zebrane dane zostały przedstawione na wykresach 8 oraz 9, co daje możliwość łatwego porównania testowanych rozwiązań. Analizując przedstawione wykresy, należy stwierdzić, że oba gotowe rozwiązania, znacznie odstają od teoretycznego maksimum wydajności przy wyższych prędkościach transferu. Dla niezwykle popularnej prędkości 115200 dodatkowe obciążenie jest niezauważalne. Przy tym rzędzie prędkości nie ma znaczenia, które rozwiązanie zostanie wykorzystane. Różnice widać dopiero przy wyższych prędkościach, gdzie lepiej spisuje się interfejs STM, oferując pełną prędkość deklarowaną przez producenta, oraz gwarantując lepszą wydajność CMSIS. Jest to jednak wynik, którego można było się spodziewać. Uniwersalność kodu, jaką proponuje rozwiązanie od CMSIS, niesie za sobą koszt, wynikający z faktu, że rozwiązanie kompatybilne z wieloma urządzeniami nie będzie w pełni zoptymalizowane na żadnym z nich.

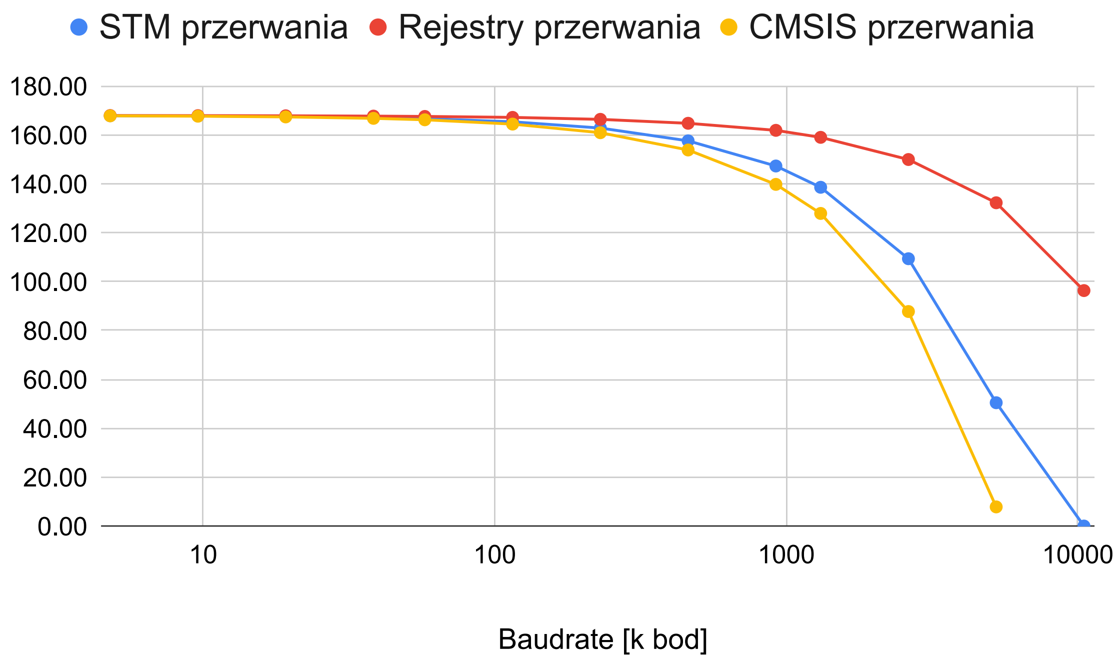
Wyniki mogą wydawać się jednak niepokojące, ponieważ wskazują, że dla prymitywnej implementacji opartej o przerwania, nawet w najgorszym możliwym przypadku nadal ponad połowa czasu procesora była dostępna, a rozwiązanie od STM przy tych samych parametrach transferu wykorzystało całe zasoby procesora. Oczywiście wskazuje to na możliwości dalszego rozwoju interfejsu, jednak w praktycznym zastosowaniu różnica ta nie jest aż tak istotna. Chcąc wykorzystać tak duże prędkości znacznie sensowniejszym rozwiązaniem jest oparcie transferu o DMA, co daje już zadowalające rezultaty niezależnie od wybranego interfejsu.

Brak możliwości ustawienia maksymalnej prędkości w rozwiązaniu od CMSIS stanowi pewne utrudnienie, jednak dokumentacja jasno wskazuje, jak należy ustawić rejestry USART, aby wysyłać z prędkością $10.5Mbps$. Modyfikacja implementacji oczywiście w takiej sytuacji jest możliwa, jednak dość uciążliwa.

Prezentowane wyniki zawierają porównanie testów wykonanych jedynie dla bloku USART i choć nie testowano innych peryferiów, to można założyć, że wyniki byłyby podobne. Stwierdzenie to bazuje na fakcie, że sterowniki oparte o interfejs CMSIS-Driver wyglądają niezwykle podobnie od strony użytkownika oraz ze względu na przyjęte rozwiązania. Podczas testu udało się wykryć, że interfejs CMSIS nie obsługuje maksymalnej prędkości, jednak jest to związane z koniecznością wsparcia szerokiego zakresu urządzeń przez ten interfejs i tego typu kompromis jest akceptowalny. Dodatkowym elementem, na który należy zwrócić uwagę przy ocenie interfejsu, to fakt, że wykorzystano ogólnie dostępną implementację do testu. Oczywiście przy poświęceniu odpowiedniej ilości czasu i zasobów możliwe byłoby stworzenie sterownika kompatybilnego z interfejsem CMSIS-Driver, który prezentowałby lepsza jakość kodu, a jego wydajność przesunęła, by się w stronę rozwiązania opartego o rejestry. Do testu wykorzystano jednak gotową implementację, ponieważ fakt istnienia dostępnej implementacji stanowi jeden z kluczowych zalet sterownika. Programista stojący przed wyzwaniem stworzenia sterownika od podstaw, najprawdopodobniej będzie się kierował wygodą użytkowania, a interfejs CMSIS-Driver, choć posiada ciekawe rozwiązania, to fundamentalnie blokuje część funkcjonalności takich jak wybór metody transferu w czasie działania programu, lub choćby rekonfiguracja wyprowadzeń.



Rysunek 8: Porównanie rozwiązań wykorzystujących DMA.



