3 Obsługa złożonych czujników i urządzeń wykonawczych mikroprocesorowego systemu automatyki.

Celem tego ćwiczenia jest zapoznanie studenta z popularnymi standardami przekazywania informacji w przemyśle. Jako klasyczny standard do analogowej transmisji danych użyty zostanie standard 4-20 mA, natomiast przykładem transmisji cyfrowej będzie MODBUS RTU. Jest to oczywiście znikomy podzbiór standardów komunikacyjnych, lecz pozwala on uświadomić, że nawet wyjątkowo proste rozwiązania mogą być i są skutecznie implementowane w przemyśle.

Pętla prądowa 4-20 mA. Pomiar z wykorzystaniem standardu 4-20 mA jest wyjątkowo łatwy w realizacji o ile opanowana została umiejętność pomiaru napięcia na jednym z pinów mikrokontrolera. Ponieważ rozważany zestaw rozwojowy ZL27ARM nie posiada możliwości bezpośredniego pomiaru prądu, należy wykorzystać znajomość prawa Ohma. Załóżmy, że posiadamy opornik o oporze R, przez który płynie prąd I. