POLITECHNIKA WARSZAWSKA
Wydział Elektroniki i Technik Informacyjnych
Instytut Systemów Elektronicznych

Michał Paszta
numer albumu 206147

PRACA DYPLOMOWA INŻYNIERSKA

Implementacja algorytmów samokorygujących w sensorowych sieciach bezprzewodowych na bazie mikrokontrolera z rdzeniem ARM i cyfrowego łącza transmisji radiowej

Praca wykonana po kierunkiem
mgr. inż. Krzysztofa Gołofita

Warszawa, 2010

Implementacja algorytmów samokorygujących w sensorowych sieciach bezprzewodowych na bazie mikrokontrolera z rdzeniem ARM i cyfrowego łącza transmisji radiowej

Celem pracy było zaprojektowanie, wykonanie i oprogramowanie urządzenia, które mogłoby stać się funkcjonalnym węzłem bezprzewodowej sieci sensorowej. Podstawowa architektura urządzenia zawiera mikrokontroler z rdzeniem ARM oraz moduł nadawczo-odbiorczy. Oprogramowanie, napisane w języku C, powstało w oparciu o system czasu rzeczywistego *FreeRTOS*, przy użyciu jednego z narzędzi *open-source*. Istotną cechą aplikacji jest jej modularna budowa, umożliwiająca dodawanie nowych funkcji (np. w postaci modułów kodowania korekcyjnego). Powstałe w rezultacie pracy urządzenie pozwoliło na implementację i przetestowanie transmisji przy użyciu kilku odmiennych kodów korekcyjnych.

Error correcting codes implementation for wireless sensor networks, based on a microprocessor with an ARM core and digital radio communication

The aim of this thesis was to design, construct and programme a device, which could become a functional node in a wireless sensor network. The basic architecture of the device consists of an ARM microcontroller and a transceiver. Software was entirely based on the C programming language and one of the *open-source* environments. Application uses a real-time operating system – *FreeRTOS* and has a modular structure, allowing easy addition of new features (e.g. error-correction coding modules). The created device was tested for proper communication, using several different coding methods.

Michał Paszta urodzony 21.06.1987 w Piotrkowie Trybunalskim uczęszczał w latach 2003–2006 do I Liceum Ogólnokształcącego im. Władysława Broniewskiego, które ukończył z wyróżnieniem. Od 2006 roku student Wydziału Elektroniki i Technik Informacyjnych Politechniki Warszawskiej — obecnie na specjalności Elektronika i Inżynieria Komputerowa. W 2009 roku, w ramach programu LLP Erasmus, studia na Metropolia University of Applied Sciences, w Espoo, w Finlandii. Finalista Olimpiady Języka Angielskiego (rok 2005) oraz reprezentant Politechniki Warszawskiej (w 2010 roku) w Ogólnopolskiej Olimpiadzie Języka Angielskiego Wyższych Uczelni Technicznych z wynikiem: II miejsce.

Współpraca z Instytutem łączności

Instytut Łączności, powstały w 1951 roku w efekcie podziału Państwowego Instytutu Telekomunikacyjnego na dwie instytucje, jest niezależną placówką badawczo-rozwojową, w której wiodącymi kierunkami są telekomunikacja i techniki informacyjne. W ramach działalności Instytut przyczynia się do rozwoju krajowej gospodarki poprzez udzielanie wsparcia technicznego, badawczego i naukowego różnym instytucjom państwowym.

Praktyczna część pracy dyplomowej została zrealizowana w Instytucie Łączności pod merytoryczną opieką dr. inż. Roberta Samborskiego. Z tworzeniem pracy powiązane były półroczne praktyki studenckie. Instytut udostępnił stanowisko robocze oraz sprzęt niezbędny do realizacji pracy oraz umożliwił wykonanie obwodów drukowanych dla prototypu urządzenia. Niektóre rozwiązania zastosowane w pracy (m. in. sposób połączenia układów) zostały zaczerpnięte z badań dr. inż. Roberta Samborskiego oraz mgr. inż. Edwarda Chrustowskiego.

Spis treści

Wykaz skrótów:

ARM	Advanced RISC Machine
BCH	kod Bosego Ray-Chauduriego i Hocquenghema
CRC	Cyclic Redundancy Check
FSK	Frequency-shift keying
FreeRTOS	Free Real Time Operating System
GF	Galois Field
GNU	GNU is Not Unix
JTAG	Joint Test Action Group
LDPC	Low-Density Parity-Check
OpenOCD	Open On-Chip Debugger
OSI	Open System Interconnection
PCB	Printed Circuit Board
PIO	Programmed Input/Output
PPP	Point to Point Protocol
RISC	Reduced Instruction Set Computers
SECDED	Single Error Correction Double Error Detection
SPI	Serial Peripheral Interface
SRAM	Static Random Access Memory
TCP/IP	Transmission Control Protocol / Internet Protocol
USART	Universal Synchronous and Asynchronous Receiver and Transmitter
USB	Universal Serial Bus
XOR	Exclusive OR
YAGARTO	Yet Another GNU ARM Toolchain

1. Wstęp

Bezprzewodowe sieci sensorowe są sieciami złożonymi z wielu urządzeń rozproszonych na pewnym obszarze i komunikujących się bezprzewodowo z innymi urządzeniami. Stosowane są obecnie powszechnie – szczególnie tam, gdzie użycie komunikacji kablowej jest niewygodne lub niemożliwe. Poszczególne węzły sieci sensorowych składają się przeważnie z czujnika, mikrokontrolera, układu nadawczo-odbiorczego i źródła zasilania [9]. W zależności od doboru tych składników, sieć może posiadać różne cechy, a zatem i różne zastosowania.

Czujniki używane w sieciach sensorowych mogą różnić się sposobem pracy – np. w zależności od mierzonej wielkości, czy roli spełnianej przez węzeł. Pomiar najczęściej następuje w pewnych odstępach czasowych a uzyskane w skutek pomiaru dane zapisywane są w pamięci lub przesyłane natychmiast. Pomiary ciągłe stosuje się w odniesieniu do szybko zmieniających się wielkości i gdy istnieje potrzeba ciągłego ich monitorowania (co przekłada się niestety na wysoki pobór mocy). Ze sposobu dokonywania pomiaru i obsługi czujnika wynikają kolejne cechy węzła. Może on pracować w sposób ciągły lub przechodzić w stan obniżonego poboru energii, kiedy pozostaje bezczynny. W zależności od ilości danych dostarczanych przez czujnik, wolnej pamięci oraz wagi informacji, aktywność radiowa węzła może się zmieniać.

Dobór źródła zasilania również zależy od specyfiki pomiaru. W związku z tym, że węzły zasilane są najczęściej bateryjnie, pojemność baterii dobierana jest tak, aby osiągnąć założoną żywotność przy oczekiwanym zużyciu energii. Złożoność obliczeniowa, której musi podołać węzeł, determinuje wybór mikrokontrolera i sposób jego pracy. Węzeł może również dokonywać wstępnej obróbki uzyskanych danych, co dodatkowo zwiększa obciążenie mikrokontrolera, a co za tym idzie, także zużycie mocy.

Kolejną ważną cechą jest struktura sieci i sposób komunikacji pomiędzy węzłami. Najczęściej stosuje się komunikację przy pomocy fal radiowych (rzadziej podczerwieni lub mikrofal[9]) a moc nadajników zależy od stopnia rozproszenia sieci oraz przeszkód i zakłóceń. Węzły mogą przesyłać dane z czujnika do wyznaczonych elementów docelowych sieci albo mogą same pełnić rolę transmisyjną – przekazywać informacje pomiędzy innymi węzłami. Wybór koncepcji zależy od możliwości urządzeń i obszaru, który ma zostać pokryty – co determinuje też topologię sieci (gwiazda, pierścień, drzewo itp.). W związku z tak szerokim wachlarzem zastosowań, węzły sieci tworzone są z przeznaczeniem do funkcjonowania w konkretnych warunkach.

Celem niniejszej pracy jest stworzenie takiego urządzenia, które można by było wykorzystać w sieci o wysokim stopniu rozproszenia, tj. przy odległościach między węzłami dochodzących do granicy zasięgu. Dane, przesyłane na takie odległości, łatwo ulegają zakłóceniom i przekłamaniom, ponieważ moc docierającego do odbiornika sygnału jest istotnie osłabiona.

Aby niwelować negatywne skutki transmisji, stosowane są specjalne protokoły komunikacji oraz nadmiarowe kodowanie korekcyjne. Wybór konkretnych rozwiązań w dużym stopniu zależy jednak od ostatecznego przeznaczenia węzła, które z perspektywy niniejszej pracy pozostaje nieznane. W związku z tym, położony został duży nacisk na modularną budowę oprogramowania mikrokontrolera w stosunku do rozwiązań dotyczących kodów korekcyjnych i protokołu komunikacji – umożliwiając zdefiniowanie funkcjonalności urządzenia przez ostatecznego użytkownika. Możliwe jest na przykład zwiększenie bezpieczeństwa transmisji kosztem narzutu informacji nadmiarowej, które wydłuża czas przesyłania danych, ale zarazem zmniejsza liczbę koniecznych retransmisji.

Optymalizacja parametrów kanału komunikacyjnego jest zagadnieniem złożonym, zależnym od wielu czynników (w tym od środowiska pracy sieci) i w związku z tym nie będzie szerzej omawiana na łamach tej pracy. Celem implementacji protokołu komunikacji (wyposażonego w kanał zwrotny oraz funkcję retransmisji) i kilku przykładowych kodów korekcyjnych jest zapewnienie możliwości weryfikacji funkcjonalności urządzenia.

Oprogramowanie węzła sieci sensorowej wymaga doboru odpowiednich narzędzi programistycznych. Wiele dostępnych na rynku urządzeń opiera się o rozwiązania komercyjne, jednak z uwagi na pilotażowy charakter pracy, całość tworzonego oprogramowania oparto o narzędzia *open-source*[1], które osiągnęły już wysoki stopień stabilności i użyteczności.

[1] oprogramowanie o udostępnionym publicznie kodzie źródłowym

2. Przegląd istniejących rozwiązań

2.1. Procesory z rdzeniem ARM

Procesory z rdzeniem ARM wykorzystywane są powszechnie w urządzeniach mobilnych, m.in. ze względu na niski pobór mocy. Najnowsze jednostki z tej rodziny dysponują mocą obliczeniową porównywalną z komputerami osobistymi [6], ale do węzła sieci czujnikowej wystarcza najczęściej starsze urządzenie, np. wyposażone w rdzeń ARM7.

Jednym z rozwiązań mogących stać się platformą docelową dla powstającego oprogramowania jest procesor ARM7TDMI, umieszczony w układzie scalonym AT91SAM7S256, firmy Atmel. Procesor taktowany jest zegarem o częstotliwości do 55 MHz. Jego budowa oparta jest o architekturę von Neumanna ze zredukowaną listą rozkazów (ang. RISC – *Reduced Instruction Set Computers*). Mimo, że nie jest to najnowsze dostępne urządzenie[1], posiada ono wystarczające parametry do realizacji zadań. Mikrokontroler AT91SAM7S256, w którym osadzony jest procesor, dysponuje 256 kilobajtami pamięci flash oraz 64 kilobajtami pamięci SRAM (od ang. *Static Random Access Memory*) [1].

Układ AT91SAM7S został umieszczony na minimodule MMSAM7S firmy Propox, który posiada wygodne wyprowadzenia (rozmieszczone w jednocalowych odstępach) pasujące do typowych układów prototypowych. Układ ten jest też wyposażony w stabilizatory napięcia (5 i 3,3V) [16].

Firma Propox wyprodukowała również płytę ewaluacyjną dla układu MMSAM7S, umożliwiającą jeszcze wygodniejsze korzystanie z mikroprocesora. Układ EVBsam7s wprowadza szereg ułatwień: możliwość zasilania układu z gniazdka sieciowego, klawisze i diody, port RS232, złącze JTAG (od ang. *Joint Test Access Group*), złącza wszystkich peryferiów, które obsługuje układ AT91SAM7S oraz inne, których nie wykorzystano do realizacji projektu [15]. Programowanie pamięci flash procesora, z której odczytywany jest program, następuje przy pomocy programatora [3]. Do realizacji pracy dyplomowej użyto programatora USB, opartego o popularny układ FTDI-2232.

2.2. Układ nadawczo-odbiorczy

Układem elektronicznym, niezbędnym do wykonania postawionego zadania, było urządzenie umożliwiające powstanie cyfrowego łącza transmisji radiowej. W tym przypadku zastosowany został układ nadawczo-odbiorczy AT86RF211 firmy Atmel, wykorzystujący modulację FSK (od ang. *Frequency-shift keying*) [2], w której informacja kodowana jest dwoma różnymi częstotliwościami sygnału [18].

[1] ARM7 powstał w 1994 roku, potem produkowano także ARM9 (od 1995 roku), ARM10 (od 1998 roku) i ARM11 (od 2002 roku). Dane ze strony firmy ARM: www.arm.com

W specyfikacji technicznej układ ten jest opisany jako działający na zasadzie jednego z peryferiów mikrokontrolera [2]. Istotnie, po poprawnej konfiguracji rejestrów układu, wystarczy na jego wejście danych DATAMSG nadawać bity przesyłanego komunikatu, a modulacja zachodzi w pełni automatycznie. Również w przypadku odbioru danych, odczyt następuje poprzez przesył automatycznie zdemodulowanych danych na końcówkę DATAMSG.

Szybkość na jaką pozwala układ AT86RF211 w transmisji synchronicznej waha się między 2400 b/s (bitów na sekundę) a 64 kb/s. W pracy została jednak wykorzystana, wspominana w dokumentacji firmy Atmel, transmisja asynchroniczna. Z obserwacji dokonanych podczas testów układu wynika, że transmisje synchroniczna i asynchroniczna mają takie same zdolności do przesyłania danych. Przy szybkości transmisji przekraczającej 38400 b/s oba tryby transmisji danych powodowały zakłócenia. Układ AT86RF211 charakteryzuje się niedużą przepustowością, ale za to dużym zasięgiem, jak na tego typu układy nadawczo-odbiorcze (rzędu kilkuset metrów [2]).

Wybór komponentów urządzenia – mikrokontrolera i układu nadawczo-odbiorczego – determinuje obszar zastosowań, w jakim węzeł może zostać użyty. Na przykład niski pobór mocy, daleki zasięg, lecz niewielka przepustowość pozwalają na skonstruowanie sieci czujnikowej o wysokim stopniu rozproszenia węzłów. W warunkach pracy na granicy zasięgu urządzeń, przydatne może się okazać zabezpieczenie przesyłanych informacji kodem korekcyjnym lub detekcyjnym.

2.3. Środowisko programistyczne

Do zaprogramowania mikrokontrolera niezbędny jest odpowiedni interfejs sprzętowy oraz oprogramowanie pozwalające go wykorzystać. Pierwszym etapem programowania jest stworzenie kodu źródłowego. Następnie kod należy skompilować tak, aby działał on na platformie docelowej[2]. Platformą macierzystą może być dowolny procesor stacji roboczej, używanej przez programistę, a docelową jest w tym wypadku 32-bitowy mikrokontroler z rdzeniem ARM7. Kiedy gotowy jest skompilowany program trzeba go załadować do pamięci flash urządzenia docelowego. Do tego służy programator łączący port wyjściowy stacji roboczej, na której powstał kod, a więc, w rozpatrywanym przypadku, port USB z odpowiednim interfejsem w układzie docelowym – na przykład z interfejsem JTAG. W programatorze następuje przekształcenie danych ze stacji roboczej na sygnały elektryczne, zdolne do zapisania informacji w pamięci flash. Do obsługi programatora potrzebne jest odpowiednie oprogramowanie po stronie stacji roboczej, oraz odpowiednia warstwa fizyczna układu docelowego. Takie zestawienie oprogramowania i sprzętu – edytora tekstu, kompilatora i programatora – stanowi *toolchain*[3].

[2] w języku angielskim proces ten określany jest wyrażeniem *cross-compilation*

[3] zestaw narzędzi programistycznych, w tym przypadku pozwalających na zaprogramowanie mikrokontrolera; nie istnieje polski odpowiednik angielskiego słowa *toolchain*, więc w pracy używane będzie wyrażenie angielskie

Aby zbadać użyteczność narzędzi *open-source* w programowaniu mikrokontrolerów oraz wykorzystać zainstalowany na stacji roboczej system operacyjny Linux, ograniczono wybór środowiska programistycznego, rezygnując z popularnych programów komercyjnych. Rozwiązania o otwartym kodzie źródłowym i na licencji pozwalającej na bezpłatne wykorzystanie posiadają pewne wady: wymagają większego nakładu pracy i doświadczenia przy instalacji oraz nie posiadają oficjalnego wsparcia technicznego. Zalety to bezpłatność i szerokie wsparcie społeczności tworzącej i użytkującej takie oprogramowanie.

Poniżej przedstawiono wybrane *toolchainy*.

2.3.1. GNU ARM ™

Jednym ze sposobów skonstruowania *toolchaina* jest instalacja kompilatora GCC (od ang. *GNU*[4] *Compiler Collection*) wraz z pakietami systemu Linux: *newlib* i *binutils*[5]. Odpowiednia konfiguracja tych pakietów (przed kompilacją) umożliwia późniejszą kompilację kodu źródłowego z przeznaczeniem na inną platformę niż stacja robocza.

Projekt GNU ARM udostępnia skompilowane już odpowiednio oprogramowanie, które można uruchomić w systemie Linux lub w systemie Microsoft Windows – wykorzystującym oprogramowanie Cygwin, pozwalające na uruchamianie aplikacji linuksowych w systemach Win32. Na stronie domowej projektu[6] istnieją także wskazówki jak skompilować własny toolchain na bazie tych narzędzi. W skład GNU ARM wchodzi też *debugger*[7] GDB (GNU Debugger). Całe oprogramowanie jest bezpłatne i jest dostarczane z kodem źródłowym. Projekt GNU ARM nie dostarcza oprogramowania sterującego programatorem.

2.3.2. YAGARTO

Twórcy *toolchaina* YAGARTO (od ang. *Yet Another GNU ARM Toolchain*) przyświecały (jak napisał na stronie projektu[8]) następujące idee:

- nieużywanie oprogramowania Cygwin, w przeciwieństwie do większości *toolchainów* działających w systemie operacyjnym Microsoft Windows,
- współpraca ze środowiskiem Eclipse,
- niskie koszty i łatwość instalacji – potencjalnymi użytkownikami są studenci i hobbyści.

Oprogramowanie cieszy się dobrą opinią wśród programistów mikrokontrolerów, jednak ponieważ jest ono przeznaczone dla użytkowników systemu Windows, nie zostanie wykorzystane w niniejszej pracy.

[4] GNU pochodzi od *GNU is Not Unix*

[5] *newlib* zawiera podstawowe biblioteki języka C, i jest odpowiednikiem pakietu *glibc* dla systemów wbudowanych, a *binutils* zawiera oprogramowanie niezbędne do tworzenia plików wykonywalnych

[6] www.gnuarm.com

[7] z ang. dosł. *odpluskwiacz* – aplikacja służąca do poszukiwania błędów (ang. *bugs*) w programie. W niniejszej pracy będzie używane angielskie określenie.

[8] www.yagarto.de

2.3.3. Eclipse

Eclipse jest zintegrowanym środowiskiem programistycznym, napisanym w języku Java, początkowo stworzonym przez firmę IBM, a obecnie, po upublicznieniu kodu źródłowego, rozwijanym przez powołaną w tym celu fundację. Pełne narzędzie – *toolchain*, oparte o Eclipse, składa się z maszyny wirtualnej Java, środowiska Eclipse, opisanego w poprzednim punkcie kompilatora GNU ARM™, *debuggera* GDB oraz oprogramowania obsługującego programator (np. OpenOCD). Po instalacji i konfiguracji wszystkich komponentów można uzyskać wygodne, w pełni bezpłatne, graficzne środowisko pracy, które wyglądem i sposobem użytkowania przypomina komercyjne środowiska programistyczne warte kilkaset dolarów[9].

2.3.4. OpenOCD

Projekt OpenOCD (Open On-Chip Debugger) powstał pierwotnie jako praca dyplomowa Dominica Ratha, studenta University of Applied Sciences w Augsburgu. Obecnie jest intensywnie rozwijany przez społeczność, ale wciąż nie powstała oficjalna, stabilna wersja programu. Celem projektu jest dostarczenie narzędzia do programowania i *debuggowania* urządzeń wbudowanych. OpenOCD obsługuje kilkadziesiąt różnych typów programatorów a na liście obsługiwanych procesorów docelowych jest obecnie 16 pozycji [5].