Rysunek 9: Porównanie rozwiązań wykorzystujących przerwania.

3 Przykładowa aplikacji oparta o CMSIS-Driver

Jedną z podstawowych zalet interfejsu CMSIS-Driver jest niezależność od wykorzystanego mikrokontrolera. Aby sprawdzić tę funkcjonalność, stworzono projekt z wykorzystaniem płytki deweloperskiej *STM32F407VET6*, który następnie przeniesiono na platformę Open1768, której punktem centralnym jest mikrokontroler *LPC1768*. Mikrokontrolery te wybrano, ponieważ są to popularne jednostki wykorzystywane w płytkach deweloperskich, co sprawia, że są dość tanie i łatwo dostępne. Zawierają podobny zestaw peryferiów, a przy tym reprezentują różne rodziny. *LPC1768* jest oparte na CortexM3, natomiast jednostka STM na CortexM4. Stworzona aplikacja ma na celu odpowiedzieć na pytanie, czy interfejs CMSIS-Driver w praktyce spełnia wymagania konieczne do stworzenia praktycznego projektu, oraz zebrać doświadczenia z przenoszenia gotowego projektu na zupełnie inną platformę.

Oczywiste jest, że konieczne będą jakieś zmiany w projekcie, aby ten działał na innej platformie, jednak CMSIS (potencjalnie) oferuje, kuszącą możliwość, przeniesienia kodu użytkownika bez modyfikacji. Nadal konieczne będzie przygotowanie odpowiedniego skryptu linkera, systemu budowania, dostarczenie niskopoziomowych implementacji oraz ponowna konfiguracja CMSIS-Driver. Są to jednak operacje, które częściowo mogą być wykonane automatycznie, lub ograniczają się jedynie do odzwierciedlenia w konfiguracji fizycznych połączeń wykonanych na płytce deweloperskiej.

3.1 Struktura oraz funkcjonalności projektu[11]

Podstawową funkcjonalnością projektu jest monitorowanie temperatury, ciśnienia oraz wilgotności za pomocą czujnika BME280 [8]. Dostęp do tych wartości odbywa się poprzez Ethernet. Peryferium to w systemach wbudowanych zazwyczaj jest podzielone na dwie części: Ethernet MAC jest częścią mikrokontrolera, do którego dołączane jest zewnętrzne urządzenie Ethernet PHY. To właśnie Ethernet PHY ma za zadanie fizyczny transport danych, ale dane te są przygotowywane w mikrokontrolerze i wysyłane standardowym protokołem RMII (*eng. reduced media-independent interface*). Taka architektura pozwala podłączyć do mikrokontrolera różne fizyczne media do obsługi transferu Ethernet. Najprostszą metodą jest moduł Ethernet PHY, który zakończony jest wtyczką RJ45, a transfer odbywa się kablem UTP (*eng. Unshielded Twisted Pair*). Ta sama fizyczna warstwa wykorzystywana jest w sieciach Ethernet, co sprawia, że gotowy projekt można łatwo wpiąć do domowej sieci LAN (*eng. Local Area Network*).

Projekt na pierwszy rzut oka wydaje się prosty, jednak wymaga ciągłej akwizycji danych oraz implementacji serwera, który obsłuży zapytania z sieci Ethernet. Pozornie istnieją więc jedynie dwa kluczowe zadania, ale konieczność obsługi sieci sprawia, że implementacja wymaga użycia systemu czasu rzeczywistego. Wykorzystanie CMSIS-Driver do implementacji komuni-

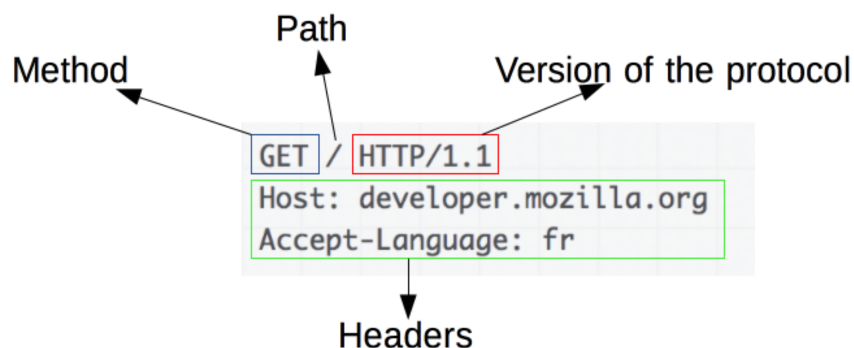
[11] Implementacja projektu dostępna jest w publicznym repozytorium `https://github.com/robgal519/stm_www`

kacji z peryferiami mikrokontrolera, będzie praktycznym testem użyteczności tego interfejsu, oraz pozwoli przetestować możliwość przeniesienia implementacji na inną platformę.

Implementacja serwera jest dość prosta i opiera się na analizie nagłówka zapytania HTTP (*eng. Hypertext Transport Protocol.*), jakie zostało odebrane przez serwer. Zapytanie to składa się z kilku elementów przedstawionych na rys 10. Na początku każdego zapytania HTTP znajduje się tzw.metoda. Rozróżniamy następujące metody HTTP:

- GET — klient oczekuje na dane,
- POST — w zapytaniu HTTP znajdują się dane dla serwera,
- UPDATE — klient wysyła dane do zaktualizowania na serwerze,
- DELETE — klient prosi o usunięcie zasobu,
- PUT — pełni podobną funkcję do POST.

Serwer implementowany w ramach projektu obsługuje jedynie zapytania GET, ale w bardzo łatwy sposób można dostosować go do obsługi innych typów zapytań. Kolejnym elementem w nagłówku HTTP jest ścieżka, na którą zapytanie to jest kierowane. Z punktu widzenia serwera, nie ma znaczenia czy w systemie plików istnieje taka ścieżka. Implementacja wykorzystana w projekcie nie korzysta nawet z systemu plików. Istotne jest jednak to, czy serwer rozpoznaje tę ścieżkę jako zarejestrowany element. Wersja protokołu jest ostatnim elementem w pierwszej linii wiadomości. Serwer ignoruje te informacje, jak również nagłówki znajdujące się w kolejnych liniach.



Rysunek 10: Elementy składające się na nagłówek zapytania HTTP.

Do obsługi zapytań HTTP, wykorzystana została biblioteka LWIP (*eng. Lightweight IP stack*). Jest to lekka implementacja stosu sieciowego przeznaczona do zastosowania w mikrokontrolerach [3]. Obsługuje ona dwa tryby pracy. Pierwszy z nich operuje na tzw. *pooling*, czyli wymaga, aby funkcja do obsługi zdarzeń była regularnie wołana w trakcie działania programu, natomiast w drugim trybie stos sieciowy funkcjonuje pod kontrolą systemu czasu rzeczywistego.

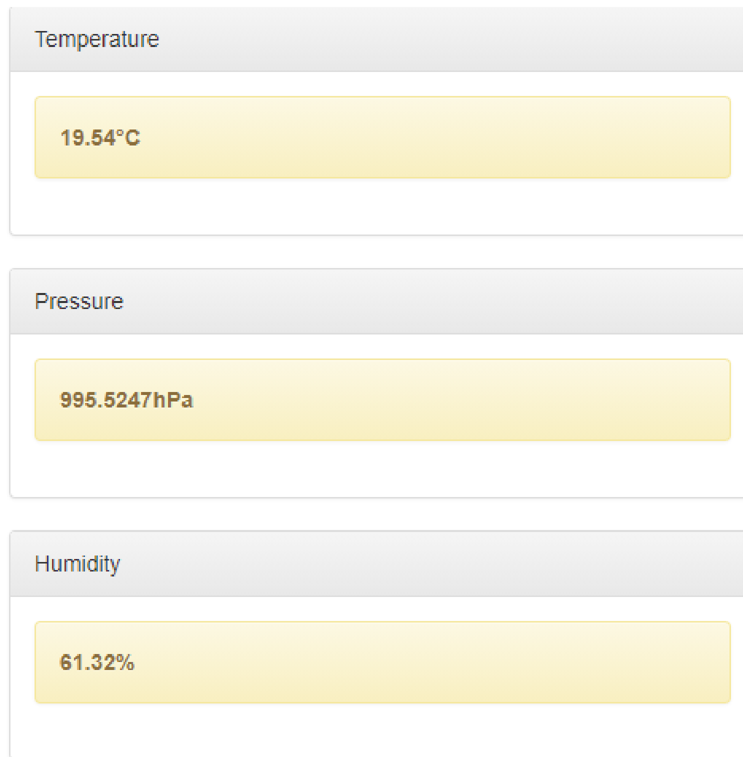
W prezentowanym projekcie wykorzystany został FreeRTOS z nakładką od CMSIS (CMSIS-RTOS) do realizacji systemu czasu rzeczywistego i, co za tym idzie, biblioteka LWIP pracuje w drugim ze wspomnianych trybów.

Aplikacja składa się zaledwie z 3 zadań. Pierwszym z nich jest zadanie odpowiedzialne za obsługę czujnika BME280. Implementacja każdego zadania w FreeRTOS wygląda podobnie i składa się z instrukcji, które inicjalizują wszelkie konieczne elementy, a następnie w pętli nieskończonej wykonywana jest główne zadanie. W przypadku zadania obsługi czujnika, w części inicjalizacyjnej konfigurowane jest I2C z CMSIS-Driver, i z jego pomocą konfigurowany jest sterownik. Następnie w pętli głównej zadania dokonywany jest odczyt wartości z czujnika po którym, zadanie jest usypiane na $500ms$ poprzez wywołanie funkcji `osDelay`. Funkcja ta powoduje, że uśpienie nie marnuje zasobów, ale informuje system czasu rzeczywistego, że przez najbliższe $500ms$ może zostać wykonane inne zadanie (potencjalnie o niższym priorytecie), lub jeśli żadne inne zadanie nie oczekuje na czas procesora, system czasu rzeczywistego wejdzie w specjalne zadanie, które uśpi mikrokontroler do czasu, aż pojawi się żądanie zasobów.

Kolejne zadanie jest tworzone przez stos sieciowy. Nie widać go bezpośrednio w kodzie, ponieważ powstaje ono w procesie inicjalizacji stosu LWIP, jednak zadanie to jest odpowiedzialne za obsługę przepływu danych pomiędzy fizyczną warstwą przysyłającą ramki Ethernet oraz stosem LWIP.

Ostatnim zadaniem wartym uwagi, jest obsługa serwera HTTP. Działanie tego serwera opiera się o działanie stosu LWIP. Nasłuchuje ono wszelkich połączeń na porcie 80. Następnie na podstawie ścieżki zapisanej w zapytaniu, wywołuje jedną z zarejestrowanych funkcji, która wyśle odpowiedź. Poza obsługą zapytań HTTP aplikacja reaguje również na zwykłe zapytanie od przeglądarki internetowej, w odpowiedzi wysyłając statyczną stronę HTML z niewielkim skryptem JavaScript, który w regularnych odstępach dokonuje zapytań do serwera o dane z czujnika. Dzięki temu po podłączeniu urządzenia do sieci LAN, możliwa jest interakcja z aplikacją z pomocą przeglądarki internetowej. Aby otworzyć aplikację w przeglądarce internetowej, w miejsce URL (*ang. Uniform Resource Locator*) należy podać adres IP mikrokontrolera. Aplikacja posiada statyczny adres IP (10.0.0.200) z sieci lokalnej. Odczytanie danych w ten sposób jest więc dość wygodne.

Stos TCP/IP to właściwie tylko pewien zestaw struktur przechowujących dane, których fizyczny transfer może odbywać się różnymi metodami. Najbardziej popularne w systemach wbudowanych to sieć GSM, protokół SLIP, który wykorzystuje UART do transferu pakietów, moduł Wi-Fi najczęściej połączony z mikrokontrolerem poprzez SPI lub rozwiązanie wykorzystujące wbudowany w mikrokontroler Ethernet MAC wraz z zewnętrznym Ethernet PHY. W projekcie wykorzystana została ostatnia z wymienionych możliwości, ponieważ pozostałe albo ograniczają znacząco możliwości transferu (UART i SPI) nie stanowiąc żadnego wyzwania, albo mocno komplikują projekt, nie wpływając na szybkość transmisji danych. Dodatkowym argumentem za wyborem Ethernet PHY, jest możliwość uruchomienia transmisji w standardzie



Rysunek 11: Serwis WWW działający na mikrokontrolerze.