Najlepszą kontrolę procesu powstawania programu wykonywalnego dla mikroprocesora zapewnia najprostsze rozwiązanie, polegające na ręcznej kompilacji i konfiguracji poszczególnych elementów pakietu GNU ARM oraz wykorzystaniu narzędzia OpenOCD do programowania pamięci flash. *Toolchain* złożony z tych elementów został wykorzystany do stworzenia oprogramowania. Sposób jego instalacji i konfiguracji w systemie Slackware Linux 13.0 jest zaprezentowany w dodatku C.

Językami programowania wykorzystywanymi do programowania mikrokontrolerów są assembler i język C. Do tworzenia kodu źródłowego został użyty ten drugi, ze względu na większy komfort w pisaniu skomplikowanych aplikacji. Kod źródłowy powstawał w oparciu o rozwiązania sugerowane przez producentów układów i opisanego w rozdziale 4 oprogramowania[10].

[9] Jako przykład można podać środowisko CrossWorks firmy Rowley Associates, które do zastosowań edukacyjnych kosztuje 300$ (dane z 21.05.2010, z oficjalnej strony firmy: http://www.rowley.co.uk/).

[10] Twórcy systemu FreeRTOS, opisanego w rozdziale 4, wprost zalecają tworzenie własnych projektów na bazie odpowiednich projektów z katalogu *Demo*, dostarczanych wraz z dystrybucją[4]

2.4. Kody korekcyjne

W każdej cyfrowej transmisji danych istnieje pewne prawdopodobieństwo wystąpienia błędu. Jest ono tym większe, im gorszej jakości jest kanał transmisyjny i im gorsze są warunki w jakich pracują urządzenia nadawcze i odbiorcze. W celu umożliwienia poprawnej pracy systemów korzystających z niepewnych kanałów przesyłu danych, konieczne jest zabezpieczenie ich przed błędami. W tym celu przesyłane są informacje nadmiarowe, które po odebraniu danych umożliwiają detekcję i/lub naprawę ewentualnych błędów.

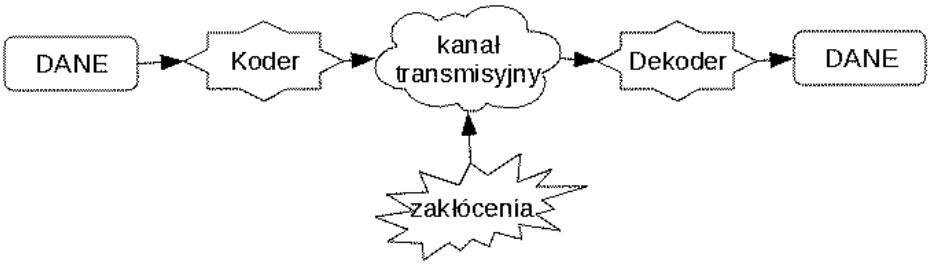
Poszczególne sposoby kodowania korekcyjnego różnią się między sobą trzema podstawowymi parametrami:

- sprawnością, czyli stosunkiem liczby bitów wejściowych k do liczby bitów wyjściowych n kodera [11],
- złożonością obliczeniową (najczęściej zależną od rozmiarów przetwarzanych bloków),
- zdolnościami korekcyjnymi i detekcyjnymi (zależnymi od zastosowanych algorytmów).

Parametry te są ze sobą powiązane a ich oczekiwana wartość determinuje wybór kodu korekcyjnego, dopasowanego do docelowych warunków pracy urządzenia.

Konieczne jest rozróżnienie pojęć detekcji i korekcji błędów. Jeśli dysponujemy kodem pozwalającym tylko na detekcję błędów, to w przypadku wykrycia przekłamania jedyną możliwością jest odrzucenie błędnych danych i ich ponowny przesył. Korekcja natomiast sprawia, że po detekcji błędu możliwe jest stwierdzenie jak wyglądał poprawny komunikat na podstawie nadmiarowych informacji, które przesłano wraz z nim.

Prosty i przejrzysty model systemu komunikacyjnego z korekcją danych przedstawił W. Mochnacki w [11], a jego uproszczoną wersję pokazano na rysunku 2.1.



Rysunek 2.1. Model systemu komunikacyjnego.

Przed omówieniem kodów korekcyjnych konieczne jest wprowadzenie pewnych pojęć z zakresu algebry, o które opierają się konstrukcje kodów. Zagadnienia te zostały dokładnie opisane w dodatku A (również w [8, 11]).

Przy opisie działania urządzeń elektronicznych, a więc także przy opisie działania kodów korekcyjnych, często używa się pojęcia ciała skończonego $GF(2)$[11], posiadającego dwa elementy: 0 i 1, w którym zdefiniowane są dodawanie modulo 2 i mnożenie [11].

Wielomiany nad ciałami skończonymi $GF(q)$ mają ogólną postać:

$$p(x) = \sum_{i=0}^{m} a_i x^i = a_m x^m + a_{m-1} x^{m-1} + ... + a_1 x + a_0, \text{ gdzie } a_i \in GF(q) \qquad (2.1)$$

W przypadku $q = 2$ można przyjąć, że współczynniki będą miały jedynie wartości 0 lub 1.

Kody korekcyjne dzielone są w literaturze [10, 11] na dwie kategorie: kody splotowe i kody blokowe.

2.4.1. Kody splotowe

Kody splotowe przekształcają k-elementowy ciąg informacyjny na n-elementowy ciąg kodowy[12], na podstawie bieżącego elementu informacyjnego[13] oraz pewnej liczby elementów, które go poprzedzały [11]. W związku z tym informacja o poprzednich elementach informacyjnych musi być zapamiętywana przez koder [12].

Kodowanie polega na obliczaniu bitów kodowych na podstawie bitów informacyjnych, znajdujących się w pamięci kodera [12]. Bity kodowe powstają poprzez sumowanie[14] określonych bitów informacyjnych. To, które bity zostaną zsumowane, regulują tzw. wielomiany generujące. Każdemu bitowi wyjściowemu przypisany jest jeden wielomian generujący. Od doboru tych wielomianów zależą właściwości kodu.

W wektorach kodowych powstałych w efekcie kodowania splotowego bity informacyjne i nadmiarowe są przemieszane. Stąd nazwa tej grupy metod kodowania.

Dekodowanie kodów splotowych może zachodzić na kilka różnych sposobów, które można podzielić na dwie grupy: dekodowanie sekwencyjne i dekodowanie metodą największego prawdopodobieństwa.

Dekodowanie sekwencyjne polega na analizie nadchodzącego ciągu kodowego bit po bicie. Ilekroć otrzymane bity nie zgadzają się z możliwymi do otrzymania, dekoder przyjmuje jedną z możliwych prawidłowych kombinacji, zwiększa licznik błędu i, bazując na poprzednim wyborze, analizuje kolejne bity. W przypadku gdy licznik błędu osiągnie ustaloną wcześniej wartość, następuje cofnięcie dekodera o jeden krok, gdyż oznacza to, że jeden z przyjętych za prawidłowe bitów, został przyjęty błędnie. Jeśli możliwe jest przyjęcie wszystkich bitów tak, aby pasowały do oczekiwanych, prawidłowych słów kodowych, bądź też

[11] ciało skończone oznacza się również przez $GF(q)$ (od ang. *Galois Field*), gdzie q oznacza liczbę elementów ciała

[12] Mianem „słowo kodowe", „ciąg kodowy" lub „wektor kodowy" określane są ciągi zakodowanych bitów w kanale komunikacyjnym. Dla odróżnienia niezakodowane dane nazywane są „wektorem informacyjnym" lub „ciągiem informacyjnym".

[13] Przy programowej implementacji kodów korekcyjnych dla mikroprocesora, można mówić jedynie o binarnej reprezentacji danych, więc elementami ciągów informacyjnych i kodowych są bity.

[14] sumowanie modulo 2

możliwa jest zmiana bitów uznanych za błędne tak, aby licznik błędów nie przekroczył zadanej wartości, to otrzymany ciąg zostanie zdekodowany.

Dekodowanie metodą największego prawdopodobieństwa polega na obliczeniu różnic między możliwymi do otrzymania ciągami bitów a prawidłowymi, oczekiwanymi ciągami. Najczęściej stosuje się w tym przypadku algorytm Viterbiego, dokładniej opisany w [12].

Kodów splotowych używa się tam, gdzie dane przesyłane są strumieniowo: w komunikacji satelitarnej, w telefonii komórkowej, modemach, przy bezprzewodowym dostępie do internetu [12].

2.4.2. Kody blokowe

Kody blokowe operują na blokach danych o określonej długości. Na podstawie danych w bloku obliczana jest sekwencja kontrolna, która dodawana jest do ciągu informacyjnego.

Istotnym pojęciem przy analizie kodów blokowych jest tzw. odległość Hamminga. Jest to parametr mówiący o tym, na ilu pozycjach różnią się między sobą wektory kodowe. Minimalna odległość Hamminga, czyli odległość Hamminga pomiędzy dwoma możliwie najbliższymi wektorami danego kodu korekcyjnego, decyduje o możliwościach detekcji i korekcji błędów [10].

Kody blokowe można dalej podzielić na liniowe i cykliczne ze względu na konstrukcję i wykorzystywane struktury algebraiczne.

2.4.2.1. Kody liniowe

Blokowy kod liniowy (n, k)[15] to kod opisany następującym układem równań [11]:

$$\sum_{i=1}^{n} b_{ij} a_i = 0 \quad \text{, gdzie } \; j = 1, 2, ..., n-k \tag{2.2}$$

gdzie współczynniki a są elementami wektora kodowego, zaś b przyjmują tylko wartości 0 i 1.

Powyższy układ równań można też przedstawić w postaci macierzy[16] P o wymiarach $k \times (n-k)$. Z macierzy P można wyznaczyć tzw. macierz generującą, mającą postać:

$$G_{k \times n} = [P_{k \times (n-k)} \quad I_{k \times k}] \tag{2.3}$$

gdzie G powstaje w wyniku konkatenacji poziomej[17] (sklejenia poziomego) macierzy $P_{k \times (n-k)}$ i $I_{k \times k}$. Macierz I jest macierzą jednostkową[18].

[15] zapis „kod (n,k)" oznacza kod o długości słowa wejściowego k i długości słowa wyjściowego n

[16] Funkcję $A : \{1, 2, .., m\} \times \{1, 2, .., n\} \rightarrow K$ nazywamy macierzą stopnia $m \times n$ o elementach z ciała K. Wartość tej funkcji dla par (i, j) oznaczamy przez a_{ij} zaś macierz o elementach a_{ij} przez $A = [a_{ij}]_{m \times n}$ [8].

[17] pierwsza kolumna macierzy I jest $n - k + 1$-szą kolumną macierzy G

[18] macierz, na której głównej przekątnej są same jedynki, a pozostałe miejsca są wypełnione zerami

Iloczynem[19] wektora informacyjnego i macierzy generującej jest wektor kodowy[20]:

$$c = m \cdot G \tag{2.4}$$

Działanie takie jest równoznaczne rozwiązaniu układu równań 2.2.

Proces dekodowania, w dużym uproszczeniu, polega na dopasowaniu otrzymanego wektora do takiego poprawnego wektora kodowego, który go najbardziej przypomina. W szczególności może się zdarzyć, że po przejściu przez kanał komunikacyjny wektor kodowy uległ przekształceniu w inny, poprawny wektor kodowy i wówczas błędy nie zostaną zdetekowane. Prawdopodobieństwo takiego zdarzenia jest tym większe im bardziej zakłócony jest kanał transmisyjny i im mniejszą minimalną odległość Hamminga posiada kod zabezpieczający.

Do dekodowania niezbędna jest informacja o tym, jak wektor był zakodowany. W tym celu można wykorzystać tzw. macierz kontroli parzystości. Ma ona postać podobną do macierzy generującej (2.3), ale powstaje poprzez konkatenację poziomą macierzy jednostkowej $I_{(n-k)\times(n-k)}$ z transponowaną[21] macierzą $P_{k\times(n-k)}$:

$$H = [I_{(n-k)\times(n-k)} \quad P^T_{k\times(n-k)}] \tag{2.5}$$

Do obliczeń wykorzystuje się ją po wykonaniu transpozycji (wówczas jest ona wynikiem konkatenacji pionowej dwóch macierzy):

$$H^T = \begin{bmatrix} P_{k\times(n-k)} \\ I_{(n-k)\times(n-k)} \end{bmatrix} \tag{2.6}$$

Po pomnożeniu otrzymanego wektora przez transponowaną macierz parzystości, powstanie wektor nazywany syndromem. Syndrom wskazuje ile błędów wystąpiło w transmisji i na których miejscach wektora kodowego się znajdują. Jeśli syndrom jest wektorem zerowym, to transmisja przebiegła bez zakłóceń lub zakłócenia przekształciły dany wektor kodowy w inny, poprawny wektor kodowy. W przeciwnym wypadku, można przy pomocy syndromu wyznaczyć tzw. wektor błędu, który po dodaniu do otrzymanego wektora, spowoduje jego korekcję. Liczba bitów w syndromie jest równa liczbie bitów nadmiarowych, dodawanych w procesie kodowania.

Poniżej przedstawiono najczęściej spotykane liniowe kody korekcyjne.

Kod powtórzeniowy jest najprostszym możliwym kodem liniowym i polega na wysłaniu danych kilka razy z rzędu. Dzięki temu można po otrzymaniu zakłóconego wektora, przy pomocy zasad prawdopodobieństwa, ocenić jak wygląda poprawna wiadomość. Kod powtórzeniowy nie znalazł szerokiego zastosowania praktycznego ponieważ, w porównaniu z innymi

[19] iloczynem macierzy $A = [a_{jk}]_{m\times p}$ i $B = [b_{ki}]_{p\times n}$ nazywamy taką macierz $C = [c_{ji}]_{m\times n}$, że dla dowolnych $j = 1, 2, .., m$ i $i = 1, 2, .., n$ prawdą jest, że $c_{ij} = a_{jk} \cdot b_{ki}$ [8]

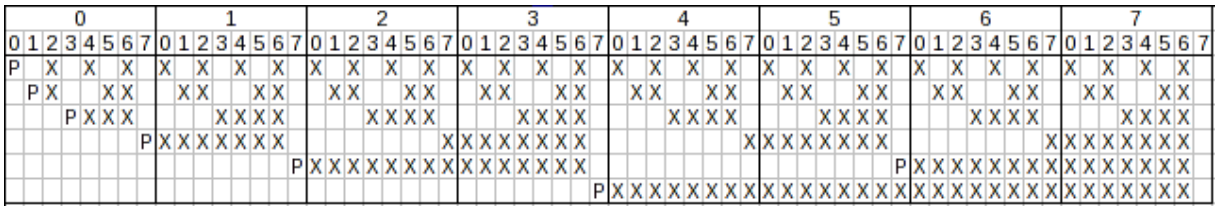
[20] wektor można rozpatrywać jako macierz jednokolumnową

[21] transpozycja macierzy polega na zamianie jej kolumn na wiersze i wierszy na kolumny – tzn. jeśli $A = [a_{ij}]$, to $A^T = [a_{ji}]$

kodami korekcyjnymi, posiada niewielkie możliwości korekcji oraz duży narzut informacji nadmiarowej. Jego zaletą jest bardzo prosty mechanizm kodowania.

Kod Hamminga jest kodem o minimalnej odległości Hamminga równej 3 oraz możliwości detekcji i korekcji pojedynczego zakłócenia, bądź też detekcji podwójnego zakłócenia, bez możliwości korekcji. Kod ten jest tzw. kodem doskonałym dla kanałów binarnych, co oznacza, że koryguje maksymalną liczbę błędów, jaka jest możliwa przy danym narzucie informacji nadmiarowej [11]. Kody Hamminga znalazły zastosowanie m. in. przy korekcji danych w pamięci operacyjnej komputerów. Mogą one mieć różną długość[22], ale zawsze mają zdolność skorygowania tylko jednego bitu w wektorze kodowym.

Sposób kodowania opiera się na tej samej zasadzie niezależnie od długości kodu. Warto zwrócić uwagę na to, że rozkład bitów parzystości w kodzie Hamminga jest bardzo regularny i rzednie wraz z liczbą bitów. Na rys. 2.2, przedstawiono schematycznie sposób tworzenia kodu Hamminga. Bity kontrolne oznaczone przez „P", sprawdzają występujące w danej linii bity oznaczone przez „X" i występują na pozycjach będących wielokrotnościami liczby 2. Jeśli przyjmiemy, że kod ma długość $n = 2^N - 1$, to m-ty[23] bit kontrolny „sprawdza" $2^{N-1} - 1$ bitów, rozłożonych w charakterystycznych grupach o długości 2^{m-1}, oddzielonych od siebie 2^{m-1} bitami[24].



Rysunek 2.2. Schemat kodowania kodem liniowym Hamminga (63,57).

Kody LDPC (od ang. *low-density parity-check*) pozwalają na przesłanie kanałem transmisyjnym prawie maksymalnej ilości danych, ograniczonej fizycznymi możliwościami[25]. Idea kodowania opiera się na pojęciu macierzy rzadkich[26].

Turbo kody łączą cechy kodowania blokowego i splotowego. Konkurują pod względem wydajności z kodami LDCP, ale ich efektywna implementacja jest dość złożona.

[22] wynoszącą $2^x - 1$, gdzie x jest liczbą naturalną, większą od 1

[23] $m \in \{1, .., N\}$

[24] wyłączając pierwszą grupę, która zawiera bit kontrolny, por. rys. 2.2

[25] zgodnie z twierdzeniem Shannona-Hartleya, maksymalna przepustowość kanału komunikacyjnego C zależy od szerokości pasma B i stosunku mocy szumów N do mocy użytecznej sygnału S, zgodnie ze wzorem: $C = log_2(1 + \frac{S}{N})$ [18]

[26] tj. takich, w których większość elementów ma wartość zero

2.4.2.2. Kody cykliczne

Największe zastosowanie w telekomunikacji znalazły kody cykliczne. Pierwszą przyczyną tego stanu rzeczy jest możliwość konstrukcji kodu cyklicznego o zadanych parametrach, w sposób efektywny, w oparciu o algebraiczne pojęcia takie jak pierścienie wielomianów i ciała Galois. Drugą przyczyną jest łatwość sprzętowej implementacji koderów i dekoderów, z wykorzystaniem rejestrów przesuwnych ze sprzężeniem zwrotnym [11].

Kod korekcyjny jest kodem cyklicznym, jeśli każdy wektor kodowy po przesunięciu cyklicznym daje inny wektor kodowy tego samego kodu. Najczęściej wektory kodów cyklicznych opisuje się przy pomocy wielomianów. Z uwagi na binarną interpretację danych w procesorze, do kodowania danych używane są wielomiany rozpięte nad ciałem binarnym – $GF(2)$. Przyjmuje się, że ciało takie zawiera dwa elementy: 0 i 1, a działania w tym ciele (dodawanie i mnożenie) są działaniami modulo 2, tak więc dodawanie i odejmowanie są równoważne i można je zrealizować przy pomocy operacji XOR[27]. Poniżej wyjaśniono jak operacje na wielomianach, niezbędne do implementacji kodów korekcyjnych, znajdują przełożenie na operacje w procesorze.

Zapis wielomianu w postaci ciągu bitów, w przypadku wielomianu nad ciałem $GF(2)$, polega na zapisie współczynników przy kolejnych potęgach zmiennej. Np. wielomian $x^5 + x^3 + x^2 + 1$ po zapisaniu w 8-bitowej zmiennej miałby postać 00101101.

Mnożenie wielomianów przez x^n odpowiada operacji przesunięcia ciągu bitów o *n* miejsc w lewo (ku bardziej znaczącym bitom).

Dodawanie wielomianów polega (podobnie jak odejmowanie) na przeprowadzeniu operacji XOR na dwóch ciągach bitów.

Dzielenie wielomianów jest operacją bardziej skomplikowaną. Składa się z następujących kroków:

1. Obliczana jest różnica między stopniami wielomianów r.
2. Dzielnik mnożony jest przez x^r.
3. Jeśli dzielnik jest większy od dzielnej, to na r-tej pozycji ilorazu wpisywane jest 0 i należy przejść do kroku 5. W przeciwnym wypadku wpisywane jest 1 i należy przejść do kolejnego kroku.
4. Dzielnik jest dodawany do dzielnej. Suma staje się nową wartością dzielnej.
5. Jeśli $r > 0$, to należy je zmniejszyć i przejść do kroku 2. W przeciwnym wypadku dzielenie dobiegło końca. Ostatnia wartość dzielnej jest resztą z dzielenia.