Fast Ethernet. W niektórych projektach transfer tego rodzaju może być konieczny, więc wykorzystanie tego typu warstwy fizycznej do transferu ramek Ethernet jest ciekawą możliwością.

Na rysunku 11 widać stronę www wysłaną przez serwer działający na mikrokontrolerze. Za aktualizację danych odpowiada krótki skrypt dołączony do strony. Przedstawiony jest on na listingu 26. Funkcja ta wysyła zapytania GET na adres serwera, który rozpoznaje, jakie dane ma wysłać na podstawie ścieżki wysyłanej w zapytaniu. Aby odczytać aktualną temperaturę, wysyłane jest więc zapytanie na ścieżkę `temperature`, analogicznie pobierana jest wartość ciśnienia oraz wilgotności odpytując ścieżkę `pressure` oraz `humidity`. Przeglądarka komputera wykonująca ten skrypt zna adres mikrokontrolera, ponieważ to z tego adresu początkowo została pobrana wyświetlana strona. Po odczytaniu odpowiedzi od mikrokontrolera ta jest rozpakowywana, a następnie umieszczana w odpowiednim komponencie na wyświetlanej stronie. Wartości te są aktualizowane co 500ms.

3.2 Implementacja komunikacji z wykorzystaniem CMSIS-Driver

Praca programisty częściowo polega na umiejętności wyszukiwania gotowych rozwiązań, ponieważ wywarzanie otwartych drzwi to po prostu marnowanie czasu, który w wielu projektach jest najcenniejszym zasobem. W prezentowanym projekcie wykorzystano więc gotową bibliotekę do obsługi czujnika BMP280 dostarczoną przez producenta[12]. Sterownik ten stanowi bardzo

[12] `https://github.com/BoschSensortec/BMP280_driver`

```
$(document).ready(function () {

function refreshValues() {
  $.get("temperature", function (res) {
    $("#BME-temp").find("b").html(res.temperature + res.unit);
  });
  $.get("pressure", function (res) {
    $("#BME-press").find("b").html(res.pressure + res.unit);
  });
  $.get("humidity", function (res) {
    $("#BME-hum").find("b").html(res.humidity + res.unit);
  });
  setTimeout(refreshValues, 500);
}
setTimeout(refreshValues, 500);
});
```

Listing 26: Skrypt odpowiedzialny za aktualizację danych w aplikacji WWW.

dobry przykład, jak powinno się tworzyć tego typu oprogramowanie. Jedną z jego podstawowych cech jest generyczne podejście do sposobu komunikacji. Autorzy nie mogli sobie pozwolić na narzucenie konkretnego HAL'a, ponieważ w ten sposób sztucznie ograniczyliby możliwość wykorzystania modułu w niektórych projektach. Ogólność przyjętego rozwiązania polega na oddelegowaniu niskopoziomowej komunikacji do zaimplementowania przez programistę. Takie podejście dostarcza także jasny i wyraźnie rozdzielony interfejs. Pozwala to np. na wykonanie testów jednostkowych aplikacji, dla których implementacja tych funkcji jest doskonałym miejscem na wprowadzenie sztucznych funkcji symulujących prawdziwe peryferium.

Jest to zalecana forma tworzenia sterowników do peryferiów zewnętrznych z punktu widzenia mikrokontrolera, ponieważ dzięki temu kod sterownika nie jest w żaden sposób związany z żadną konkretną platformą [7]. Migracja tak stworzonego sterownika do innego projektu wymaga więc jedynie dostarczenia implementacji kilku fundamentalnych funkcji wymaganych przez sterownik. W prezentowanym projekcie w tym miejscu dostarczone zostały oczywiście implementacje oparte o CMSIS-Driver, co oznacza, że przy migracji kod ten nie powinien ulec zmianie.

Na listingu 27 przedstawiona została struktura konfiguracyjna czujnika BME280 wykorzystywana w projekcie. Zawiera ona 3 wskaźniki na funkcje, które są konieczne do interakcji z fizycznym urządzeniem. Funkcja `read` ma za zadanie odczytanie danych, `write` wysyła polecenie do czujnika, a funkcja `delay_ms` odpowiada za wstrzymanie wykonania programu przez wyznaczony czas wyrażony w milisekundach. Pierwsze dwie funkcje wydają się oczywiste, w jakiś sposób sterownik musi porozumiewać się z miernikiem. Funkcja opóźniająca wykorzystywana jest w trakcie procesu inicjalizacji, gdzie po resecie urządzenia należy odczekać określoną w do-

kumentacji ilość czasu, aby wysłać kolejne komendy.

Jeśli zwrócimy uwagę na funkcje odpowiedzialne za transfer danych, zauważymy, że nigdzie w kodzie nie znajdziemy wzmianki, że protokołem komunikacyjnym jest I2C. Fakt ten wynika z tego, że urządzenie BME280, posiada dwa interfejsy I2C, oraz SPI, do komunikowania się z mikrokontrolerem. Ponadto, sterownik nie wnika w sposób transferu, zakładając jedynie, że funkcje te są blokujące. Rozwiązanie to pozwala na wykorzystanie dowolnego interfejsu, zarówno w wersji wykorzystującej przerwania do obsługi transferu, jak i DMA. Jedynym wymaganiem jest, aby funkcje przekazane do sterownika były blokujące. Można to osiągnąć na wiele sposobów. Najprostszym jest wejście w pętlę oczekującą zaraz po rozpoczęciu transferu, której warunek końcowy uzależniony jest od statusu tego transferu. Ponieważ takie rozwiązanie jest wyjątkowo proste, a wydajność nie była głównym celem prezentowanego projektu, taką właśnie metodę wykorzystano do stworzenia blokujących transferów. Znacznie bardziej eleganckim rozwiązaniem byłoby zarejestrowanie funkcji callback na transfer, która zwolniłaby semafor, co pozwoliłoby wznowić działanie w danym wątku, i nie blokowałoby systemu czasu rzeczywistego podczas oczekiwania, jednak zdecydowano się na prostsze w implementacji oraz analizie rozwiązanie w postaci pętli oczekującej.