[27] tzw. alternatywa wykluczająca, oznaczana przez $\oplus$, zdefiniowana przy pomocy następującej tablicy prawdy:

p	q	p $\oplus$ q
0	0	0
0	1	1
1	0	1
1	1	0

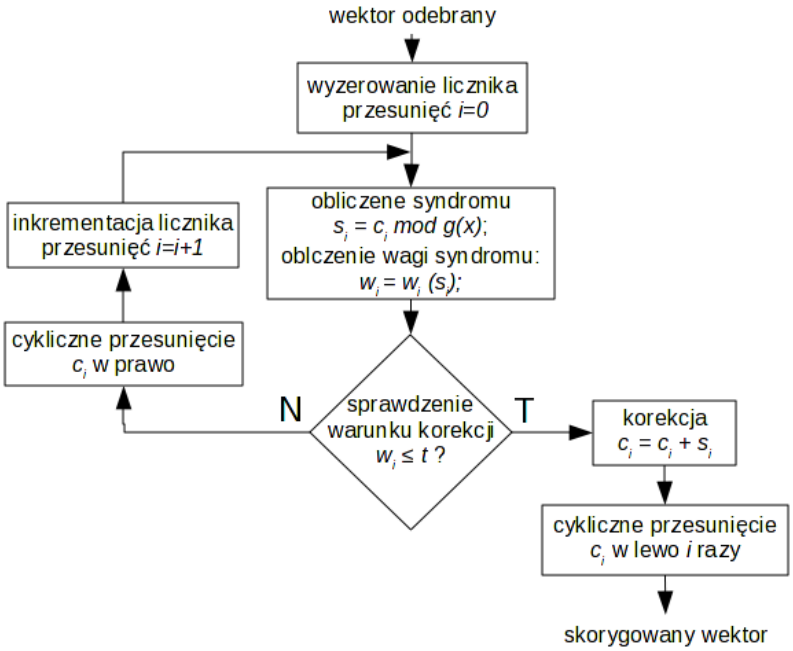
Mając wyobrażenie o sposobie implementacji poszczególnych działań na wielomianach, można przystąpić do omówienia sposobu kodowania, dekodowania i korekcji błędów w kodach cyklicznych. Do implementacji kodów cyklicznych wykorzystany został, opisany w [11], algorytm dekodowania z wyłapywaniem błędów.

Kluczowym elementem przy wszystkich operacjach jest wielomian generujący dany kod. Informacja nadmiarowa w wektorze kodowym jest bowiem resztą z dzielenia wielomianu zawierającego informację przez wielomian generujący. Aby zakodować dane należy:

1. Wielomian reprezentujący kodowane informacje pomnożyć przez x^{n-k}, gdzie $n-k-1$ jest stopniem wielomianu generującego.
2. Powstały w ten sposób wielomian stopnia n podzielić przez wielomian generujący.
3. Resztę z tego dzielenia dodać do dzielnej – w ten sposób powstaje słowo kodowe.

Każde słowo kodowe ma tę właściwość, że dzieli się bez reszty przez wielomian generujący.

Po odebraniu zakłóconego wektora kodowego, wyróżnia się dwa etapy postępowania [11], przedstawione na rysunku 2.3:



Rysunek 2.3. Schemat algorytmu dekodowania (za [11]).

Obliczanie syndromu i jego interpretacja. Najpierw następuje podzielenie otrzymanego wektora przez wielomian generujący. Jeśli reszta z tego dzielenia wynosi 0, to otrzymany wektor jest istniejącym wektorem kodowym – część informacyjna (ważniejsze k bitów) może zostać odczytana. Jeśli reszta z dzielenia nie jest równa 0, ale nie przekracza zdolności korekcyjnej kodu t, to błędy w transmisji wystąpiły, ale jedynie w części kodowej wektora. Jeśli reszta przekracza zdolność korekcyjną kodu, to należy przejść do drugiego etapu.

Przesunięcie cykliczne kodu. Jeśli syndrom wskazuje na błędy w transmisji, których nie można skorygować, to może to oznaczać jedną z dwóch sytuacji:

- Zakłócenia wystąpiły w części informacyjnej wektora – po przesunięciu cyklicznym zakłócenia mogą się przesunąć do części kodowej, co pozwoli odzyskać dane. Nie jest to jednak pewne – może się zdarzyć, że błędy są położone w odległościach uniemożliwiających ich zgromadzenie w części kodowej.
- Zakłócenia przekraczają zdolność korekcyjną kodu – wówczas niezależnie od liczby przesunięć nie da się odzyskać informacji.

Po cyklicznym przesunięciu wektora, następuje powtórne obliczenie syndromu i jego interpretacja.

Istnieją skomplikowane algorytmy, zależne od rodzaju wielomianu generującego, pozwalające szybko odczytać dane z zakłóconych wektorów kodowych, przy minimalnym nakładzie obliczeń, dzięki wnikliwej analizie syndromu. Są one wydajniejsze niż sprawdzanie wszystkich przesunięć wektora otrzymanego, aż do uzyskania odpowiednio małego syndromu lub wyczerpania możliwości.

Poszczególne kody cykliczne różnią się od siebie wielomianami generującymi. Długość tego wielomianu jest równa ilości bitów nadmiarowych dodawanych do wektora informacyjnego. Ponadto właściwości wielomianu generującego decydują o możliwościach korekcyjnych i detekcyjnych danego kodu. Ma to na tyle znaczenie, że wyróżnia się różne cykliczne kody korekcyjne o rozmaitych właściwościach.

Kody Golaya mogą być implementowane przy pomocy wielomianu generującego $x^{11} + x^{10} + x^6 + x^5 + x^4 + x^2 + 1$ i mają ściśle określone długości: (24,12) lub (23,12).

Kody BCH (skrót pochodzi od nazwisk autorów: Bose, Ray-Chaudhuri i Hocquenghem) wymagają mniejszej złożoności obliczeniowej niż inne kody, a szczególnie dobre właściwości wykazują przy dużych długościach bloków [18].

W bardzo silnie zakłóconych kanałach używa się kodów **Hadamarda**, z powodu wysokiego narzutu informacji nadmiarowej, a przy korekcji tzw. błędów grupowych[28] znalazły zastosowanie kody **Reeda-Solomona**.

2.4.2.3. Przeplot danych

Dodatkowym zabezpieczeniem transmisji może być tzw. przeplot. Polega on na umieszczeniu w wysyłanym pakiecie fragmentów różnych komunikatów, z różnych słów kodowych. Wymaga to zgromadzenia odpowiedniej ich liczby i podzielenia na fragmenty. Po odbiorze, fragmenty danych są ze sobą ponownie łączone i następuje proces dekodowania. Jeśli któryś z pakietów nie został odebrany lub jeśli wystąpiły tzw. zakłócenia blokowe (powodujące utratę wielu bitów położonych obok siebie) wciąż możliwe jest zdekodowanie wszystkich danych, ponieważ w ramach pojedynczego komunikatu uszkodzeniu uległy pojedyncze bity.

[28] polegających na przekłamaniach kilku bitów położonych obok siebie

3. Realizacja sprzętowa bezprzewodowego łącza cyfrowego

Do realizacji łącza cyfrowego transmisji danych wykorzystano układy firmy Atmel: AT91SAM7S i AT86RF211. Istnieje kilka sposobów połączenia tych dwóch układów w jedno urządzenie. Oddzielnie należy rozpatrywać połączenie wyprowadzeń do przesyłu danych i do konfiguracji urządzenia nadawczo-odbiorczego.

3.1. Konfiguracja urządzenia AT86RF211

Konfiguracja urządzenia nadawczo-odbiorczego następuje poprzez zapis odpowiednich wartości do rejestrów konfiguracyjnych. Wszystkich rejestrów jest trzynaście, jednak część z nich dotyczy pracy w nieużywanym trybie *Wake-up*[1]. Następujących siedem rejestrów wymaga skonfigurowania:

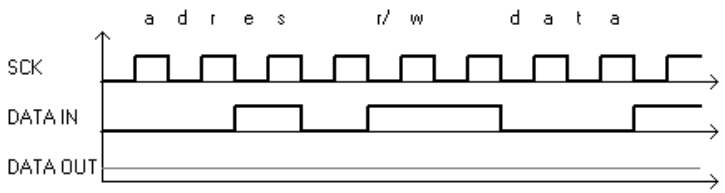
F0, F1, F2, F3 to rejestry do których zapisywane są wielkości odpowiadające za częstotliwość fali nośnej oraz jej modulację. Jak podaje dokumentacja producenta, zawartości rejestrów nie są w prosty sposób powiązane z wartościami, które regulują [2].

CTRL1 to główny rejestr konfiguracyjny układu. Przechowuje dane o tym w jakim trybie pracuje układ (nadawanie/odbiór), jaka jest moc nadawania, o jakiej mocy sygnał odebrano, czy połączenie z mikrokontrolerem jest synchroniczne czy asynchroniczne oraz kontroluje pracę w trybie *Wake-up*.

RESET – po zapisaniu do tego rejestru jedynki, we wszystkich rejestrach konfiguracyjnych pojawiają się wartości domyślne.

CTRL2 jest użyteczny tylko w przypadku zastosowania transmisji synchronicznej. Przechowuje wartość częstotliwości zegara taktującego, odtwarzanego na wyjściu DCLK przy odbiorze sygnału (opcja Clock Recovery).

Zapis do rejestrów odbywa się w sposób przedstawiony na rysunku 3.1.

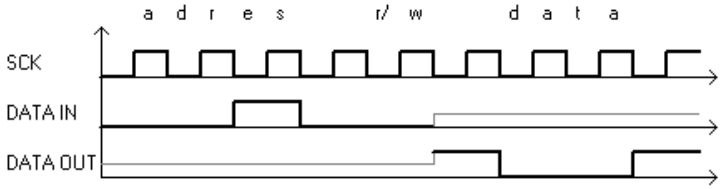


Rysunek 3.1. Przebieg czasowy operacji zapisu do rejestru AT86RF211.

[1] tryb *Wake-up* służy do oczekiwania w trybie uśpienia na wiadomość z innych urządzeń; taka opcja powinna być zaimplementowana wraz z siecią czujnikową

Najpierw przesyłane są cztery bity adresu, następnie bit ustalający kierunek dalszej transmisji – odczyt lub zapis. W przypadku operacji zapisu danych do rejestru, przesyłane są kolejno bity, począwszy od najważniejszego. Dopuszczalny jest częściowy zapis rejestru, wówczas mniej ważne bity nie ulegają zmianie.

Operacja odczytu, przedstawiona na rysunku nr 3.2, przebiega podobnie. Dodatkowym utrudnieniem jest jednak konieczność natychmiastowej zmiany kierunku transmisji po wysłaniu bitów adresu i bitu odczytu.



Rysunek 3.2. Przebieg czasowy operacji odczytu z rejestru AT86RF211. Sygnały DATA IN i DATA OUT są przesyłane tym samym wyprowadzeniem.

Dysponując peryferiami układu AT91SAM7S, można wykonać komunikację z układem nadawczo-odbiorczym w celu jego konfiguracji, na dwa sposoby:

- wykorzystując porty PIO (Programmed Input/Output),
- wykorzystując interfejs SPI (Serial Peripheral Interface).

W każdym z nich konieczne jest podłączenie następujących końcówek:

- SCK, przez którą odbierany lub nadawany jest sygnał taktujący,
- SDATA, służącej do przesyłania danych w takt sygnału SCK,
- SLE, którą przesyłany jest sygnał zezwolenia na transmisję (od ang. *enable*).

3.1.1. Porty PIO

Przy tym podejściu niezbędne jest ręczne opisanie każdego przesyłanego sygnału. Trzeba w związku z tym wykorzystać czasomierz (od ang. *timer*) układu AT91SAM7S do generacji sygnału na końcówkę SCK. Gdy konfiguracja jest odczytywana, port danych (SDATA) przełączany jest z funkcji portu wyjściowego na port wejściowy.

Jak sprawdzono, transmisja przy pomocy tego sposobu komunikacji trwa dłużej, niż przy pomocy interfejsu SPI. Wynika to z powolnego przełączania stanu portów PIO z wysokiego na niski lub odwrotnie. Transmisje są jednak na tyle krótkie[2], że częstotliwość sygnału zegarowego jaką udało się uzyskać, czyli 50 kHz, jest wystarczająca do sprawnej komunikacji.

[2] wystarczy przesłać kilkaset bitów, aby skonfigurować cały układ

3.1.2. Interfejs SPI

Między sposobem przesyłania danych do konfiguracji układu nadawczo-odbiorczego i sposobem funkcjonowania protokołu SPI istnieje podobieństwo, polegające na przesyłaniu danych synchronicznie z wykorzystaniem sygnału zezwolenia. Istnieją jednak dwie główne przeszkody w implementacji tego rozwiązania. Pierwsza to konieczność szybkiego przełączenia interfejsu SPI z trybu nadawania do trybu odbioru, w przypadku przesłania 4 bitów adresu i bitu odczytu. Druga to konieczność podziału przesyłanego komunikatu na ośmiobitowe ciągi, ponieważ takimi operuje interfejs SPI.

Obie przeszkody pokonano pisząc funkcję *ReadWriteReg*, która wykorzystuje właściwość interfejsu SPI, polegającą na tym, że transmisja zachodzi zawsze obustronnie. Urządzenie nadrzędne (*master*) przesyła do urządzenia podrzędnego (*slave*) dane szyną MOSI, podczas gdy szyną MISO przesyłane są dane w odwrotnym kierunku. Zmiana kierunku transmisji może więc nastąpić bez potrzeby rekonfiguracji interfejsu SPI.

Funkcja *ReadWriteReg* nie może oddzielać bitów adresu i bitu wskazującego kierunek transmisji od danych, gdyż wszystkie bity muszą zostać przesłane lub odebrane jeden po drugim. W przypadku odczytu rejestru konfiguracyjnego, po przełączeniu kierunku transmisji, trzy ostatnie bity pierwszego 8-bitowego komunikatu i wszystkie kolejne odczytane bity, zapisywane są do zmiennej typu *unsigned long* i zwracane na zakończenie działania. Przy odczycie rejestrów konfiguracyjnych urządzenia nadawczo-odbiorczego dane te są zawartością rejestru, natomiast przy zapisie, są to te same dane, które wysłano. Końcówki MOSI i MISO są bowiem zwarte do jedynego wyprowadzenia urządzenia nadawczo-odbiorczego AT86RF211 służącego do konfiguracji – do końcówki SDA. Do zliczania przesłanych bitów zostało wykorzystane przerwanie zewnętrzne. Port IRQ1 został zwarty z sygnałem taktującym transmisję. Na każdym zboczu narastającym sygnału inkrementowana jest wartość licznika, a w przypadku, gdy wzrośnie ona do 5 i dane mają zostać odczytane, następuje przełączenie kierunku transmisji.

Komunikacja z użyciem SPI jest trudniejsza w implementacji, ale pozwala na przesył danych z częstotliwością sygnału taktującego rzędu 200kHZ. W układzie AT91SAM7S znajduje się również drugi interfejs SPI, wykorzystujący inne wyprowadzenia, więc wciąż możliwa jest komunikacja z urządzeniami zewnętrznymi. Do dyspozycji pozostaje też drugie przerwanie zewnętrzne, które może okazać się przydatne do połączenia z czujnikiem lub do przełączania układu nadawczo-odbiorczego w tryb uśpienia.

3.2. Fizyczna warstwa komunikacji

Przesył danych pomiędzy układem nadawczo-odbiorczym a mikrokontrolerem następuje przy pomocy wyprowadzeń DCLK i DATAMSG. Pierwszym z nich przesyłany jest w transmisji synchronicznej sygnał taktujący dane, a drugim przesyłane są same dane. Spośród peryferiów układu AT91SAM7 należało wybrać to, które umożliwi najszybszą komunikację. Specyfika

działania urządzenia nadawczo-odbiorczego AT86RF211 sprawia, że istotnych jest kilka przesłanek przydatnych przy wyborze:

- Ze względu na jedno połączenie elektryczne istnieje możliwość przesłania tylko jednego sygnału – interfejsy, które wykorzystują w swoim działaniu linię zezwolenia *enable*, nie mogą zostać użyte.
- Pomimo możliwości przesyłania tylko jednego sygnału można wykorzystać interfejsy, które wymagają sygnału zegarowego. Układ AT86RF211 dysponuje bowiem opcją odzyskiwania sygnału zegarowego na podstawie nadsyłanych danych[3] – pojawia się on na porcie DATACLK. Istnieją jednak pewne ograniczenia w używaniu tej opcji. Częstotliwość sygnału DATACLK jest konfigurowana w rejestrach urządzenia, a jej maksymalna wartość wynosi 64 kb/s.
- Przy poprawnie skonfigurowanym urządzeniu nadawczym AT86RF211 port DATAMSG, z punktu widzenia programisty, zwarty jest z analogicznym portem urządzenia odbiorczego.

Układ AT91SAM7S dysponuje 4 różnymi interfejsami komunikacyjnymi (por. [3]). Dodatkowo istnieje możliwość utworzenia własnego interfejsu z wykorzystaniem portów PIO, podobnie jak to miało miejsce w przypadku komunikacji z układem w celu jego konfiguracji.

SPI (Serial Peripheral Interface) wymaga sygnału zezwolenia, którego nie da się przesłać drogą radiową z wyjaśnionych wyżej powodów. Nie jest więc możliwe wykorzystanie tego interfejsu. Ponadto jeden z dwóch interfejsów SPI został już wykorzystany do konfiguracji. Gdyby do komunikacji wykorzystany został drugi, niemożliwa byłaby już komunikacja z innymi urządzeniami peryferyjnymi wykorzystującym tego typu interfejs – np. kartami pamięci lub czujnikami.

TWI (Two-wire Interface) po każdej wiadomości wymaga zwrócenia, przez odbiorcę wiadomości, bitu potwierdzenia ACK. Przełączanie kierunku transmisji co 8 bitów jest jednak bardzo czasochłonne – sama rekonfiguracja urządzenia nadawczo-odbiorczego zajmuje bowiem ok. $200\mu s$ [2].

SSC (Synchronous Serial Controller) również nie może być wykorzystany w projektowanym urządzeniu. Warunkiem rozpoczęcia transmisji jest bowiem zmiana stanu linii TF a nie można jej przesłać drogą radiową.

USART (Universal Synchronous Asynchronous Receiver Transmitter) nadaje się do przesyłania danych przy pomocy układu AT86RF211, zarówno w trybie asynchronicznym jak i synchronicznym, przy wykorzystaniu funkcji odzyskiwania sygnału zegarowego.

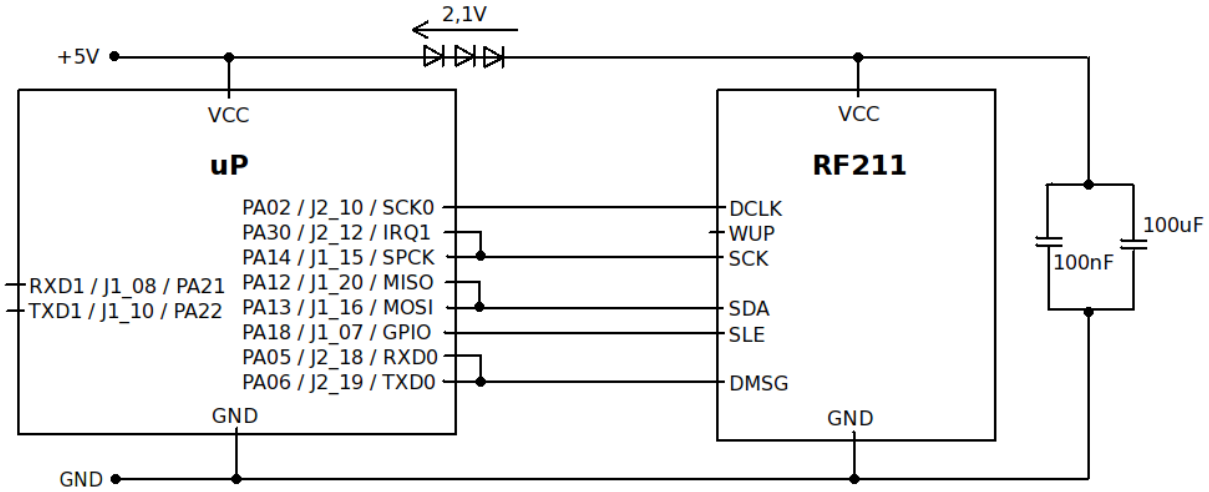
własny interfejs zaimplementowany przy użyciu portu PIO jest możliwy do zrealizowania, ale nie różniłby się wydajnością od gotowego interfejsu USART.

[3] jest to funkcja *clock recovery* (ang.) – odzyskiwanie sygnału zegarowego

Jak wynika z powyższych rozważań najwłaściwszym interfejsem, który można wykorzystać do komunikacji z układem nadawczo-odbiorczym AT86RF211, jest interfejs USART.

3.3. Połączenie AT86RF211 z AT91SAM7S

Układ AT86RF211 ma pięć wyprowadzeń, istotnych z punktu widzenia komunikacji z otoczeniem. Do konfiguracji rejestrów (tak jak to opisano w p. 3.2) używane są trzy z nich: SCK, SDA oraz SLE. Zdecydowano, że do konfiguracji używany będzie protokół SPI, ze względu na szybsze działanie niż interfejs z portów PIO. Powinno zostać wykonane elektryczne połączenie odpowiednich wyprowadzeń peryferiów układu AT91SAM7S z układem nadawczo-odbiorczym. Sygnały SCK i SLE powinny być sterowane przez mikroprocesor, co ma miejsce w przypadku skonfigurowania SPI do pracy w trybie *master*. Schemat połączeń elektrycznych na płycie głównej urządzenia przedstawiono na rysunku 3.3.



Rysunek 3.3. Schemat płyty głównej urządzenia.

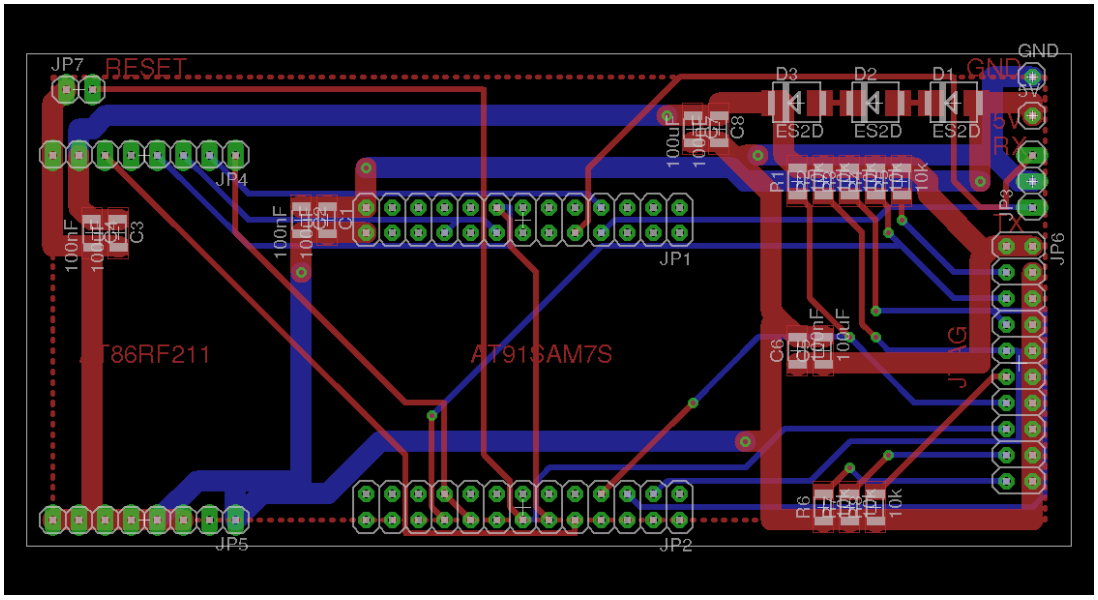
Wyprowadzenia MOSI (od ang. *Master Out Slave In*), MISO (od ang. *Master In Slave Out*) i SDA są zwarte jednym połączeniem elektrycznym, aby umożliwić szybkie przełączanie kierunku transmisji. Wyprowadzenie MISO jest skonfigurowane jako wejściowe, znajduje się więc w stanie wysokiej impedancji i nie może zakłócić danych wychodzących z mikroprocesora wyprowadzeniem MOSI. Jednak wyprowadzenie MOSI jest skonfigurowane jako wyjściowe i interfejs SPI wysyła nim dane również w trakcie odbioru, ponieważ transmisja tym interfejsem zawsze zachodzi obustronnie. Gdyby przy tej konfiguracji odbierane były dane z układu AT86RF211, to zostałyby one zakłócone przez sygnał z portu MOSI. Aby tej sytuacji uniknąć, zmieniona została konfiguracja portu PIO, do którego podłączone jest wyprowadzenie MOSI. Port ten zostaje skonfigurowany jako wejściowy w sytuacji, gdy dane przesyłane są do mikroprocesora (czyli przy odczycie rejestrów). Oznacza to, że port znajduje się w stanie wysokiej impedancji i nie zakłóca nadchodzących danych.

Analogiczny problem zaistniał przy podłączaniu wyprowadzeń odpowiadających za komunikację szeregową mikroprocesora do końcówki DATAMSG układu

nadawczo-odbiorczego. Znów potrzebne było połączenie elektryczne trzech wyprowadzeń, z których dwa, umieszczone na płytce drukowanej układu AT91SAM7S, były używane na zmianę: jedno z nich jako wejściowe, a drugie jako wyjściowe. Zastosowano takie samo rozwiązanie, polegające na rekonfiguracji portu wyjściowego na wejściowy, w przypadku nadchodzącej transmisji.

Na rysunku 3.3, oprócz wymienionych wyżej połączeń, widać także połączenie końcówki DCLK układu nadawczo-odbiorczego z końcówką SCK0, odpowiadającej w układzie mikroprocesora za taktowanie transmisji synchronicznej. Połączenie zostało utworzone w celu analizy różnic pomiędzy transmisją synchroniczną i asynchroniczną.

Roboczym rozwiązaniem problemu różnych napięć zasilania obu układów (potrzeby zapewnienia 3V i 5V) może być prosty układ obniżający napięcie o ok. 2V, zrealizowany poprzez wstawienie trzech diod (spolaryzowanych w kierunku przewodzenia) pomiędzy wyprowadzenie zasilania układu nadawczo-odbiorczego a złącze doprowadzające napięcie do płytki.



Rysunek 3.4. Projekt płyty głównej urządzenia, wykonany w programie Eagle 5.9.0.

Projekt ostatecznej wersji układu, widoczny na rysunku 3.3 został zrealizowany przy użyciu Eagle i wykonany jako płytka PCB (od ang. *Printed Circuit Board*) wyposażona w wyprowadzenia zasilania 5V, układ eliminujący zakłócenia z linii masy[4], układ obniżający napięcie do 3V, wyprowadzone połączenia do złącza JTAG (umożliwiającego programowanie mikroprocesora) oraz wyprowadzone połączenia do układu USART (służącego do komunikacji z komputerem). Schemat płyty głównej urządzenia przedstawiono na rysunku 3.4.

[4] Oba układy, używając wspólnej masy, silnie się zakłócają. Dzięki wcześniejszym eksperymentom przeprowadzanym w Instytucie Łączności od razu zastosowano uziemienie szumów na liniach masy i zasilania przy pomocy dwóch kondensatorów o pojemnościach $100\mu F$ i $100nF$.

3.4. Cyfrowe łącze transmisji radiowej

Komunikacja radiowa, na poziomie sprzętu, zaczyna się w chwili wpisania bajtu danych do rejestru nadawczego interfejsu USART. Dane te są następnie bit po bicie podawane na wyprowadzenie wyjściowe układu AT91SAM7S, połączone z wyprowadzeniem wejściowym DATAMSG[5] układu AT86RF211. Na końcówce tej przez pewien czas[6] utrzymywany jest niski (0V) lub wysoki (5V) poziom napięcia, w zależności od tego czy bit jest zerem czy jedynką.

Układ nadawczo-odbiorczy, w zależności od tego czy na końcówce wejściowej znajduje się poziom niski czy wysoki, moduluje w różny sposób falę nośną. W układzie AT86RF211 wykorzystywana jest modulacja częstotliwościowa FSK (od ang. Frequency-shift keying). Częstotliwość fali nośnej wynosi 433 MHz, 868 MHz lub 915 Mhz i może być zwiększana lub zmniejszana o ustaloną wartość 10 kHz, 20 kHz lub 50 kHz, w zależności od zawartości rejestrów konfiguracyjnych urządzenia. Konfiguracja została domyślnie ustawiona na 433 MHz z modulacją $\pm$50kHz, aby zapewnić najszerszy kanał komunikacyjny i wykorzystać najszybszą transmisję danych. Jeżeli więc na końcówce wejściowej ustalony jest poziom niski, to układ nadawczo-odbiorczy wysyła falę elektromagnetyczną o częstotliwości 433,05 MHz a jeśli poziom wysoki – 432,95 MHz.

Drugi układ nadawczo-odbiorczy, skonfigurowany do odbioru, otrzymuje jedną z dwóch zmodulowanych fal i po wykryciu jak jest ona zmodulowana, natychmiast wystawia poziom niski lub wysoki na końcówkę DATAMSG (tym razem skonfigurowaną jako wyjście układu AT86RF211 i wejście interfejsu USART układu AT91SAM7S). Na tej podstawie zostaje wypełniony rejestr odbiorczy interfejsu USART w układzie odbiorczym. Bajt danych z rejestru nadawczego układu nadawczego może więc zostać odczytany w rejestrze odbiorczym układu odbiorczego.

[5] Końcówka DATAMSG jest portem wejściowym, kiedy układ jest skonfigurowany do nadawania, a portem wyjściowym, kiedy układ jest skonfigurowany do odbioru.

[6] w przypadku transmisji z częstotliwością 56 kb/s każdy bit wystawiany jest na wyprowadzenie na ok. $20\mu s$

4. Oprogramowanie osadzone w systemie czasu rzeczywistego

4.1. System czasu rzeczywistego FreeRTOS

Procesor jednordzeniowy jest w stanie wykonywać na raz jedną operację. Powoduje to, że programy pisane na jednordzeniowe procesory mają strukturę liniową, w której zadania wykonywane są jedno po drugim. Na przykład w programie przetwarzającym i wysyłającym dane można je najpierw w całości przetworzyć, a potem wysłać. Występuje przy tym strata czasu i mocy obliczeniowej procesora podczas oczekiwania na wysłanie kolejnych bajtów. Lepiej jest napisać program tak, aby kolejne dane były przetwarzane w czasie oczekiwania na wysłanie poprzednich.

Można to zrealizować przy pomocy przerwań i okresowego sprawdzania gotowości urządzeń peryferyjnych. Jednak lepszym sposobem zrównoleglenia zadań jest zastosowanie systemu operacyjnego, który zarządza wątkami[1] wykonywanymi w programie i w miarę potrzeby rozdziela pomiędzy nie czas procesora. Takim właśnie narzędziem jest FreeRTOS (Free Real-time operating system).

4.1.1. Zasada działania

System FreeRTOS jest zaprojektowany tak, aby można go było zaimplementować na procesorach o małej mocy obliczeniowej. Posiada przez to ograniczoną funkcjonalność i wymaga zaawansowanej wiedzy. Jego działanie, z punktu widzenia programisty, ogranicza się do uruchomienia wątków wykonujących program oraz *schedulera*[2], a następnie blokowanie jednych i odblokowywanie innych wątków, zgodnie z ustalonymi wcześniej priorytetami i aktualnymi potrzebami wątków.

Wątek w systemie czasu rzeczywistego powinien reprezentować część funkcjonalności programu, która może działać jednocześnie z inną (np. kodowanie może zachodzić w trakcie wysyłania danych). Nie ma sensu tworzyć osobnych wątków do zadań, które wzajemnie się wykluczają i nigdy nie będą zachodzić jednocześnie.

Każdy wątek ma przypisany priorytet i może znajdować się w jednym z czterech stanów [4]:

działania (*running*), kiedy jest wykonywany przez procesor – w danej chwili tylko jeden wątek może być w tym stanie,

[1] w dokumentacji angielskiej używa się określenia *task*, czyli „zadanie", jednak działanie *tasku* nie różni się znacząco od wątku – *thread*, stosowane więc będzie określenie „wątek", które jest bardziej zrozumiałe oraz pozwoli dalej używać słowa „zadanie" w potocznym rozumieniu

[2] wątek o najwyższym priorytecie, zarządzający pozostałymi wątkami

gotowości (*ready*), kiedy wątek jest gotów do pracy i oczekuje aż *scheduler* przydzieli mu dostęp do zasobów, bo w danej chwili używa ich inny wątek o takim samym lub wyższym priorytecie,

zablokowany (*blocked*), w przypadku gdy wątek oczekuje na jakieś zdarzenie – np. na pojawienie się danych w kolejce lub opuszczenie semafora,

zawieszony (*suspended*), kiedy wątkowi nie są przydzielane zasoby procesora (po wywołaniu systemowej funkcji *vTaskSuspend()*).

Wątek jest w systemie FreeRTOS implementowany jako funkcja, która nie zwraca wartości, a jako argument przyjmuje wskaźnik na zmienną *pvParameteres*, która służy do przekazywania parametrów wątkowi podczas uruchamiania. Wewnątrz wątku znajduje się pętla nieskończona, charakterystyczna dla programów pisanych na mikrokontrolery.

4.1.2. Mechanizmy komunikacji międzywątkowej

Kiedy system na raz wykonuje wiele operacji przydatny może się okazać mechanizm komunikacji między wątkami. FreeRTOS taki mechanizm dostarcza w postaci dwóch narzędzi: semaforów i kolejek.

Kolejki w systemie FreeRTOS to kolejki FIFO (od ang. *First In First Out*), choć istnieje także możliwość pisania do początku kolejki [4]. Zarówno proces pisania do kolejki jak i odczytu mogą być operacjami blokującymi. Dzięki temu, jeśli nie ma danych do odczytania, bądź nie ma miejsca w kolejce, aby dane zapisać, to wątek przejdzie w stan zablokowania i zostanie ponownie uruchomiony przez *scheduler* dopiero w przypadku możliwości kontynuacji zadania od miejsca w którym uległ zablokowaniu [4]. W programie nie wykorzystującym systemu czasu rzeczywistego najczęściej stosuje się tzw. *polling* – tj. ciągłe lub okresowe sprawdzanie czy warunki do kontynuacji zadania są spełnione, czy nie.

Funkcje *xQueueReceive* oraz *xQueueSendToBack*, służące do obsługi kolejek, jako argument wywołania otrzymują liczbę taktów systemu operacyjnego, które mogą odczekać w przypadku niemożności odczytania lub zapisania do kolejki. W szczególności, mogą one blokować wątek w nieskończoność, aż zostanie spełniony warunek dalszego wykonania programu. W przypadku, gdy po określonym czasie w wątku wciąż nie mogą być wykonywane dalsze instrukcje, funkcja blokująca zwraca informację o błędzie.

Semafory zostały zaimplementowane w systemie FreeRTOS jako jednoelementowe kolejki. Istnieją semafory binarne, które reprezentują tylko stan opuszczony lub podniesiony oraz semafory liczące, zdolne do przyjęcia większej liczby stanów [4]. Używa się ich przeważnie w sytuacjach, gdy program powinien poczekać na jakieś zdarzenie nie związane z przepływem danych.

Czasami warto semafor zastąpić jednoelementową kolejką, gdyż oprócz stanu można przy jej pomocy przekazać dodatkową informację. Takie rozwiązanie zostało wykorzystane przy

uruchamianiu wątków *vAT86Application*[3] i *vAT86Communication*[4]. Otrzymują one wówczas informację o trybie w jakim powinny działać.

4.2. Model OSI i jego zastosowanie

Model OSI (od ang. *Open System Interconnection*) jest dziś powszechnie stosowany w większości protokołów komunikacji zarówno przewodowej jak i bezprzewodowej. Powstał na początku lat 80, aby umożliwić różnym producentom wytwarzanie współpracujących ze sobą urządzeń. Model składa się z siedmiu warstw, pełniących różne funkcje w komunikacji.

Wyróżniamy następujące warstwy modelu (począwszy od najwyższej, za [14]):

Warstwa aplikacji zajmuje się poszczególnymi aplikacjami, dostarczając im podstawowych usług komunikacyjnych.

Warstwa prezentacji odpowiedzialna jest za zachowanie odpowiedniego formatu danych, zgodnego z reprezentacją systemu docelowego.

Wartwa sesji odpowiada za konfigurację, otwarcie i ew. taryfikację połączenia między użytkownikami.

Warstwa transportowa przetwarza dane otrzymane z warstwy sesji, dzieląc ją na mniejsze bloki. Jest też odpowiedzialna za rozdzielanie połączenia pomiędzy kilka warstw sieciowych. Powinna być w pełni niezależna od sprzętu.

Warstwa sieciowa posiada informacje o topologii sieci i decyduje, którym połączeniem przesłać dane.

Warstwa łącza danych odpowiada za odbiór i przesyłanie opakowanych w ramki danych w taki sposób, aby nie zawierały one błędów. Reguluje także szybkość transmisji i synchronizuje ją.

Warstwa fizyczna przekłada dane na sygnały oraz przesyła je fizycznym połączeniem.

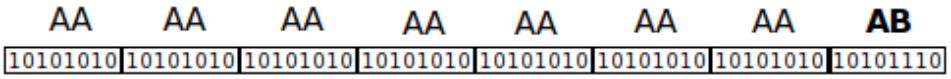
W niniejszej pracy wykorzystana została uproszczona wersja modelu OSI. Warstwa fizyczna musiała zostać zaimplementowana przy pomocy jednego z peryferiów układu AT91SAM7 służącego do komunikacji – wybrany został w tym celu protokół USART. Funkcje warstwy łącza danych i warstwy transportowej przejął wątek systemu operacyjnego *vAT86Communication*. Natomiast wątek *vAT86Application* realizuje funkcje warstwy aplikacji. Pozostałe warstwy modelu: warstwa prezentacji, sesji i sieciowa nie zostały zaimplementowane, jednak program jest przygotowany na ich dodanie w ramach wątku aplikacji.

Wśród gotowych rozwiązań komunikacyjnych, korzystających z modelu OSI, najbliższe stawianym wymaganiom są: protokół PPP (od ang. *Point to Point Protocol*) oraz ramka sieci *Ethernet*.

[3] nazywany też w dalszej części pracy „wątkiem aplikacji"
[4] nazywany też w dalszej części pracy „wątkiem komunikacji"

Twórcy protokołu PPP położyli duży nacisk na komunikację między węzłami sieci i uzgodnienie parametrów transmisji. Cechy te są zbędne w implementowanym, prostym systemie, podobnie jak rozbudowany system komend i faz transmisji [17]. Ostatecznie, żadne fragmenty protokołu PPP nie zostały zaimplementowane.

Standardy w technologii *Ethernet* przystosowane są do transmisji dużej ilości danych przy pomocy kabla, jednak ramka transmisyjna posiada rozbudowany, ośmiobajtowy nagłówek[5], który pozwolił rozwiązać problem odróżnienia szumu[6] od nadchodzącej wiadomości [7]. Ten pomysł został zaczerpnięty w niezmienionej postaci i jest przedstawiony na rysunku 4.1.



Rysunek 4.1. Schemat nagłówka ramki ethernetowej.

Celem pracy nie było dobranie najlepszego protokołu do tego rodzaju transmisji – przedstawione rozwiązanie jest jedynie przykładowe i może zostać zastąpione innym, dopasowanym do docelowego sposobu komunikacji bezprzewodowej.

4.3. Opis algorytmiczny

Ze względu na równoległość zachodzących w programie procesów oraz konieczność synchronizacji kilku wątków, algorytm działania programu uległ znacznej komplikacji. Należy oddzielnie rozpatrywać proces wysyłania i odbioru danych w układzie, gdyż w każdej z tych sytuacji inaczej działają poszczególne wątki i używane są inne semafory i kolejki. Można jednak dostrzec wiele analogii w obydwu procesach jak również w działaniu poszczególnych fragmentów programu.

4.3.1. Wątki

Program wykonywany jest w pięciu oddzielnych wątkach, przy czym pierwszy z nich nigdy nie jest wykonywany jednocześnie z pozostałymi czterema:

Menu zostało napisane przy pomocy instrukcji *switch* języka C i jest sterowane przez znaki przychodzące łączem RS-232.

vAT86Application odpowiada za inicjalizację procesów komunikacyjnych i jest jedynym, z którym powinien mieć kontakt użytkownik urządzenia.

vAT86Encoding[7] odpowiada za przepływ danych, wprowadzonych przez użytkownika w wątku aplikacji, do wątku *vAT86Communication* oraz zachodzące wówczas kodowanie danych.