Poza tymi kluczowymi funkcjami struktura `bme280_dev` przechowuje takie parametry jak adres urządzenia oraz konfigurację procesu zbierania danych. Czujnik ten posiada 4 tryby dokładności pomiaru dla każdego z mierzonych parametrów. Zwiększanie dokładności pomiaru wykonywane jest kosztem czasu, jaki jest potrzebny do jego wykonania. Ponadto, dokładność pomiaru niektórych parametrów (np. ciśnienia i temperatury) jest ze sobą ściśle związana [8]. Dokumentacja czujnika opisuje jaki wpływ na uzyskiwany rezultat ma dobór odpowiednich ustawień oraz sugeruje konkretne ustawienia w zależności od popularnych zastosowań. Implementacje tych funkcji wykorzystane w projekcie zostały przedstawione na listingu 28. Obiekt `communication` jest statyczną strukturą reprezentującą komunikację I2C w CMSIS-Driver. Przed użyciem sterownika BME280, obiekt `communication` został wcześniej zainicjalizowany poza kontekstem sterownika.

Funkcja inicjalizacyjna biblioteki LWIP przedstawiona została na listingu 29. Na początku definiowany jest adres IP, maska podsieci, oraz brama domyślna, następnie inicjalizowany jest stos TCP/IP, z wykorzystaniem funkcji bibliotecznej `tcpip_init`. Funkcja ta została przygotowana pod współpracę z systemem czasu rzeczywistego i przygotowuje ona odpowiednie zadanie systemowe do obsługi stosu TCP/IP. Następnie przygotowane wartości dla adresów zdefiniowanych na początku funkcji zapisywane są w zmiennych z wykorzystaniem makra `IP4_ADDR`, które zostaną przekazane jako argument do funkcji, która tworzy nowy interfejs sieciowy. Funkcja `netif_add` ma za zadanie stworzenie nowego interfejsu sieciowego, dzięki któremu biblioteka LWIP, będzie mogła komunikować się z siecią. Poza przygotowanymi wcześniej wartościami wymaga ona wskaźników na funkcje do inicjalizacji oraz do odczytywania danych z warstwy fizycznej.

```c
/*!
 * @brief Type definitions
 */
typedef int8_t (*bme280_com_fptr_t)(uint8_t dev_id, uint8_t reg_addr, uint8_t *data, uint16_t len);
typedef void (*bme280_delay_fptr_t)(uint32_t period);

struct bme280_dev
{
    /*! Chip Id */
    uint8_t chip_id;

    /*! Device Id */
    uint8_t dev_id;

    /*! SPI/I2C interface */
    enum bme280_intf intf;

    /*! Read function pointer */
    bme280_com_fptr_t read;

    /*! Write function pointer */
    bme280_com_fptr_t write;

    /*! Delay function pointer */
    bme280_delay_fptr_t delay_ms;

    /*! Trim data */
    struct bme280_calib_data calib_data;

    /*! Sensor settings */
    struct bme280_settings settings;
};
```

Listing 27: Struktura konfiguracyjna sterownika BME280.

Ostatnim krokiem inicjalizacji, po poprawnym dodaniu interfejsu sieciowego, jest ustawienie domyślnego interfejsu, oraz uruchomienie go co wykonują funkcje odpowiednio `netif_set_default` i `netif_set_up`.

Implementacja funkcji wymaganych do stworzenia nowego interfejsu zebrano w osobnym pliku `ehternetif.c`, jednak tworzenie tego pliku od podstaw może przyczynić się do wprowadzenia niepotrzebnych błędów, oraz wymaga dobrej znajomości zarówno programowanego peryferium, jak i stosu LWIP. Z pomocą przychodzi generator projektów od STMicroelectronics CubeMX oraz przykładowy projekt wygenerowany w edytorze μVision. Po analizie projektu wygenerowanego przez CubeMX zauważono, że część funkcji można bezpośrednio przenieść do tworzonego rozwiązania, ponieważ korzystają one jedynie z funkcji dostarczonych wraz ze

```c
static void user_delay_ms(uint32_t period) { osDelay(period); }

static int8_t user_i2c_read(uint8_t dev_id, uint8_t reg_addr, uint8_t *reg_data,
                           uint16_t len) {
  int8_t rslt = 0; /* Return 0 for Success, non-zero for failure */

  communication->MasterTransmit(dev_id, &reg_addr, 1, false);
  wait_for_transmision();
  communication->MasterReceive(dev_id, reg_data, len, false);
  wait_for_transmision();

  return rslt;
}
static int8_t user_i2c_write(uint8_t dev_id, uint8_t reg_addr,
                            uint8_t *reg_data, uint16_t len) {

  int8_t rslt = 0; /* Return 0 for Success, non-zero for failure */

  uint8_t buff[64];
  buff[0] = reg_addr;
  memcpy(buff + 1, reg_data, len);
  communication->MasterTransmit(dev_id, buff, len + 1, false);
  wait_for_transmision();
  return rslt;
}
```

Listing 28: Implementacja funkcji wymaganych przez sterownik BME280 z wykorzystaniem CMSIS-Driver.

stosem LWIP i tworzą logiczną warstwę pośrednią pomiędzy niskopoziomową implementacją transmisji, oraz wysokopoziomowymi wymaganiami nałożonymi na funkcje interfejsu LWIP. Niskopoziomowe funkcje, które należy dostosować do implementacji opartej o CMSIS-Driver wyróżniają się przedrostkiem `low_level_`, co znacznie ułatwiło ich identyfikację, i uprościło analizę wygenerowanego kodu. Oryginalnie posiadają one implementację opartą o sterownik od STM, który został zastąpiony CMSIS-Driver. W tym procesie pomocna była analiza kodu od STM, jak również szablonowy projekt wygenerowany w μVision, który korzystał z CMSIS-Driver do obsługi Ethernet.