[5] w dokumentacji angielskiej nazywany *preamble* [7]

[6] w przypadku, gdy nie ma w zasięgu urządzenia, które nadaje na zadanej częstotliwości, układ odbiorczy traktuje nadchodzący szum jak dane i po demodulacji przesyła go na końcówkę DATAMSG

[7] nazywany też w dalszej części pracy „wątkiem kodowania”

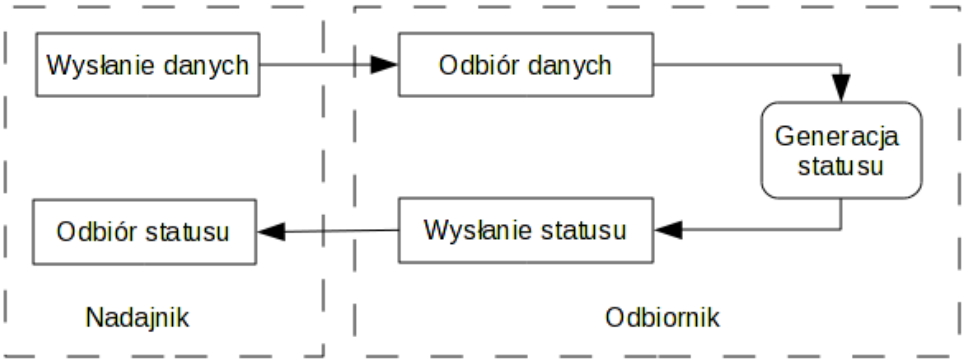
vAT86Decoding[8] pełni analogiczną rolę do *vAT86Encoding* podczas komunikacji w odwrotnym kierunku: od *vAT86Communicaton* do *vAT86Application*.

vAT86Communication służy do komunikacji z portem USART, odbioru bądź wysłania nagłówka oraz zakodowanych danych.

Wątki są ze sobą synchronizowane przy pomocy opisanych w punkcie 4.1.2 mechanizmów. Zaimplementowano 4 semafory i 10 kolejek umożliwiających płynny przepływ danych oraz powodujących blokowanie niepotrzebnych wątków. Działanie zastosowanych mechanizmów komunikacji międzywątkowej jest dokładnie opisane w p. 4.3.3 i 4.3.4.

4.3.2. Kanał główny i zwrotny

W urządzeniu zastosowano komunikację z wykorzystaniem kanału głównego i zwrotnego. Kanałem głównym przesyłane są właściwe dane, a kanał zwrotny służy do okresowego sprawdzenia poprawności transmisji. Schemat, według którego przebiega każda operacja przesyłu danych przedstawiony został na rys. 4.2.



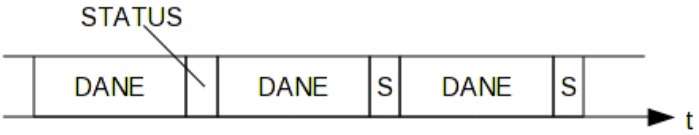
Rysunek 4.2. Ilustracja działania pojedynczej operacji przesyłu danych.

Najpierw następuje transmisja danych z nadajnika, zakodowanych jednym z kodów korekcyjnych. Liczba przesyłanych po zakodowaniu bajtów została ograniczona do 255. Odbiornik, podczas odkodowywania danych, o ile jest to możliwe, detekuje niekorygowalne błędy[9]. W zależności od tego, czy takie błędy wystąpiły czy nie, odpowiednio ustawia bity w bajcie statusu, przy pomocy funkcji *AT86CreateStatus*.

Następnie oba moduły zmieniają funkcję: moduł nadawczy, który wysłał komunikat zmienia konfigurację na odbiorczą i oczekuje na status, a moduł odbiorczy, który wcześniej odebrał dane, zmienia konfigurację na nadawczą. Następuje transmisja stworzonego wcześniej bajtu statusu do modułu, który nadał komunikat. Status, po odebraniu, jest analizowany. Jeśli wynika z niego, że transmisja przebiegła poprawnie, to następuje koniec cyklu. W przeciwnym przypadku transmisja zostaje powtórzona, aż do uzyskania potwierdzenia odbioru od modułu odbiorczego.

[8] nazywany też w dalszej części pracy „wątkiem dekodowania”
[9] kod Hamminga (63,57) nie ma takiej możliwości

Status ma długość jednego bajtu, więc jego przesłanie zajmuje stosunkowo mało czasu, w porównaniu z transmisją wielobajtowych danych. Zilustrowano to na rys. 4.3, który przedstawia transmisję kilku komunikatów na osi czasu.



Rysunek 4.3. Przesył danych przedstawiony na osi czasu.

Odbiór i nadawanie danych różnią się od odbioru i nadawania statusu na poziomie wątków kodujących/dekodujących. Wątki te są uruchamiane w różnym trybie w zależności od tego, czy powinny się zajmować kodowaniem/dekodowaniem danych czy statusu. Wątki komunikacji oraz aplikacji działają w całkowitym oderwaniu od kanału zwrotnego.

Wadą zastosowania kanału zwrotnego jest spowolnienie komunikacji między urządzeniami. Przełączenie układu AT86RF211 zajmuje około 200 μs, przesłanie jednego bajtu danych z prędkością 38400 b/s[10] również ok. 200 μs. Aby jednak mieć pewność, że przesył danych nie rozpocznie się zanim urządzenie odbiorcze zmieni konfigurację, co powodowałoby utratę danych i konieczność ich ponownego przesyłu, wprowadzone jest opóźnienie transmisji o ok. 6ms. W takim czasie mogłoby zostać przesłanych 30 bajtów danych. Jeśli więc przesyłane są krótkie pakiety danych, to zastosowanie kanału zwrotnego zmniejsza szybkość transmisji nawet kilkukrotnie. W przypadku przesyłania liczby bajtów zbliżonej do maksymalnej, wpływ okresowego przesłania bajtu statusu na czas trwania transmisji jest niewielki (ok. 10%).

4.3.3. Nadawanie danych

Proces został zilustrowany na rysunku 4.4.

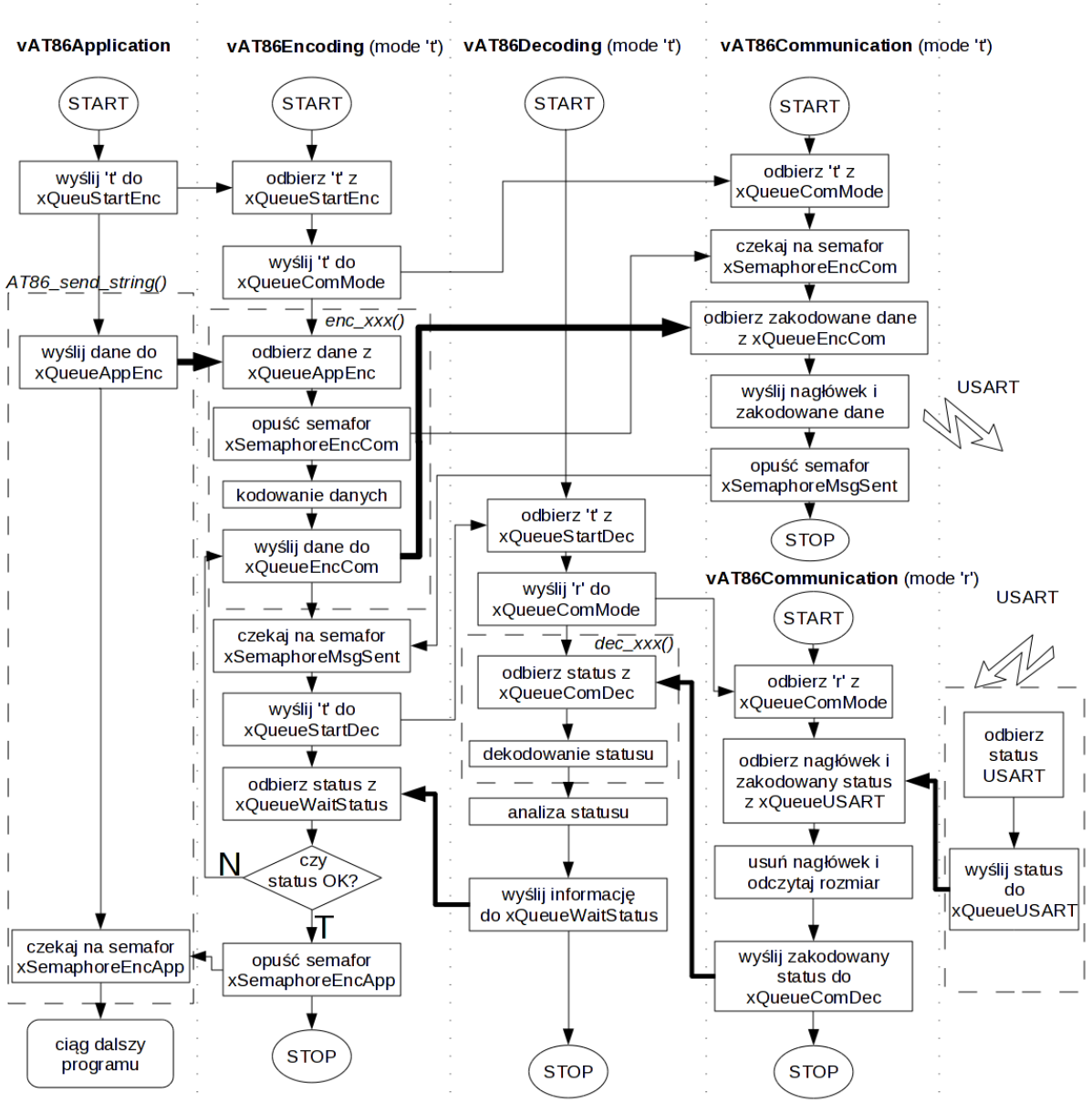
Inicjalizacja przesyłania danych następuje w wątku *vAT86Application*. Jak konkretnie ma ona miejsce, zależy od aplikacji. Dane mogą być pobierane z klawiatury, z pamięci lub ze zmiennej, ale zawsze trafiają do funkcji *AT86_send_string()*, która jako argumenty przyjmuje wskaźnik do początku tablicy zmiennych typu *char*[11] oraz długość komunikatu. Dane te trafiają do kolejki *xQueueAppEnc*, której początek znajduje się w funkcji kodującej. Ponadto do kolejki *xQueueStartEnc* zapisywany jest znak „t", wybudzając z uśpienia wątek *vAT86Encoding*. Następnie funkcja *AT86_send_string*, a przez nią także cały wątek *vAT86Application*, zostają zablokowane do czasu opuszczenia semafora *xSemaphoreEncApp*, a więc do chwili, kiedy wszystkie dane zostaną wysłane drogą radiową.

Jeszcze zanim wszystkie dane trafią do kolejki *xQueueAppEnc*, zaczyna działać wątek *vAT86Encoding*. Wchodzi w tryb wskazany przez znak odczytany z kolejki *xQueueStartEnc*,

[10] najszybsza komunikacja, przy częstotliwości fali nośnej 433 MHz z modulacją ±50kHz, przy której nie powstają przekłamania niezwiązane z zakłóceniami kanału transmisyjnego

[11] cały czas program operuje 8-bitowymi zmiennymi

a więc w przypadku transmisji danych w tryb „t” – *transmission*. Następuje bezzwłoczna konfiguracja układu nadawczo-odbiorczego vAT86RF211, gdyż potrzebuje on $200\mu s$ [2], aby przełączyć się w tryb nadawania. Następnie zostaje wysłany znak ’t’ do kolejki *xQueueComMode*, co powoduje rozpoczęcie pracy wątku *vAT86Communication*. Na koniec wywoływana jest funkcja kodująca.



Rysunek 4.4. Schemat algorytmu transmisji danych w programie. Cieńsze strzałki symbolizują przebieg programu i instrukcji sterujących, grubsze strzałki oznaczają przepływ danych lub statusu pomiędzy wątkami, linią przerywaną wydzielono zadania wykonywane wewnątrz funkcji.

To właśnie w funkcji *enc_xxx()*[12] odbierane są z kolejki *xQueueAppEnc* wszystkie dane, poczynając od rozmiaru wiadomości, który bez kodowania zostaje przekazany do wątku

[12] gdzie „xxx” to skrócona nazwa kodu korekcyjnego, np. „enc_hamlin63_57”

vAT86Communication. Długość zakodowanego komunikatu jest większa od pierwotnie wysłanego przez aplikację i należy do wątku komunikacyjnego przesłać odpowiednio wykalkulowaną wartość. Rozpoczyna się kodowanie, a każdy zakodowany bajt trafia do kolejki *xQueueEncCom* oraz do bufora *global_encoded_buffer*, na wypadek konieczności retransmisji danych. W odpowiedniej chwili do wątku *vAT86Communication* przekazywana jest informacja o gotowości danych do wysłania poprzez opuszczenie semafora *xSemaphoreEncCom*. Moment ten może zależeć od funkcji, ale powinien nastąpić jak najwcześniej, aby szybko rozpocząć relatywnie powolne, w porównaniu z innymi procesami, przesyłanie danych. Po zakończeniu kodowania funkcja zwraca liczbę zakodowanych bajtów.

Wątek *vAT86Encoding* nie może rozpocząć dalszej pracy dopóki wszystkie dane nie zostaną przesłane, oczekuje on więc na opuszczenie semafora *xSemaphoreMessageSent*.

W tym czasie dane, które trafiły w funkcji kodującej do kolejki *xQueueEncCom*, są odczytywane z drugiego jej końca, w wątku *vAT86Communication*. Wcześniej wątek nadaje ośmiobajtowy nagłówek z bajtem synchronizującym, który sygnalizuje nadejście użytecznych danych oraz koduje rozmiar informacji i także go przesyła. Po zakończeniu wysyłania komunikatu opuszczany jest semafor *xSemaphoreMessageSent* a wątek ulega zablokowaniu przy próbie odczytu z kolejki *xQueueComMode*.

Odczyt ten nastąpi dopiero kiedy wątek *vAT86Encoding* rozpocznie odbiór informacji zwrotnej. W tym celu, kiedy dane zostały już wysłane, powoduje on uruchomienie wątku *vAT86Decoding* w trybie transmisji, poprzez zapis znaku „t” do kolejki *xQueueStartDec*. Następnie przechodzi w stan oczekiwania na odebrany status, który powinien pojawić się w kolejce *xQueueWaitStatus*.

Uruchomiony w trybie transmisji wątek *vAT86Decoding* konfiguruje układ nadawczo-odbiorczy do odbioru oraz uruchamia przerwanie układu USART. Dlatego konieczne jest oczekiwanie na fizyczne wysłanie wszystkich danych. Wątek *vAT86Communication* zostaje ponownie uruchomiony, tym razem w trybie odbioru, po czym zostaje wywołana funkcja dekodująca.

W wątku *vA86Communication* następuje oczekiwanie na nadchodzącą zakodowaną informację z drugiego węzła o powodzeniu lub niepowodzeniu transmisji. Dane nadchodzą poprzez układ nadawczo-odbiorczy do portu USART i następnie, z procedury obsługi przerwania, trafiają kolejką *xQueueUSART* do wątku komunikacyjnego. Jego zadaniem jest wykryć i usunąć nagłówek oraz odczytać liczbę danych, które nadejdą. Liczba ta zależy od użytego sposobu kodowania i jest różna, mimo że status zawsze jest jednobajtowy.

Ostatecznie, wciąż zakodowany status trafia do kolejki *xQueueComDec*, której drugi koniec znajduje się w funkcji dekodującej. Odbiera ona rozmiar komunikatu, oblicza długość rzeczywistego komunikatu i przesyła ją do kolejki *xQueueDecApp*, a za nią kolejne zdekodowane bajty danych. Funkcja zwraca liczbę bajtów, których nie udało się zdekodować.

Wątek dekodujący w trybie transmisji powoduje wyczyszczenie kolejki *xQueueDecApp* – dane nie są przeznaczone dla wątku aplikacji, a gdyby użytkownik przełączył kierunek transmisji, to by te dane otrzymał. Następnie ma miejsce analiza statusu, a jej efekt zostaje zapisany do kolejki *xQueueWaitStatus*, w postaci znaku „r", gdy konieczna jest retransmisja lub dowolnego innego znaku, gdy pierwsza transmisja się powiodła. Wątek *vAT86Decoding* na tym kończy swoje zadanie i ulega zablokowaniu przy odczycie z pustej kolejki *xQueueStartDec*.

Kolejnym krokiem jest zakończenie lub ponowienie transmisji. W wątku *vAT86Encoding*, na podstawie otrzymanej z kolejki *xQueueWaitStatus* informacji, wykonywany jest jeden albo drugi scenariusz. W pierwszym przypadku, opuszczany jest semafor *xSemaphoreEncApp*, który dotąd blokował wykonanie wątku aplikacji i zakończenie funkcji *AT86_send_string*. W drugim przypadku operacja przesyłu danych i odbioru statusu, opisana powyżej, jest ponawiana tak długo, aż nie zostanie otrzymany pozytywny status, z tą różnicą, że źródłem danych zapisywanych do kolejki emphxQueueEncCom jest bufor *global_encoded_buffer*.

4.3.4. Odbiór danych

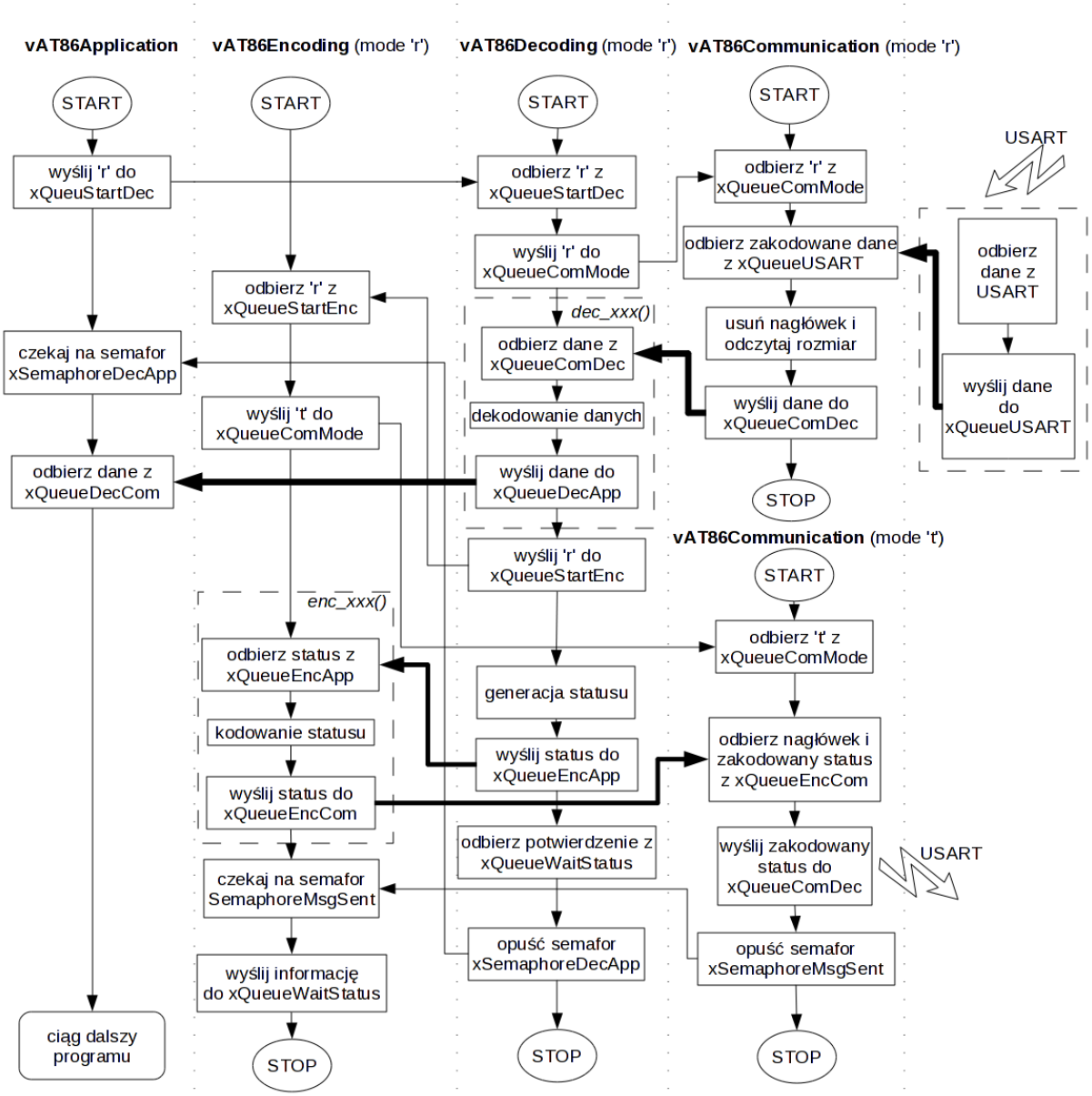
Proces odbioru danych przebiega podobnie do procesu ich wysyłania. Został on schematycznie zilustrowany na rysunku 4.5.

Rozpoczęcie odbioru danych powoduje użytkownik (np. poprzez przejście do odpowiedniej pozycji menu), zapisując do kolejki *xQueueMenu* znak „r". Wówczas zostaje uruchomiony wątek *vAT86Application*. Pierwszą podjętą w wątku akcją jest nadanie do kolejki *xQueueStartDec* litery „r", która spowoduje inicjalizację wątku *vAT86Decoding*. Wątek sygnalizuje gotowość do odbioru danych pozostałym wątkom a następnie zostaje zablokowany w oczekiwaniu na nadejście danych i możliwość ich obróbki.

Po uruchomieniu wątku *vAT86Decoding* następuje uruchomienie wątku *vAT86Communication* analogicznie jak w przypadku transmisji – poprzez zapis znaku do kolejki *xQueueComMode*. W przypadku odbioru, do kolejki zapisany zostaje znak „r" (*reception*), aby uruchomić komunikację w odpowiednim trybie. Wątek dekodujący odpowiada też za konfigurację układu nadawczo-odbiorczego oraz uruchomienie funkcji dekodującej, która natychmiast blokuje cały wątek, oczekując na pojawienie się danych w kolejce *xQueueComDec*.

Dane pojawiają się najpierw w funkcji obsługi przerwania *USART_IRQ_Handler()*, która natychmiast przesyła je kolejką *xQueueUSART* do wątku *vAT86Communication*. Tam następuje pozbawienie pakietu bajtów nagłówka, bajtów synchronizujących oraz zostaje zdekodowany rozmiar wiadomości. To właśnie informacja o długości komunikatu zostaje jako pierwsza przesłana do kolejki *xQueueComDec*, odblokowując wątek dekodowania.

W funkcji dekodującej odbierane są kolejne bajty zakodowanych danych. Każdy zdekodowany bajt jest natychmiast przesyłany do kolejki *xQueueDecApp*. Jej długość została dobrana tak, aby mogła pomieścić nawet maksymalną ilość 255 bajtów danych, nie ulegając



Rysunek 4.5. Schemat algorytmu odbioru danych w programie. Cieńsze strzałki symbolizują przebieg programu i instrukcji sterujących, grubsze strzałki oznaczają przepływ danych lub statusu pomiędzy wątkami, linią przerywaną wydzielono zadania wykonywane przez funkcje.

przepełnieniu. Do takiej sytuacji może dojść ponieważ wątek aplikacji nie odczytuje danych z kolejki, dopóki wszystkie nie zostaną zdekodowane. Niektóre algorytmy mogą bowiem mieć zdolność detekcji błędów przekraczającą zdolność ich korekcji. Wówczas pojawia się możliwość, że niektóre wykryte błędy nie będą mogły być naprawione – nie nastąpi więc walidacja danych poprzez opuszczenie semafora *xSemaphoreDecApp*, kolejka *xQueueDecApp* zostanie wyczyszczona, a status będzie zawierał informację o żądaniu ponownego przesłania danych.

W wątku dekodowania analizowana jest informacja zwrotna z funkcji dekodującej. Jeśli otrzymano zero, to znaczy, że wszystkie dane w kolejce *xQueueDecApp* powinny być

poprawne i można opuścić semafor *xSemaphoreDecApp*, umożliwiając odczyt danych w aplikacji. W przeciwnym wypadku walidacja danych nie będzie miała miejsca. Niezależnie od powodzenia dekodowania należy jednak przesłać status. W tym celu wątek dekodowania uruchamia wątek *vAT86Encoding* i zapisuje do kolejki *xQueueAppEnc* (z której przy transmisji danych korzysta aplikacja) rozmiar wiadomości równy 1 oraz bajt statusu. Następuje zablokowanie wątku do czasu wysłania statusu.

Wątek kodujący dokonuje kodowania oraz uruchamia wątek komunikacji tak, jak ma to miejsce przy transmisji danych. Różnica w stosunku do trybu nadawania polega na tym, że po powrocie z funkcji kodującej nie są podejmowane żadne działania poza zapisem dowolnej danej do kolejki *xQueueWaitStatus*, co spowoduje odblokowanie wątku dekodowania.

Jeśli dane zostały odebrane i zdekodowane poprawnie, to wątek dekodujący kończy swoją pracę docierając do blokującego go odczytu z pustej kolejki *xQueueStartDec*, wcześniej opuszczając semafor *xSemaphoreDecApp* i umożliwiając dalszą pracę wątkowi aplikacji. Jeśli jednak w komunikacji nastąpiły niekorygowalne ale wykrywalne błędy i wysłano kanałem zwrotnym żądanie retransmisji, to należy dane ponownie odebrać. W tym celu wątek aplikacji zapisuje do kolejki *xQueueStartDec* znak 'r', powodując w ten sposób swoje ponowne uruchomienie w trybie odbioru. Czynność ta będzie powtarzana aż do skutecznego przesłania danych lub przerwania działania programu.

Wątek aplikacji, po skutecznym podniesieniu semafora *xSemaphoreDecApp*, odczytuje dane z kolejki *xQueueDecApp* i przechodzi do dalszych czynności zdefiniowanych przez użytkownika.

4.4. Możliwości dalszego rozwoju projektu

Powstające w ramach pracy dyplomowej oprogramowanie, aby mogło być dalej rozwijane, wymaga zgromadzenia szczegółowych informacji o tym, jak powstałe narzędzie rozbudować i dostosować do potrzeb konkretne aplikacji.

4.4.1. Wymagania dla funkcji kodującej i dekodującej

W ramach pracy dyplomowej zrealizowano cztery algorytmy kodowania korekcyjnego. Położono duży nacisk na to, aby kolejne algorytmy można było łatwo dodać do programu. Aby skutecznie zaimplementować nowy kod korekcyjny lub detekcyjny, muszą zostać stworzone dwie funkcje: kodująca i dekodująca. W pracy przyjęto konwencję nazywania funkcji polegającą na poprzedzeniu mnemoniku nazwy algorytmu literami *enc_* lub *dec_*.

Funkcja kodująca musi:

- być wywoływana w odpowiednim miejscu wątku *vAT86Encoding* – zostanie ona uruchomiona po zapisie do kolejki *xQueueAppEnc* rozmiaru danych,

- odbierać dane do zakodowania nadchodzące bajtami (char) z kolejki *xQueueAppEnc*; jako pierwszy z kolejki powinien być odczytany jeden bajt, informujący o liczbie bajtów do zakodowania,
- przesyłać zakodowane dane do wysłania przy pomocy kolejki *xQueueEncCom*; jako pierwszy do kolejki powinien być wstawiony niezakodowany bajt reprezentujący objętość zakodowanej wiadomości w bajtach,
- jednocześnie wszystkie zapisywane do kolejki dane, włącznie z informacją o długości wiadomości, powinny być jednocześnie zapisywane do tablicy *global_encoding_buffer*, poczynając od jej pierwszego elementu, co umożliwi w przypadku nieudanej transmisji ponowne przesłanie danych, bez potrzeby ich kodowania,
- powiadamiać wątek komunikacji o gotowości danych do wysłania poprzez opuszczenie semafora *xSemaphoreEncCom*,
- zwracać liczbę zakodowanych bajtów, a w przypadku błędu zwracać 0.

Funkcja dekodująca musi:

- być wywoływana w wątku *vAT86Decoding* – zostanie ona uruchomiona po odblokowaniu wątku *vAT86Communication*,
- odbierać dane do odkodowania nadchodzące bajtami z kolejki *xQueueComDec*; jako pierwszy z kolejki powinien być odczytany bajt informujący o liczbie bajtów do odkodowania (długość zakodowanej wiadomości),
- przesyłać odkodowane dane przy pomocy kolejki *xQueueDecApp* do wątku aplikacji *vAT86Application*; jako pierwszy musi zostać przesłany bajt, informujący o liczbie odkodowanych bajtów (długość zdekodowanej wiadomości),
- powiadamiać o gotowości odkodowanych danych do odczytu poprzez opuszczenie semafora *xSemaphoreDecApp*,
- zwracać liczbę bajtów, których nie udało się zdekodować.

4.4.2. Konstrukcja nowych aplikacji

Zaimplementowany wątek aplikacji służy jedynie do badania algorytmów kodowania korekcyjnego. Może on jednak posłużyć jako baza do powstania kolejnych, bardziej zaawansowanych projektów. Kod programu powinien być umieszczony w wątku aplikacji. Aby przesłać lub odebrać wiadomość muszą zostać wywołane z wątku aplikacji odpowiednie funkcje. Stanowią one interfejs użytkownika, umożliwiający wykorzystanie stworzonego protokołu komunikacyjnego.

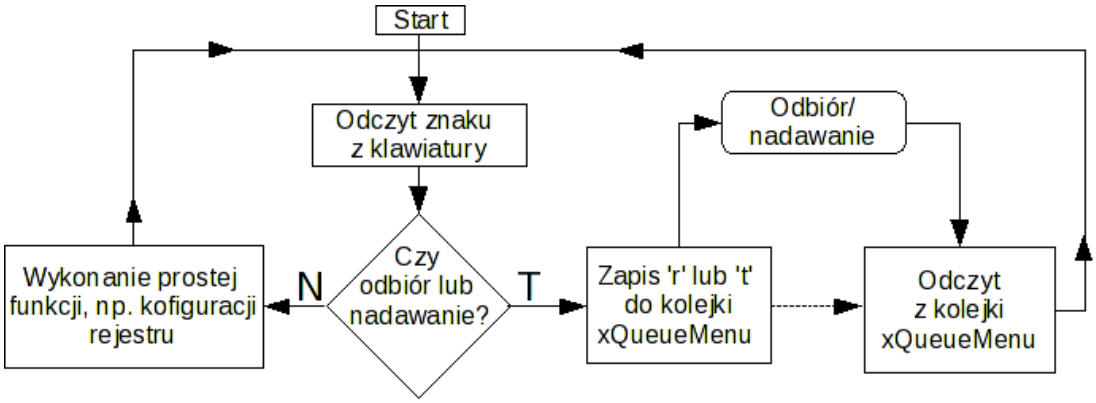
Wysyłanie wiadomości odbywa się przez przekazanie wskaźnika do danych, wraz z informacją o ich długości, do funkcji *AT86_send_string()*. Wartością zwracaną przez funkcję, po udanym przesłaniu danych, jest 0. Wywołanie funkcji jest operacją blokującą wykonanie

wątku do czasu ukończenia transmisji. Ma to na celu zapobieżenie sytuacji, w której układ nadawczo-odbiorczy zostanie zrekonfigurowany na odbiór zanim wysłane zostaną wszystkie dane. Jeśli powstająca, nowa aplikacja nie będzie stwarzała takiego zagrożenia, to blokujący semafor *xSemaphoreMessageSent* można usunąć z ciała funkcji *AT86_send_string*, przyspieszając tym samym wykonanie programu.

Odbiór wiadomości rozpoczyna się od uruchomienia wątku *vAT86Decoding* i zapisania do kolejki *xQueueStartDec* znaku 'r'. Dalsze operacje przebiegają automatycznie. O gotowości danych do odbioru użytkownik zostanie powiadomiony poprzez opuszczenie semafora *xSemaphoreDecApp* — dane będzie można wówczas odczytać z kolejki *xQueueDecApp*, przy czym pierwszym odczytanym bajtem będzie informacja o długości komunikatu.

4.4.3. Sterowanie

Zastosowane w pracy menu charakteryzuje się prostą budową – schemat działania menu przedstawia rysunek 4.6. Dopóki użytkownik wykonuje czynności nie wymagające przej-



Rysunek 4.6. Schemat działania menu.

ścia w tryb nadawania lub odbierania, program pozostaje w wątku *vMenu*. W chwili, kiedy konieczne jest rozpoczęcie transmisji lub odbioru, wątek *vMenu* przesyła kolejką *xQueueMenu* informację do wątku *vAT86Application*, po czym zostaje zablokowany na próbie odczytu z tej kolejki.

Kiedy użytkownik wciśnie dowolny klawisz, wątek aplikacji prześle informację o zaprzestaniu komunikacji tą samą kolejką, odblokowując w ten sposób menu.

4.4.4. Zmiana parametrów komunikacji

W kanale transmisyjnym mogą być zmieniane następujące parametry komunikacji:

- prędkość transmisji protokołu szeregowego,
- wybór częstotliwości pracy układu nadawczo-odbiorczego,
- wybór kodu korekcyjnego,

- wykorzystanie kanału zwrotnego.

Wyboru kodu korekcyjnego dokonuje się w menu aplikacji, poprzez zapisanie do zmiennej globalnej *global_code* odpowiedniego znaku, oznaczającego dany kod korekcyjny:

- m – zwielokrotnianie danych,
- h – kod Hamminga (7,4),
- i – kod Hamminga (63,57),
- b – kod BCH (32,16).

Analogicznie można ustalić, który kod będzie używany do kodowania statusu w kanale zwrotnym, zapisując znak do zmiennej *global_code_status*.

Prędkość transmisji szeregowej jest głównym czynnikiem sterującym przepustowością łącza. Można ją zmieniać w granicach od 1200 b/s do 64000 b/s poprzez zmianę argumentu wywołania funkcji *UART_Init*, używanej jednokrotnie na początku funkcji *main* (jeszcze przed uruchomieniem wątku *schedulera*). Wraz ze wzrostem szybkości powyżej 38400 b/s pojawiają się w kanale transmisyjnym drobne zakłócenia, nawet na bliskich odległościach, spowodowane trudniejszą synchronizacją. Ustawiając maksymalną dopuszczalną przez układ nadawczo-odbiorczy szybkość transmisji (64000 b/s), podczas jednej milisekundy można wysłać lub odebrać 8 bajtów. Zakodowanie 8 bajtów danych trwa poniżej 0,3 ms, co zostało zmierzone[13]. Prędkość transmisji można też regulować w trakcie działania programu, przy pomocy wyżej wymienionej funkcji, ale należy zwrócić uwagę, że oba węzły: odbiorczy i nadawczy muszą pracować z tą samą szybkością.

Wybór częstotliwości pracy układu nadawczo-odbiorczego AT86RF211 odbywa się poprzez zapis do odpowiednich rejestrów konfiguracyjnych, jednak ich wartości nie są wprost powiązane z uzyskiwaną częstotliwością, a producent nie dostarcza żadnych narzędzi pozwalających je uzyskać. Dane wykorzystane w niniejszej pracy pochodzą z [13]. Szerokość pasma można regulować przy pomocy tych samych rejestrów. Kilka przykładowych zawartości odpowiednich rejestrów, konfigurujących różne szerokości pasma przy częstotliwości 433 MHz, zawarto w komentarzach do kodu źródłowego.

[13] pomiaru dokonano w programie działającym na systemie FreeRTOS

5. Implementacja algorytmów i kodów korekcyjnych

5.1. Kody korekcyjne

Przy doborze kodowania korekcyjnego powinno się zwracać uwagę na:

- poziom zakłóceń w kanale transmisyjnym,
- oczekiwaną szybkości transmisji,
- rozmiar i szybkość przyrostu danych,
- protokół transmisji,
- specyfikę urządzeń wykorzystanych do stworzenia cyfrowego łącza danych.

Powyższe kryteria są ze sobą powiązane, np. w silnie zakłóconym kanale transmisyjnym duża zdolność korekcyjna kodu może być okupiona spowolnieniem transmisji. Nie są znane ostateczne warunki pracy urządzenia, więc również wybór odpowiedniego kodu korekcyjnego nie jest możliwy. Dlatego zaimplementowanych zostało kilka przykładowych kodów korekcyjnych. Można wybrać ostatecznie jeden z nich lub zastosować inny algorytm.

Specyfika pracy z mikroprocesorem sprawia, że należy przy wyborze kodów korekcyjnych brać pod uwagę długości typów danych, reprezentowanych przez daną architekturę oraz interfejs komunikacji na poziomie sprzętu. W tworzonym oprogramowaniu najmniejszą jednostką informacji jest bajt przechowywany w zmiennej *char*[1]. Ponadto interfejs USART, którym wysyłane są dane, operuje ośmiobitowymi rejestrami. Warto więc wybierać takie kody korekcyjne, w których długość wektora wyjściowego jest wielokrotnością liczby osiem. Mają one największą szansę okazać się użyteczne, gdyż można nimi przesłać dowolną ilość danych i nigdy nie nastąpi przesłanie nieużytecznych bitów, wynikających z architektury urządzenia[2].

Długość wektora wejściowego również powinna być dopasowana do ilości przesyłanych danych. Czujniki, które obsługują węzły sieci najczęściej operują danymi o określonej, stałej długości. Znając tę długość można dobrać kod korekcyjny tak, aby kodował dokładnie tyle bitów, ile potrzeba. Mimo, że zmienna w pamięci mikrokontrolera przechowywana będzie w zmiennych o rozmiarze będącym wielokrotnością ośmiu, to przesyłane drogą radiową powinny być tylko znaczące bity tych zmiennych.

Prosty protokół komunikacji, zaimplementowany w systemie FreeRTOS i opisany w rozdziale 4 operuje wyłącznie bajtami danych. Zmienna *char* jest najmniejszą jednostką dla jakiej możliwe jest zadeklarowanie kolejki systemu.

[1] zmienna *bit* języka C bazuje na zmiennej *char*, więc reprezentacja danych przy jej pomocy, choć upraszcza kod źródłowy, jest mało efektywna

[2] przykład: ma zostać wysłany 5-bitowy wektor kodowy, ale nie można do rejestru wyjściowego interfejsu USART zapisać mniej niż osiem bitów – trzy bity zostaną wysłane niepotrzebnie

Powyższe przesłanki zostały uwzględnione przy wyborze do implementacji przykładowych kodów korekcyjnych. Posiadają one długości wektorów wejściowych i wyjściowych równe bądź zbliżone do wielokrotności liczby osiem. W przypadku kodowania liczby bitów nie odpowiadającej dokładnie długości wektora wejściowego, zastosowano wypełnienie wektora kodowego sekwencją bitów o wartości 0, rozpoczynającą się od bitu o wartości 1. Innymi słowy: w każdym pakiecie pierwszy bit ustawiony na „1" oznacza początek nieużytecznych danych.

5.1.1. Powtórzeniowy

Pierwszym zaimplementowanym kodem korekcyjnym był najprostszy możliwy kod – powtórzeniowy, którego działanie polega na przesłaniu każdego bajtu cztery razy. Jest to więc kod liniowy (32,8). Jego minimalna odległość Hamminga wynosi $d = 8$, ale zdolność korekcyjna silnie zależy od rozkładu błędów, ponieważ przy kodowaniu i dekodowaniu brane są pod uwagę całe bajty a nie bity danych. Ta modyfikacja w stosunku do najbardziej intuicyjnej implementacji takiego kodu, polegającej na powielaniu bitów, wynika z dodatkowej oszczędności czasu procesora. Przy kodowaniu informacji nie są wówczas wymagane żadne operacje na bitach, a jedynie na całych bajtach.

Dekodowanie danych polega na porównywaniu ze sobą czterech odebranych bajtów danych, w wyniku czego można wyróżnić cztery możliwości:

wszystkie cztery bajty są takie same – co interpretowane jest jako brak przekłamań w transmisji, bowiem prawdopodobieństwo wystąpienia dokładnie takiego samego przekłamania, w każdym z czterech odebranych bajtów jest bardzo małe – $1 : 2^{24}$.

jeden z bajtów różni się od pozostałych – wystąpił błąd (lub błędy), ale najprawdopodobniej trzy identyczne bajty wskazują prawidłową treść oryginalnego komunikatu,

dwa bajty są takie same, a dwa pozostałe różne od nich i od siebie – taka sytuacja ma najczęściej miejsce, kiedy zakłócenia wystąpiły w dwóch bajtach, a dwa pozostały niezakłócone i to one wskazują poprawną treść komunikatu,

pozostałe przypadki: dwa bajty mają identyczną wartość, ale pozostałe dwa również się między sobą nie różnią, bądź wszystkie cztery bajty mają różne wartości; w tych przypadkach nie sposób ocenić która część wiadomości uległa zakłóceniom – komunikat należy więc odrzucić.

5.1.2. Liniowy Hamminga (7,4)

Kod Hamminga (7,4) został zaimplementowany ze względu na nieskomplikowany sposób kodowania i dekodowania, pozwalający na wydajną implementację.

Jako dane wejściowe przyjmuje 4-bitowe słowo. Do zakodowania jednego bajtu danych będą więc potrzebne dwie operacje kodowania: jedna dla mniej ważnych i jedna dla ważniejszych czterech bitów. Pod tym względem rozkład informacji odpowiada rozmiarowi używanych

zmiennych. W wyniku działania takiego kodu powstawałyby jednak dwa komunikaty siedmiobitowe, co powodowałoby utratę jednego bitu na osiem.