Pierwszą z kluczowych funkcji pomocniczych jest `low_level_init`, przedstawiona została na listingach 31 oraz 32. Na początku funkcji do struktury opisującej interfejs sieciowy zapisywane są podstawowe informacje takie jak adres MAC modułu Ethernet, MTU (*ang. Maximum Transfer Unit*) oraz odpowiednie flagi wskazujące działanie interfejsu. W drugiej części funkcji zaimplementowana została inicjalizacja sterownika CMSIS-Driver do obsługi peryferium Ethernet MAC oraz Ethernet PHY. Struktura `eth` stanowi wewnętrzny kontekst dla pliku `ethernetif.c`

```
void MX_LWIP_Init(void)
{
  /* IP addresses initialization */
  IP_ADDRESS[0] = 10;
  IP_ADDRESS[1] = 0;
  IP_ADDRESS[2] = 0;
  IP_ADDRESS[3] = 200;
  NETMASK_ADDRESS[0] = 255;
  NETMASK_ADDRESS[1] = 255;
  NETMASK_ADDRESS[2] = 0;
  NETMASK_ADDRESS[3] = 0;
  GATEWAY_ADDRESS[0] = 0;
  GATEWAY_ADDRESS[1] = 0;
  GATEWAY_ADDRESS[2] = 0;
  GATEWAY_ADDRESS[3] = 0;
  /* Initilialize the LwIP stack with RTOS */
  tcpip_init( NULL, NULL );
  /* IP addresses initialization without DHCP (IPv4) */
  IP4_ADDR( &ipaddr,
            IP_ADDRESS[0], IP_ADDRESS[1], IP_ADDRESS[2], IP_ADDRESS[3]);
  IP4_ADDR( &netmask,
            NETMASK_ADDRESS[0], NETMASK_ADDRESS[1], NETMASK_ADDRESS[2], NETMASK_ADDRESS[3]);
  IP4_ADDR( &gw,
            GATEWAY_ADDRESS[0], GATEWAY_ADDRESS[1], GATEWAY_ADDRESS[2], GATEWAY_ADDRESS[3]);
  /* add the network interface (IPv4/IPv6) with RTOS */
  netif_add(&gnetif, &ipaddr, &netmask, &gw, NULL, &ethernetif_init, &tcpip_input);
  /* Registers the default network interface */
  netif_set_default(&gnetif);
  netif_set_up(&gnetif);
}
```

Listing 29: Inicjalizacja stosu sieciowego LWIP.

i jest ona dostarczana do tej funkcji poprzez pole `state` z argumentu wywołania funkcji. Pole to, zdefiniowane jest wewnątrz stosu LWIP jako wskaźnik na typ `void`. W ten sposób twórcy LWIP umożliwiają przekazanie niskopoziomowego kontekstu pomiędzy różnymi wywołaniami niskopoziomowych funkcji implementowanych przez programistę. W prezentowanym przypadku kontekst ten stanowi struktura `ethernetif`, która została przedstawiona na listingu 30. Kontekst ten jest bardzo minimalistyczny i poza wskaźnikami na obiekty z CMSIS-Driver zawiera flagę sygnalizującą ramkę do odczytu, pole reprezentujące czy konfiguracja przebiegła poprawnie, oraz semafor służący do zabezpieczenia krytycznych sekcji kodu. Oczywiście nie jest to jedyna forma, jaką mogła przybrać przedstawiona struktura. Jest ona uzależniona od tego co programista chce przekazać pomiędzy wywołaniami funkcji niskopoziomowych, a co za tym idzie od implementacji tych funkcji. Kod przedstawiony na listingu 32, stanowiący kontynuację funkcji `low_level_init` inicjalizuje sterowniki CMSIS-Driver. Na początku zapisywane są obiekty

do obsługi peryferiów. Następnie otwierany jest semafor, aby wejść do sekcji krytycznej. Jako pierwszy konfiguracji podlega Ethernet MAC. Konfiguracja ta odbywa się w klasyczny sposób tak jak każdy inny sterownik z CMSIS-Driver. Funkcja `Initialize` przyjmuje wskaźnik na `callback` które będzie wołane z każdym zdarzeniem, następnie uruchamiane jest peryferium, i konfigurowany jest adres MAC.

Do inicjalizacji Ethernet PHY, wymagane jest przekazanie funkcji do zapisu oraz odczytu dostarczanych przez Ethernet MAC. Jeśli operacja ta się powiedzie, można przystąpić, do dalszej konfiguracji, gdzie uruchamiane jest urządzenie, ustawiany jest interfejs komunikacji pomiędzy Ethernet MAC, oraz wybierany jest tryb pracy.

Funkcja odpowiedzialna za obsługę zdarzeń od Ethernet, przekazana przy inicjalizacji Ethernet MAC, również została zamieszczona na wspomnianym listingu. Jej minimalistyczna implementacja ogranicza się jedynie do sygnalizacji ramki oczekującej na odbiór poprzez ustawienie flagi w kontekście.

```
1  struct ethernetif {
2    ARM_DRIVER_ETH_MAC *mac;              // Registered MAC driver
3    ARM_DRIVER_ETH_PHY *phy;              // Registered PHY driver
4    ARM_ETH_LINK_STATE link;              // Ethernet Link State
5    bool               event_rx_frame;   // Callback RX event generated
6    bool               phy_ok;           // PHY initialized successfully
7    bool               rx_event;         // Frame received
8    sys_sem_t          sem;
9  };
```

Listing 30: Niskopoziomowy kontekst do obsługi Ethernet.

Funkcja `low_level_output` przedstawiona na listingu 33 odpowiada za poprawne wysłanie danych, wykorzystując CMSIS-Driver. Początkowo implementacja może wydawać się trywialna, jednak należy tutaj zwrócić uwagę, w jaki sposób LWIP przekazuje dane. Bufor danych to tak naprawdę jednokierunkowa lista buforów i konieczne jest wysłanie wszystkich elementów w pętli. Również w tym przypadku implementacja została dostosowana z przykładowego projektu, co znacznie ułatwiło uruchomienie projektu. Odebranie danych realizowane w funkcji `low_level_input` realizowane jest analogicznie, ale w przeciwnym kierunku, czyli odczytane dane ze sterownika CMSIS-Driver, należy zapakować do struktury oczekiwanej przez LWIP.