Aby tego uniknąć wykorzystano nadmiarowy bit jako sumę kontrolną pozostałych siedmiu[3]. Wydłuży to czas kodowania i dekodowania, ale zwiększy zdolność detekcyjną kodu.

Obliczanie wektora kodowego polega na sumowaniu ze sobą odpowiednich bitów wektora informacyjnego, przy pomocy operacji XOR i przesunięć bitowych. Wykorzystywany jest więc opisany w punkcie 2.4.2.1 układ równań 2.2.

5.1.3. Liniowy Hamminga (63,57)

Do korekcji sporadycznych jednobitowych przekłamań, które mogą się zdarzać w kanale komunikacyjnym dobrze nadaje się dłuższy, 63-bitowy kod Hamminga. Można go przesyłać przy pomocy ośmiobitowego łącza RS232, tracąc jeden bit z 64. Ponadto liczba bitów informacyjnych (57) pozwala na zakodowanie do 7 całych bajtów danych.

W kodzie Hamminga (63,57) może zaistnieć potrzeba kodowania mniej niż 57 bitów danych. W takim przypadku puste bity, niezbędne do obliczenia wartości elementów zostają wypełnione sekwencją bitów o wartości 0, rozpoczynającą się bitem o wartości 1.

5.1.4. Cykliczny BCH (31,16)

Kody cykliczne pozwalają na korekcję większej liczby błędów niż kody Hamminga, a więc znajdą zastosowanie w silniej zaszumionych kanałach komunikacyjnych.

Do implementacji w postaci funkcji kodującej i dekodującej, wybrano kod BCH (31,16), który charakteryzuje się prostszą konstrukcją koderów i dekoderów niż inne kody tego typu i dobrymi właściwościami korekcyjnymi [11]. Kod posiada zdolność korekcji do trzech przekłamań w każdym 31-bitowym pakiecie. O wyborze długości kodu zadecydowała ponownie specyfika przetwarzania danych w protokole aplikacji i interfejsie USART. Liczba kodowanych bitów pasuje do architektury procesora i zmiennych wykorzystywanych do komunikacji międzywątkowej w aplikacji, wynosi ona 16 bitów, czyli dokładnie 2 bajty.

Tak jak w przypadku kodu Hamminga (63,57) może zaistnieć problem przy wysyłaniu ilości danych mniejszej niż pełne dwa bajty. Zastosowane zostało analogiczne rozwiązanie, polegające na wypełnieniu wektora informacyjnego odpowiednią sekwencją bitów.

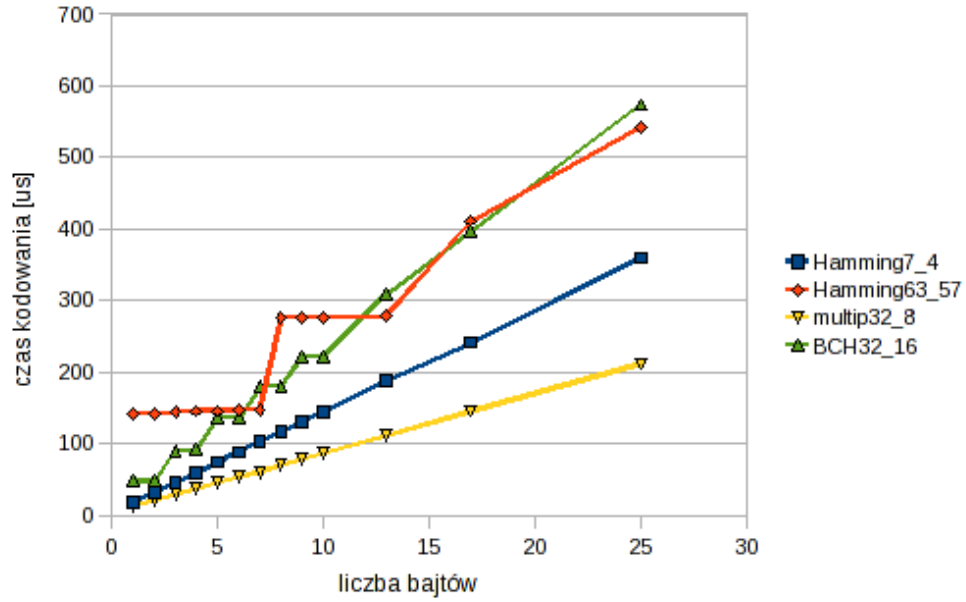
Kodowanie i dekodowanie wykonywane jest dokładnie tak, jak to opisano w punkcie 2.4.2.2. Wielomiany reprezentują 32-bitowe zmienne typu *long*. Stworzona została funkcja *PolyDivRemainder*, która przyjmuje jako argumenty dwa wielomiany i zwraca resztę z ich dzielenia. Na takim działaniu opierają się bowiem zarówno algorytm kodujący, jak i dekodujący.

[3] rozwiązanie takie jest stosowane w kodach SECDED (od ang. *Single Error Correction Double Error Detection*)

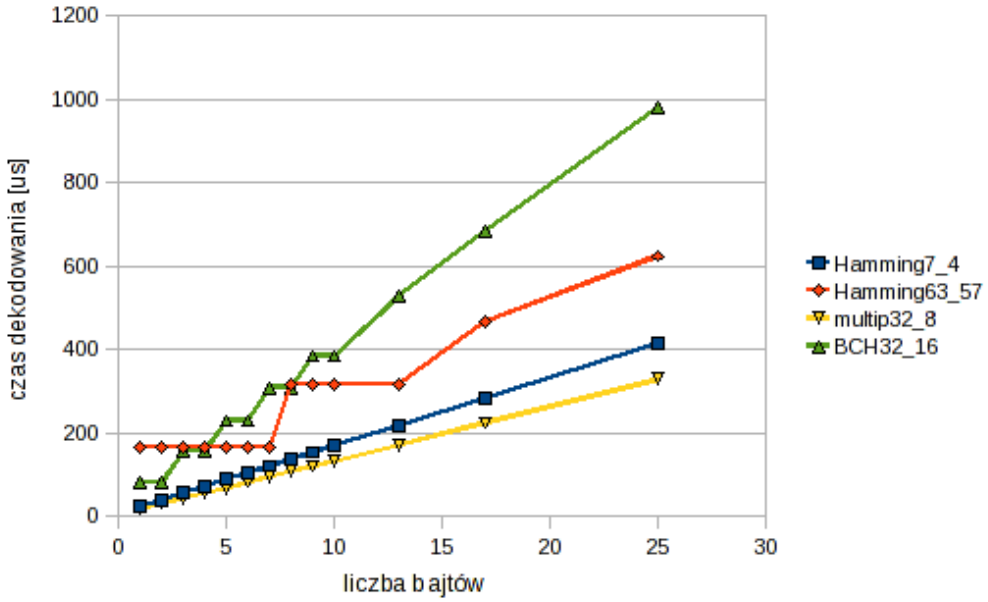
5.2. Analiza zaimplementowanych kodów

5.2.1. Szybkość działania

Ważny parametr jakim jest czas kodowania i dekodowania najłatwiej jest sprawdzić empirycznie, gdyż zależy on od mocy obliczeniowej procesora oraz optymalizacji kodu źródłowego. W celu przeprowadzenia pomiaru stworzono prostą aplikację, działającą na architek-



Rysunek 5.1. Wykres czasów kodowania wybranych kodów korekcyjnych. Początkowa część wykresu (dla 1-13 bajtów) została wykonana z większą dokładnością, aby przedstawić nieliniowość czasu kodowania.



Rysunek 5.2. Wykres czasów dekodowania wybranych kodów korekcyjnych. Początkowa część wykresu (dla 1-13 bajtów) została wykonana z większą dokładnością, aby przedstawić nieliniowość czasu kodowania.

turze ARM7 i pozwalającą zmierzyć czasy kodowania i dekodowania poszczególnych kodów, w zależności od długości komunikatu. Aplikacja ustawiała jeden z portów PIO w stan wysoki, w chwili wejścia do funkcji kodującej, a w stan niski przy wyjściu z tej procedury. Szerokość powstałego impulsu można było zmierzyć przy pomocy oscyloskopu. Mierzone parametry różnią się w tak dużym stopniu, że niedokładność pomiarowa nie ma dużego znaczenia.

Obserwacje (przedstawione na rysunkach 5.1 i 5.2) wykazały, że najdłużej zajmuje procesorowi zakodowanie i zdekodowanie wiadomości kodem cyklicznym BCH (31,16), a najmniej kodem powtórzeniowym. Czas kodowania i dekodowania wszystkich kodów zależy liniowo od liczby bajtów.

Niektóre kody: kod BCH i kod Hamminga (63,57) wymagają odpowiedniej liczby bajtów do zakodowania komunikatu[4]. Z tego powodu wykresy czasów kodowania i dekodowania tych kodów zmieniają się gwałtownie co kilka bajtów. Na przykład kod Hamminga (63,57) potrzebuje tyle samo czasu na zakodowanie jednego i siedmiu bajtów. Natomiast różnica w czasie kodowania 7 bajtów i 8 bajtów tym kodem jest dwukrotna.

5.2.2. Zdolności korekcyjne

Przez zdolność korekcyjną określa się w literaturze [10, 11] maksymalną liczbę bitów w pojedynczym wektorze kodowym, które mogą zostać skorygowane w przypadku ich przekłamania. Nie wskazuje to jednak wprost, w jak silnie zaszumionym kanale komunikacyjnym sprawdzi się dany kod. Powinna więc istnieć możliwość sprawdzenia rzeczywistych efektów zastosowania danego kodu korekcyjnego, dla komunikatów o różnych długościach i przy różnych zakłóceniach.

W ramach pracy stworzono aplikację, która pozwala na przeprowadzenie statystycznych testów kodów korekcyjnych przy pseudolosowym[5] zakłóceniu kanału komunikacyjnego i dla określonej ilości danych do przesłania. Może się ona okazać przydatna do symulacji pracy urządzenia w określonych warunkach. Szczegółowy opis aplikacji znajduje sie w dodatku D.

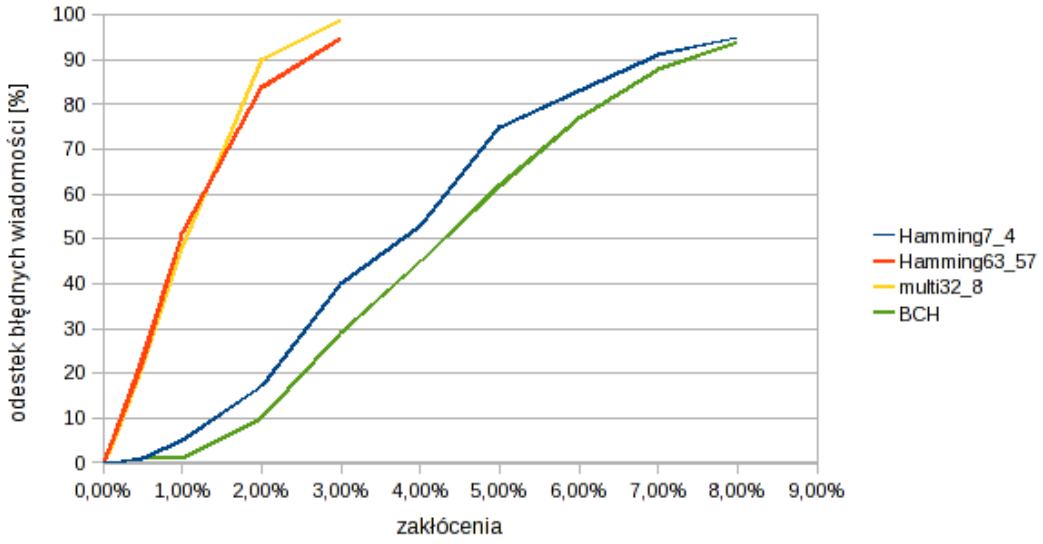
Na rysunku 5.3 przedstawione zostały efekty testów wszystkich zaimplementowanych kodów korekcyjnych, dla komunikatu o stałej długości 56 bajtów, przesyłanego kanałem zakłóconym w zmiennym stopniu.

Z wykresu nie należy odczytywać konkretnych wartości skuteczności kodów, gdyż jest ona specyficzna dla danej długości komunikatu. Ważniejsze jest jednak zestawienie skuteczności różnych kodów w identycznych warunkach.

Kod korekcyjny Hamminga (7,4) wykazuje podobne właściwości korekcyjne do kodu cyklicznego BCH (31,16), mimo że ten pierwszy ma zdolność korekcyjną 1 bitu, a drugi 3 bitów. Przyczyną jest metodologia badania, sprawdzająca rzeczywistą wydajność kodowania w transmisji komunikatu o określonej długości. Jeśli przesłana ma być określona liczba bajtów,

[4] kod Hamminga (63,57) koduje 57 bitów (czyli ponad 7 bajtów) danych; kod BCH koduje 16 bitów, (czyli 2 bajty)

[5] wykorzystano funkcję *rand()* języka C



Rysunek 5.3. Wykresy skuteczności kodów w zależności od stopnia zakłóceń w kanale transmisyjnym.

to porównując kod Hamminga z kodem BCH należy traktować go jako kod, który 16 bajtów danych dzieli na 4 mniejsze wektory informacyjne i każdy koduje z osobna. W związku z tym zdolność korekcyjna dla 4 bajtów wynosi 1 bit, a dla 16 bajtów – 4 razy więcej. Przewaga kodu BCH polega na możliwości korekcji zgrupowanych obok siebie uszkodzonych bitów, której kod Hamminga nie posiada.

Zgodnie z przewidywaniami, najsłabsze zdolności do korekcji błędów wykazał kod Hamminga (63,57). Znajdzie on zastosowanie albo w mało zaszumionych kanałach, albo przy szybkich transmisjach i małym zaszumieniu, do korekcji pojedynczych przekłamań w pakietach.

Wykres nie przedstawia zdolności detekcyjnych kodów, które przy użyciu kanału zwrotnego mają znaczenie praktyczne. Nawet dla zakłóceń rzędu 40% kody BCH i kod powtórzeniowy wykazywały w trakcie badania stuprocentową detekcyjność błędów i nie zwracały do programu żadnego przekłamanego wektora, za każdym razem żądając retransmisji. W sieci czujnikowej komunikującej się kanałem transmisyjnym podzielonym na część główną i zwrotną można nawet zastosować algorytmy posiadające tylko zdolności detekcyjne, a nie posiadające żadnych zdolności do korygowania wykrytych błędów, np. sumy CRC (od ang. *Cyclic Redundancy Check*). Możliwe jest też przeciwne rozwiązanie: kodowanie Reeda-Solomona pozwala na minimalizację ilości wysyłanych danych dzięki obliczaniu, na podstawie informacji o wykrytych błędach, wyłącznie danych uzupełniających wykryte uszkodzenia wektora kodowego. Te zagadnienia wykraczają jednak poza ramy niniejszej pracy.

Oprócz losowych przekłamań w transmisjach, mogą się też zdarzać tzw. „błędy blokowe”. Polegają one na występowaniu dużej ilości błędów, skupionych w jednym miejscu. Jedynym, zaimplementowanym w konstruowanym urządzeniu, zabezpieczeniem przed tego typu błędami jest kanał zwrotny, którym w razie niepowodzenia transmisji, można zażądać retransmisji danych. Możliwa jest też implementacja specjalnych algorytmów kodowania, pozwalających na korekcję takich przekłamań, bądź też zastosowanie przeplotu danych.

6. Podsumowanie i wnioski końcowe

Skonstruowany i oprogramowany w ramach pracy prototyp urządzenia nadaje się do zastosowania w sieciach czujnikowych o odległościach między węzłami do kilkuset metrów. Transmisja została dodatkowo zabezpieczona poprzez zastosowanie kodowania korekcyjnego oraz kanału zwrotnego. Urządzenie może działać w środowiskach zakłóconych w różnym stopniu, ponieważ dysponuje możliwością dopasowania kodu korekcyjnego do jakości kanału komunikacyjnego.

Konfiguracja *toolchaina* opartego o rozwiązania *open-source* wymaga wiedzy i doświadczenia w pracy z systemami Linux. Uzyskane dzięki temu narzędzie jest w pełni sprawne i wygodne w użytkowaniu.

6.1. Kod źródłowy programu

Kod źródłowy był tworzony tak, aby można go było łatwo dalej rozwijać. Wskazuje na to implementacja algorytmów korygujących jako osobnych funkcji oraz przeniesienie większości procesów do wątków systemu *FreeRTOS* odpowiedzialnych za komunikację i kodowanie, co pozwoliło pozostawić wątek aplikacji możliwie nieskomplikowanym.

Jeśli zajdzie taka potrzeba, sam algorytm działania może być zaimplementowany na innej architekturze i innym urządzeniu nadawczo-odbiorczym. Twórcy systemu *FreeRTOS* zapewnili już wsparcie dla kilkunastu typów procesorów [4].

6.2. Analiza kodów korekcyjnych i kanału transmisyjnego

Kody korekcyjne bardzo różnią się między sobą właściwościami. Dzięki temu możliwe jest dobre dopasowanie algorytmu kodowania do konkretnych potrzeb i optymalizacja pasma. Pod uwagę należy brać trzy podstawowe parametry kodów: zdolności korekcyjne, sprawność i złożoność obliczeniową.

W zaprojektowanym układzie celem korekcji było polepszenie przepustowości pasma, poprzez zapobieżenie częstej retransmisji danych w przypadku wystąpienia przekłamań. Im większe będą zdolności korekcyjne zastosowanego algorytmu kodowania, w tym mniejszym stopniu będzie wykorzystywany kanał zwrotny.

Stworzone oprogramowanie daje możliwość symulacji zakłóconego kanału komunikacyjnego, którym przesyłane są różne ilości danych. Dzięki temu można łatwo sprawdzić słuszność wyboru danego kodu korekcyjnego do pracy w określonych warunkach.

6.3. Możliwe usprawnienia

W ramach pracy zrealizowano najniższe warstwy komunikacji bezprzewodowej tak, aby urządzenie uzyskało funkcjonalność węzła sieci sensorowej i możliwa była implementacja (w wątku aplikacji) obsługi czujnika i konkretnej struktury sieci.

Implementacja algorytmów korygujących może mieć na celu albo zapewnienie ciągłości komunikacji, albo optymalizację pasma transmisyjnego (wówczas dopuszczalna jest nieciągłość i retransmisje). W zależności od ostatecznego zastosowania, rozwój protokołu komunikacyjnego może potoczyć się w jednym z tych dwóch kierunków.

Niewykorzystana pozostała funkcja *Wake-up* układu nadawczo-odbiorczego, pozwalająca na przechodzenie w stan obniżonego poboru mocy. Warto skorzystać z tej możliwości, w przypadku wykorzystania urządzenia jako czujnika zasilanego bateryjnie i nie pracującego w sposób ciągły.

Status przesyłany kanałem zwrotnym ma osiem bitów, z czego wykorzystywany jest tylko jeden. Taka długość statusu wynika ze specyfiki urządzenia USART, które operuje ośmiobitowymi komunikatami. Istnieje kilka możliwości wykorzystania pozostałych siedmiu bitów statusu:

- przesył informacji odczytanej z rejestru urządzenia nadawczo-odbiorczego CTRL1, mówiącej o sile sygnału odbieranego; na tej podstawie można adaptować moc sygnału transmitowanego optymalizując zużycie energii,
- przesył informacji o jakości kanału transmisyjnego (ilości błędów) celem dostosowania kodu korekcyjnego do transmisji,
- uzgadnianie szybkości działania portu szeregowego,
- czasowe wstrzymywanie komunikacji.

6.4. Dalsze prace nad projektem

Można wyróżnić trzy główne kierunki dalszego rozwoju zaprojektowanego i oprogramowanego urządzenia, zależne od jego zastosowania.

Pierwszy z nich dotyczy dopracowania protokołu komunikacyjnego pod kątem zastosowania w konkretnej sieci sensorowej.

Drugi możliwy kierunek rozwoju dotyczy implementacji algorytmów korekcyjnych, dobranych do charakteru przesyłanych danych oraz do własności kanału transmisyjnego.

Trzecim możliwym kierunkiem rozwoju oprogramownia jest dodanie obsługi czujników, zarówno w warstwie sprzętowej, poprzez elektryczne połączenie ich z układem, jak i programowej – wykorzystując odpowiednie peryferia mikrokontrolera.

Bibliografia

[1] Atmel Corp., *AT91SAM7S Series Preliminary*, rev. J, updated 09/2009, San Jose, 2008. http://www.atmel.com/dyn/resources/prod_documents/doc6175.pdf (dost. 2.08.2010)

[2] Atmel Corp., *FSK Transceiver for ISM Radio Applications AT86RF211 (aka: TRX01)*, Rev. 1942C-WIRE-06/02, San Jose, 2002. www.datasheetcatalog.org/datasheet/atmel/DOC1942.PDF (dost. 2.08.2010)

[3] Augustyn J., *Projektowanie systemów wbudowanych na przykładzie rodziny SAM7S z rdzeniem ARM7TDMI*, Wydawnictwo IGSMiE PAN, Kraków, 2007.