Niewielkim modyfikacjom poddano również inne funkcje z analizowanego modułu, tak aby dostosować implementację pod CMSIS-Driver, jednak były to zmiany wynikające z różnic w interfejsie CMSIS-Driver oraz STM HAL, który był podstawą wygenerowanego przykładowego projektu.

```
static void low_level_init(struct netif *netif)
{
  struct ethernetif *eth = netif->state;
  ARM_ETH_MAC_CAPABILITIES cap;

  /* set MAC hardware address length */
  netif->hwaddr_len = ETH_HWADDR_LEN;
  /* set MAC hardware address */
  netif->hwaddr[0] =  0x00;
  netif->hwaddr[1] =  0x80;
  netif->hwaddr[2] =  0xE1;
  netif->hwaddr[3] =  0x00;
  netif->hwaddr[4] =  0x00;
  netif->hwaddr[5] =  0x00;

  /* maximum transfer unit */
  netif->mtu = 1500;

  /* device capabilities */
  /* don't set NETIF_FLAG_ETHARP if this device is not an ethernet one */
  netif->flags = NETIF_FLAG_BROADCAST | NETIF_FLAG_ETHARP | NETIF_FLAG_ETHERNET;
#if LWIP_IPV4 && LWIP_IGMP
  netif->flags |= NETIF_FLAG_IGMP;
#endif
#if LWIP_IPV6 && LWIP_IPV6_MLD
  netif->flags |= NETIF_FLAG_MLD6;
  /*
   * For hardware/netifs that implement MAC filtering.
   * All-nodes link-local is handled by default, so we must let the hardware know
   * to allow multicast packets in.
   * Should set mld_mac_filter previously. */
  if (netif->mld_mac_filter != NULL) {
    ip6_addr_t ip6_allnodes_ll;
    ip6_addr_set_allnodes_linklocal(&ip6_allnodes_ll);
    netif->mld_mac_filter(netif, &ip6_allnodes_ll, NETIF_ADD_MAC_FILTER);
  }
#endif /* LWIP_IPV6 && LWIP_IPV6_MLD */
```

Listing 31: Inicjalizacja protokołu sieciowego dla stosu LWIP.

```
38    /* Do whatever else is needed to initialize interface. */
39    eth->phy = &ARM_Driver_ETH_PHY_(ETH_DRV_NUM);
40    eth->mac = &ARM_Driver_ETH_MAC_(ETH_DRV_NUM);
41
42    eth->link   = ARM_ETH_LINK_DOWN;
43    eth->phy_ok = false;
44
45    sys_sem_new (&eth->sem, 0);
46    /* Get MAC capabilities */
47    cap = eth->mac->GetCapabilities ();
48    eth->event_rx_frame = (cap.event_rx_frame) ? true : false;
49
50    eth->mac->Initialize (eth0_notify);
51    eth->mac->PowerControl (ARM_POWER_FULL);
52    eth->mac->SetMacAddress ((ARM_ETH_MAC_ADDR *)netif->hwaddr);
53    eth->mac->Control (ARM_ETH_MAC_CONTROL_TX, 0);
54    eth->mac->Control (ARM_ETH_MAC_CONTROL_RX, 0);
55
56    /* Initialize Physical Media Interface */
57    if (eth->phy->Initialize (eth->mac->PHY_Read, eth->mac->PHY_Write) == ARM_DRIVER_OK) {
58      eth->phy->PowerControl (ARM_POWER_FULL);
59      eth->phy->SetInterface (cap.media_interface);
60      eth->phy->SetMode (ARM_ETH_PHY_AUTO_NEGOTIATE);
61      eth->phy_ok = true;
62    }
63    sys_sem_signal (&eth->sem);
64
65    static void eth0_notify (uint32_t event) {
66    /* Send notification on RX event */
67    if (event == ARM_ETH_MAC_EVENT_RX_FRAME) {
68      eth0_status.rx_event = true;
69    }
70  }
71  }
```

Listing 32: Kontynuacja inicjalizacji Ethernet MAC oraz Ethernet PHY z wykorzystaniem CMSIS-Driver dla stosu LWIP.

```c
static err_t
low_level_output(struct netif *netif, struct pbuf *p)
{
  struct ethernetif *eth = netif->state;
  struct pbuf *q;

#if ETH_PAD_SIZE
  pbuf_remove_header(p, ETH_PAD_SIZE); /* drop the padding word */
#endif

  sys_sem_wait (&eth->sem);
  for (q = p; q != NULL; q = q->next) {
    /* Send the data from the pbuf to the interface, one pbuf at a
       time. The size of the data in each pbuf is kept in the ->len
       variable. */
    u32_t flags = (q->next) ? ARM_ETH_MAC_TX_FRAME_FRAGMENT : 0;
    eth->mac->SendFrame (q->payload, q->len, flags);
  }
  sys_sem_signal (&eth->sem);

  MIB2_STATS_NETIF_ADD(netif, ifoutoctets, p->tot_len);
  if (((u8_t *)p->payload)[0] & 1) {
    /* broadcast or multicast packet*/
    MIB2_STATS_NETIF_INC(netif, ifoutnucastpkts);
  } else {
    /* unicast packet */
    MIB2_STATS_NETIF_INC(netif, ifoutucastpkts);
  }
  /* increase ifoutdiscards or ifouterrors on error */

#if ETH_PAD_SIZE
  pbuf_add_header(p, ETH_PAD_SIZE); /* reclaim the padding word */
#endif

  LINK_STATS_INC(link.xmit);

  return ERR_OK;
}
```

Listing 33: Implementacja funkcji odpowiedzialnej za wysłanie danych.

3.3 Przenoszenie implementacji z STM na LPC [13]

Zasadniczą różnicą charakteryzującą opisywane w tej pracy platformy developerskie jest dostępna ilość pamięci RAM. O ile mikrokontroler od STM posiada $128KB$ pamięci RAM oraz dodatkowe $64KB$ CCMRAM, to LPC1768, posiada zaledwie $32KB$ pamięci RAM oraz dodatkowo $32KB$ pamięci SRAM. Jest to zasadnicze ograniczenie, które wymusiło poważne optymalizacje po stronie aplikacji.

Najważniejszą zmianą było właśnie wykorzystanie pamięci RAM. Domyślnie LWIP alokowało całą swoją pamięć w RAM, jednak na LPC1768, przestrzeń ta zaledwie wystarcza na niewielki stos oraz stertę ze względu na wykorzystanie systemu czasu rzeczywistego. Na szczęście biblioteka LWIP, posiada własny alokator, któremu można wskazać przestrzeń, na której może operować. Na LPC1768 przestrzeń SRAM, została więc przeznaczona, na obsługę stosu sieciowego, a w pamięci RAM został umieszczony stos oraz sterta głównego programu.