[4] Barry R., *Using the FreeRTOS Real Time Kernel, A Practical Guide*, Real Time Engineers Ltd., v. 1.3.0, ePrint, 2010.

[5] Brownell D., *OpenOCD User's Guide*, release 0.4.0-dev, ePrint, 2009. http://openocd.berlios.de/doc/pdf/openocd.pdf (dost. 2.08.2010)

[6] Brzoza-Woch R., *Mikrokontrolery AT91SAM7 w przykładach*, wyd. I, Wydawnictwo BTC, Legionowo, 2009.

[7] IEEE Computer Society, *Part 3: Carrier sense multiple access with Collision Detection (CSMA/CD) Access Method and Physical Layer Specifications*, IEEE Standard for Information Technology, Telecommunications and information exchange between systems, Local and metropolitan area networks, Ethernet Working Group (802.3), nr Std 802.3-2008, Nowy Jork, 2008. http://standards.ieee.org/getieee802/download/802.3-2008_section1.pdf (dost. 2.08.2010)

[8] Klukowski J., Nabiałek I., *Algebra dla studentów*, wyd. IV, Wydawnictwa Naukowo-Techniczne, Warszawa, 2004.

[9] Mahalik N.P. (red.), *Sensor Networks and Configuration*, first edition, Springer, Berlin Heidelberg, Nowy Jork, 2006.

[10] MacWilliams F.J., Sloane N.J.A., *The Theory of Error-Correcting Codes*, first edition, North Holland Publishing Company, Nowy Jork, 1997.

[11] Mochnacki W., *Kody korekcyjne i kryptografia*, wyd. I, Oficyna Wydawnicza Politechniki Wrocławskiej, Wrocław, 1997.

[12] Naubauer A., Freudenberger J., Kühn V., *Coding theory: algorithms, architectures, and applications*, John Wiley & Sons Ltd., Chichester, 2007.

[13] Neumann M. A., Paetsch C. R., Schilz S. R., Schuet D. J., *Big Blimpin' – Development of an Advanced Control Airship*, praca inżynierska, Santa Clara University, Department of Electrical Engineering, Department of Mechanical Engineering, 2003.

[14] Nowicki K., Woźniak J., *Sieci LAN, MAN i WAN – protokoły komunikacyjne*, wyd. I, Wydawnictwo Fundacji Postępu Telekomunikacji, Kraków, 1998 (podrozdz. 1.3, s. 9-17).

[15] Propox, *EVBsam7s – Instrukcja użytkownika*, rev 2, ePrint, 2006. http://www.propox.com/download/docs/EVBsam7s_rev2_pl.pdf (dost. 2.08.2010)

[16] Propox, *MMsam7s – Instrukcja użytkownika*, rev 2, ePrint, 2006. http://www.propox.com/download/docs/MMsam7s_rev2_pl.pdf (dost.2.08.2010)

[17] Simpson W. red., *Point-to-Point Protocol*, Internet Engineering Task Force (IETF), Point-to-Point Protocol Working Group, 1994. http://tools.ietf.org/html/rfc1661 (dost. 25.07.2010)

[18] Wesołowski K., *Podstawy cyfrowych systemów telekomunikacyjnych*, wyd. I, wydawnictwa Komunikacji i Łączności, Warszawa, 2003.

Dodatek A. Wybrane zagadnienia z zakresu algebry

Wszystkie definicje z dodatku A zostały opracowane w oparciu o [8].

Grupa

Jeśli w zbiorze niepustym G jest określone działanie algebraiczne $\circ$, to mówimy, że dana jest *struktura algebraiczna* $(G, \circ)$.

Strukturę algebraiczną $(G, \circ)$ nazywa się *grupą*, gdy:

i) działanie $\circ$ jest łączne

$$\forall_{x,y,z \in G}\ x \circ (y \circ z) = (x \circ y) \circ z \tag{D.1}$$

ii) istnieje element neutralny e

$$\exists_{e \in G} \forall_{x \in G}\ x \circ e = e \circ x = x \tag{D.2}$$

iii) dla każdego elementu istnieje element odwrotny

$$\forall_{x \in G} \exists_{y \in G}\ y \circ x = x \circ y = e \tag{D.3}$$

Grupę nazywamy *przemienną*, jeśli działanie $\circ$ jest przemienne:

$$\forall_{x,y \in G}\ x \circ y = y \circ x \tag{D.4}$$

Pierścień

Strukturę $(P, +, \cdot)$, nazywamy *pierścieniem*, gdy:

i) $(P, +)$ jest grupą przemienną,

ii) działanie $\cdot$ jest łączne

$$\forall_{x,y,z \in P}\ (x \cdot y) \cdot z = x \cdot (y \cdot z) \tag{D.5}$$

iii) działanie $\cdot$ jest rozdzielne względem $+$

$$\forall_{x,y \in P}\ x \cdot (y + z) = (x \cdot y) + (x \cdot z) \tag{D.6}$$

Pierścień jest przemienny, jeśli działanie $\cdot$ jest przemienne. Działanie $\cdot$ nazywa się mnożeniem a jego element neutralny oznacza się zwykle przez 0 i nazywa „zerem”. Działanie $+$ nazywa się dodawaniem, a jego element neutralny oznacza przez 1 i nazywa „jednością”.

Pierścień nazywamy *całkowitym*, jeśli jest przemienny, posiada jedność i nie posiada dzielników zera:

$$\forall_{x,y \in P-\{0\}}\ x \cdot y \neq 0 \tag{D.7}$$

Ciało

Pierścień całkowity $(K, +, \cdot)$, który ma co najmniej dwa elementy nazywamy *ciałem*, gdy:

i) istnieje element neutralny mnożenia

$$\exists_{e \in K} \forall_{x \in K}\ e \cdot x = x \cdot e = x \tag{D.8}$$

ii) mnożenie jest przemienne

$$\forall_{x,y \in K}\ x \cdot y = y \cdot x \tag{D.9}$$

iii) każdy element jest odwracalny względem mnożenia

$$\forall_{x \in K-\{0\}} \exists_{y \in K}\ x \cdot y = 1 \tag{D.10}$$

gdzie 0 jest elementem neutralnym mnożenia.

Ciało nazywamy skończonym (lub ciałem Galois) i oznaczamy przez $GF(q)$, jeśli posiada ono skończoną liczbę q elementów.

Dodatek B. Struktura kodu źródłowego

Kod źródłowy został został zawarty w następujących plikach:

aic.c zawiera funkcje konfiguracyjne kontrolera przerwań (od ang. *Advanced Interrupt Controller*,

atmel-rom.ld – skrypt konsolidatora,

boot.s – plik inicjalizacyjny,

errcode.c zawiera funkcje kodujące i dekodujące,

FreeRTOSConfig.h – konfiguracja systemu FreeRTOS,

isr.c zawiera funkcje obsługi przerwania zewnętrznego,

isrsupport.c – kod odpowiedzialny za uaktywnienie przerwań w mikrokontrolerze,

main.c zawiera deklaracje oraz inicjalizacje kolejek i semaforów, konfiguruje mikroprocesor, uruchamia wszystkie wątki systemu operacyjnego i *scheduler*,

menu.c zawiera wątek do obsługi menu oraz funkcje pomocnicze,

radio.c zawiera kod źródłowy wszystkich wątków i definicje funkcji pomocniczych,

spi.c zawiera funkcje konfigurujące urządzenie SPI,

usart.c zawiera funkcje do obsługi komunikacji szeregowej,

usart_isr.c zawiera procedury obsługi przerwania portu USART.

Wątki (deklaracje i wywołania w pliku *main.c*):

vAT86Application definicja: *radio.c*, l. 60

vAT86Communication def.: *radio.c*, l. 462

vAT86Decoding def.: *radio.c*, l. 323

vAT86Encoding def.: *radio.c*, l. 191

vUSARTMenu , def.: *menu.c*, l. 13

Funkcje pomocnicze, definicje (plik *radio.c*):

AT86_send_string l. 688

AT86CreateStatus l. 664

ErrMaskGen l. 587

WriteReadReg l. 898

Funkcje kodujące, definicje (plik *errcode.c*):

dec_bch32_16 l. 682

dec_hamlin7_4 l. 478

dec_hamlin63_57 l. 517

dec_multiply l. 56

enc_bch32_16 l. 613

enc_hamlin7_4 l. 219

enc_hamlin63_57 l. 355

enc_multiply l. 20

Dodatek C. Instalacja i używanie środowiska programowania opartego o GNU ARM

Wszystkie opisane niżej aplikacje instalowano po kompilacji ze źródeł, pobranych z oficjalnych stron. Zainstalowane zostały w następującej kolejności:

- binutils 2.17 (pobrano z: http://ftp.gnu.org/gnu/binutils/binutils-2.17.tar.gz)
- gcc 4.3.4 (pobrano z: http://gcc.fyxm.net/releases/gcc-4.3.4/gcc-4.3.4.tar.bz2)
- newlib 1.14 (pobrano z: ftp://sources.redhat.com/pub/newlib/newlib-1.14.0.tar.gz)
- gdb 7.0 (pobrano z: http://ftp.gnu.org/gnu/gdb/gdb-7.0.tar.bz2)
- OpenOCD 0.3.0 (do pobrania z: http://prdownload.berlios.de/openocd/openocd-0.3.1.tar.bz2)

Instalacja każdego programu przebiegała zgodnie ze schematem: uruchomienie skryptu konfiguracyjnego z odpowiednimi parametrami, kompilacja, przy użyciu powstałego w wyniku konfiguracji pliku *Makefile*, instalacja w systemie operacyjnym.

Pakiet binutils

Pakiet *binutils* zawiera narzędzia (m. in. asembler i konsolidator) umożliwiające pracę z plikami wykonywalnymi w różnych formatach. Konfiguracja została przeprowadzona z następującymi opcjami:

```
./configure␣--target=arm-elf
--enable-interwork␣--enable-multilib
--disable-werror
```

Kompilator GCC

Kompilator GCC służy do kompilacji źródeł programów napisanych w języku C. W efekcie jego działania powstają pliki wykonywalne dla danej architektury procesora. Standardowo w systemie Linux występuje kompilator GCC zdolny do kompilacji dla architektury macierzystej. Do wykonania projektu potrzebny był jednak kompilator dla architektury ARM. Konfiguracja została przeprowadzona z następującymi opcjami:

```
../configure␣--target=arm-elf
--enable-interwork␣--enable-multilib
--enable-languages=c␣--with-newlib
--with-headers=/home/paszcior/inz/toolchain/
newlib-1.14.0-build/newlib/libc/include/
--disable-werror
```

Jak widać, jedynym językiem programowania zrozumiałym dla kompilatora będzie język C. Przy kompilacji należy podać ścieżkę dostępu do katalogu *include* pakietu *newlib*.

Pakiet newlib

Jest to standardowa biblioteka języka C, z przeznaczeniem dla mikroprocesorów. Wspierana jest między innymi przez twórców architektury ARM. Została skompilowana po przeprowadzeniu następującej konfiguracji

```
./configure␣␣--target=arm-elf
--enable-interwork␣--enable-multilib
```

GDB lub Insight

Insight jest graficzną nakładką na debugger GDB, ale nie wymaga uprzednio zainstalowanego *debuggera*. Można tworzyć kod i programować pamięć flash bez użycia żadnego z tych dwóch narzędzi, jednak mogą się one okazać przydatne przy szukaniu błędów w programie.

Konfiguracja samego *debuggera* GDB powinna zostać przeprowadzona z następującymi opcjami:

```
../configure␣--prefix=/usr/local/arm-elf
--target=arm-elf
--with-gnu-as␣--with-gnu-ld
--enable-multilib␣--enable-interwork
```

natomiast programu Insight:

```
./configure␣--host=x86_64-unknown-linux-gnu
--target=arm-elf
```

OpenOCD

OpenOCD jest aplikacją służącą do komunikacji z pamięcią flash lub SRAM układu mikroprocesora. Można przy jej pomocy zaprogramować te pamięci, jak również *debuggować* program (w połączeniu z *debuggerem*, np. GDB).

W szczególnym przypadku używanego programatora: FTDI-2232 w program należy wkompilować bibliotekę dostarczaną przez producenta. Jest ona dostępna na oficjalnej stronie, także dla systemów Linux: http://www.ftdichip.com/Drivers/D2XX/Linux/libftd2xx0.4.16_x86_64.tar.gz.

Po pobraniu i rozpakowaniu pliku z biblioteką, OpenOCD należy skonfigurować:

```
./configure␣--enable-ft2232_ftd2xx
```

Oprócz zainstalowania konieczne jest również stworzenie pliku konfiguracyjnego, który załaduje plik do pamięci lub umożliwi podgląd pracy procesora debuggerowi. Plik konfiguracyjny tworzy się na podstawie gotowych konfiguracji dla poszczególnych elementów: procesora i programatora. W przypadku programatora należy zdefiniować w pliku konfiguracyjnym jego parametry:

```
interface␣ft2232
ft2232_layout␣usbjtag
ft2232_vid_pid␣0x0403␣0x6010
ft2232_device_desc␣"Dual␣RS232␣A"
```

a w przypadku mikroprocesora, o ile jest wspierany, wystarczy podać nazwę odpowiedniej, gotowej konfiguracji, np.:

```
source␣[find␣target/sam7se512.cfg]
```

Zapis programu do pamięci flash następuje przy pomocy komendy:

```
flash␣write_bank␣0␣nazwa_pliku.bin␣0x0
```

gdzie ostatnim elementem jest adres, pod który należy zapisać dane.

Sposób użycia

Aby dokonać kompilacji przy użyciu zainstalowanego kompilatora GCC dla architektury ARM, należy wywołać polecenie *arm-elf-gcc*. Podobnie *debugger* zostaje podczas instalacji domyślnie dowiązany do polecenia *arm-elf-gdb*.

W chwili kiedy gotowy jest skompilowany program należy załadować go do pamięci. W tym celu potrzebny jest skrypt konfiguracyjny OpenOCD, opisany w poprzednim punkcie. Aplikację wywołuje się poleceniem:

```
openocd␣-f␣nazwa_skryptu
```

Do podglądu pracy mikroprocesora przy pomocy *debuggera*, należy stworzyć osobny skrypt, który zawiesi jego pracę i pozwoli ją krokowo kontrolować. Po wywołaniu OpenOCD z takim skryptem jako argumentem, należy uruchomić program GDB lub Insight i połączyć się na wcześniej ustalonym porcie Telnet lub TCP/IP z OpenOCD. Dalej postępuje się tak samo jak w przypadku normalnego szukania błędów tą metodą – z uwzględnieniem specyfiki danego mikrokontrolera – na przykład ograniczonej ilości pułapek sprzętowych.

Dodatek D. Obsługa aplikacji

Stworzona w ramach programu aplikacja służy do testów urządzenia i prezentacji jego pracy. Aby ją uruchomić należy skompilować kod źródłowy kompilatorem GCC[1], korzystając z pliku *Makefile* umieszczonego w katalogu projektu oraz wgrać gotowy plik binarny do pamięci flash układu AT91SAM7S. Po resecie urządzenia program zacznie się wykonywać automatycznie, poczynając od wyświetlenia menu.

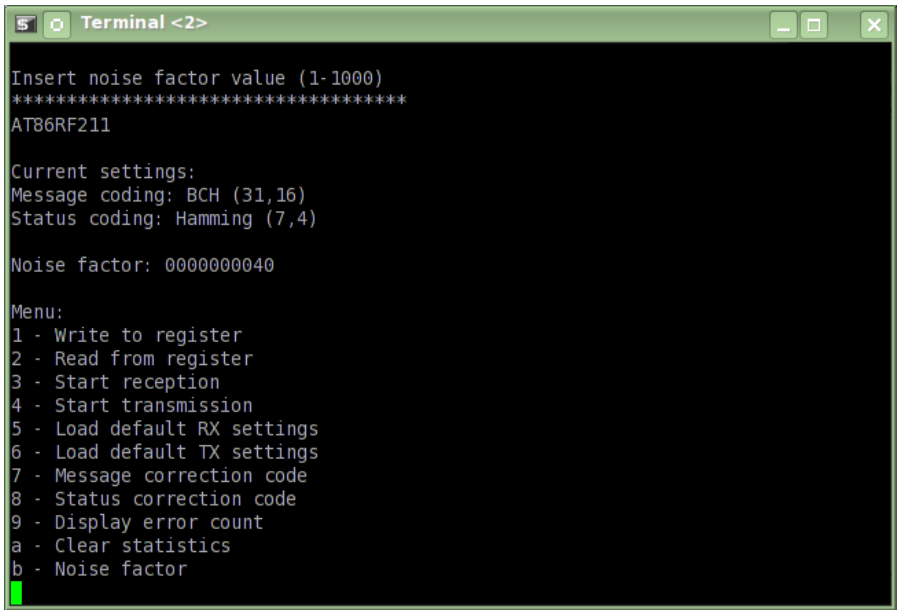
Do komunikacji z układem niezbędny jest program do obsługi transmisji szeregowej USART, skonfigurowany następująco:

- prędkość transmisji: 115200 kb/s,
- liczba bitów w wiadomości: 8,
- kontrola bitów parzystości: wyłączona,
- liczba bitów stopu: 1.

Obsługa aplikacji

Na rysunku 6.1 przedstawiono wygląd menu składającego się z 11 pozycji:

Write to register – umożliwia zapis danych do rejestrów konfiguracyjnych AT86RF211. Program zapyta kolejno o adres rejestru, który użytkownik chce zapisać, o długość wiadomości oraz o samą wiadomość.



Rysunek 6.1. Zrzut ekranu przedstawiający wygląd menu aplikacji w programie Minicom.

[1] przy pisaniu oprogramowania używany był kompilator GCC w wersji 4.3.4

Read from register – pozwala na odczyt danych z rejestru konfiguracyjnego. Użytkownik musi podać adres rejestru oraz liczbę bajtów, które należy odczytać. Wynik odczytu zostaje natychmiast wydrukowany na ekranie.

Start reception – układ zostanie automatycznie skonfigurowany do odbioru i zacznie oczekiwać na wiadomość. Podczas jej dekodowania uzupełniane są statystyki dotyczące skuteczności kodu korekcyjnego, a po zdekodowaniu otrzymana wiadomość zostaje porównana z oczekiwaną i wyświetlona na ekranie.

Start transmission – następuje konfiguracja układu do nadawania i wysyłana zostaje cyklicznie wiadomość zawarta w zmiennej *napis_out*. Przed fizycznym wysłaniem każdego bajtu, jest on zakłócony maską uzyskaną w wyniku działania funkcji *ErrMaskGen*. Prawdopodobieństwo zakłócenia pojedynczego bitu regulowane jest zmienną *global_noise* i wynosi $p = \frac{global\_noise}{1000}$. Drobne zakłócenia, rzędu kilku promili, są widoczne przy szybszej transmisji danych na porcie USART.

Load default RX settings – ładuje do rejestrów konfiguracyjnych domyślne wartości dla trybu odbioru.

Load default TX settings – ładuje do rejestrów konfiguracyjnych domyślne wartości dla trybu nadawania.

Message correction code – umożliwia zmianę stosowanego kodu korekcyjnego w kanale głównym.

Status correction code – umożliwia zmianę kodu korekcyjnego stosowanego w kanale zwrotnym.

Display error count – wyświetla statystyki powstające podczas dekodowania wiadomości.

Clear statistics – zeruje powyższe statystyki.

Noise factor – ustawia nową wartość współczynnika *global_noise*, regulując tym samym poziom sztucznych zakłóceń w kanale. Należy wpisać wartość od 0 do 1000[2].

Menu jest zabezpieczone przed wciśnięciem przypadkowego klawisza.

W trakcie nadawania lub transmisji możliwy jest powrót do menu w dowolnym momencie – po naciśnięciu klawisza na klawiaturze. Możliwe jest wielokrotne przerywanie i wznawianie transmisji lub odbioru.

Użycie innego kodu korekcyjnego do nadawania i innego do odbioru jest możliwe, ale prowadzi do niepowodzenia transmisji. Aby można było po takiej sytuacji uzyskać poprawną komunikację, konieczne jest jej przerwanie w obu węzłach i ponowne uruchomienie. Podczas przerywania transmisji lub odbioru czyszczone są bowiem wszystkie kolejki systemu czasu FreeRTOS, które mogły zawierać błędne dane.

[2] gdzie 0 to brak zakłóceń, a 1000 to 100% bitów zakłóconych