Pomimo tych modyfikacji aplikacja nie była w stanie poprawnie się uruchomić na platformie LPC1768. Po godzinach debugowania problem został zlokalizowany w implementacji sterownika CMSIS-Driver, który pobrano ze strony ARM. W funkcji do odczytywania rozmiaru odczytanej ramki sprawdzane było wystąpienie flagi `RINFO_LOST_FLAG` lub `RINFO_ERR_MASK`. Po porównaniu implementacji tego sterownika dla STM zauważono, że tam nie występuje tego typu sprawdzenie. Fragment ten został więc wyłączony z dostarczonej implementacji, co spowodowało, że aplikacja poprawnie się uruchomiła. Po kilku testach manualnych można było stwierdzić, że implementacja działa analogicznie do tej zrealizowanej na platformie STM.

3.4 Wnioski

Ostateczny efekt został osiągnięty i aplikacja z minimalnymi zmianami, które dotyczyły zarządzenia pamięcią w LWIP, została uruchomiona. Zmiany dokonane w konfiguracji LWIP nie można traktować jako porażkę, są one bowiem wynikiem istotnych różnic, a co za tym idzie ograniczeń w testowanych urządzeniach. Gdyby do testu wybrać inny model z większą ilością pamięci RAM, zmiana ta nie byłaby konieczna. Patrząc na problem od drugiej strony, gdyby proces developerski rozpocząć od implementacji na LPC1768, również natrafilibyśmy na problem zbyt małej pamięci i konieczności wskazania LWIP innych obszarów. Takie obejście problemu pamięci nie jest konieczne na STM, a nawet gdybyśmy chcieli zachować konwencję alokowania pamięci na stos TCP/IP w osobnej części pamięci mikrokontrolera, konieczna byłaby modyfikacja wskaźnika, od którego LWIP może rozpoczynać alokację. Zmiana ta jest więc koniecznością i żadne rozwiązanie nie uchroniłoby programistę przed jego dokonaniem.

Niepokojące jest jednak to, że uruchomienie aplikacji na LPC1768 wymagało modyfikacji kodu dostarczonego w pakiecie CMSIS-Pack. Fakt ten jest jasnym dowodem, że każdy kod

[13] Implementacja projektu dostępna jest w publicznym repozytorium `https://github.com/robgal519/cmsis_lpc_gcc`

narażony jest na błędy. Niewykluczone oczywiście, że problem leży w innej części projektu, a usunięcie tego sprawdzenia wyciszyło jedynie nieprzyjemny symptom. Sytuacja ta pokazuje jednak, że pomimo iż oszczędziliśmy znaczną ilość czasu, który należałoby poświecić na dostosowanie lub stworzenie odpowiedniego sterownika do Ethernet PHY, to czas ten może w każdym przypadku zostać zmarnowany na szukanie błędu w dostarczonej implementacji.

Podsumowanie

Celem prezentowanej pracy było przetestowanie interfejsu CMSIS-Driver służącego do obsługi peryferiów mikrokontrolera. Interfejs ten został przeanalizowany pod kątem wydajności w porównaniu z dwoma innymi implementacjami – STM-HAL oraz implementacji opartej o rejestry. Testy wykazały, że nie jest to rozwiązanie doskonałe pod względem wydajności, ale CMSIS-Driver może być wartym uwagi rozwiązaniem. Pomimo iż w testach szybkości transferu danych nie było możliwe osiągnięcie maksymalnej prędkości oferowanej przez mikrokontroler i układ pomiarowy, to należy stwierdzić, że problem ten wynikał głównie z konieczności dostosowania interfejsu do szerszej grupy urządzeń. W przypadku konieczności korzystania z maksymalnej prędkości konieczna byłaby modyfikacja obecnej implementacji CMSIS-Driver.

Analizowany kod implementujący interfejs CMSIS-Driver, nie zachwyca czytelnością i jakością, co sprawia, że rozwiązanie to może nie być odpowiednie w projektach, gdzie jakość kodu jest kluczowym aspektem w rozwoju oprogramowania. Jednak ogromną zaletą interfejsu jest duża baza implementacji wspierająca znaczną ilość popularnych mikrokontrolerów. Dzięki temu możliwa jest migracja kodu opartego o ten interfejs na inną platformę, co znacznie ułatwia proces prototypownia, oraz może być kluczowe w projektach pracujących w metodologiach zwinnych, gdzie nowe wymagania mogą wymusić konieczność zmiany mikrokontrolera.

Zdaniem autora interfejs ten stanowi doskonałe narzędzie do szybkiego prototypownia. Programista może stosunkowo łatwo i szybko stworzyć projekt, nie posiadając docelowego mikrokontrolera, co znacznie skraca proces rozwoju oprogramowania. Dostępne implementacje dla tego interfejsu nie reprezentują jednak najwyższej jakości, co może powodować problemy przy próbie ich modyfikacji, lub podczas szukania błędów.

Literatura

[1] CMSIS-Core (Cortex-M). https://arm-software.github.io/CMSIS_5/Core/html/index.html.

[2] DMA on embedded systems. https://docs.majerle.eu/projects/lwrb/en/latest/user-manual/hw-dma-usage.html.

[3] Lightweight IP stack. http://www.nongnu.org/lwip/2_1_x/index.html.

[4] Real-time operating system (RTOS): Components, Types, Examples. https://www.guru99.com/real-time-operating-system.html.

[5] Embedded Basics – API's vs HAL's. https://www.beningo.com/embedded-basics-apis-vs-hals, 2016.

[6] Michael Barr. *Programming embedded systems : with C and GNU development tools.* O'Reilly, Sebastopol, Calif, 2006.

[7] Jacob Beningo. Building reusable device drivers for microcontrollers. https://www.embedded.com/building-reusable-device-drivers-for-microcontrollers/, January 2013.

[8] BOSCH. *BME280 Combined humidity and pressure sensor*, Nov 2014.

[9] STMICROELECTRONICS. *Datasheet STM32F405xx STM32F407xx*, Sep 2016.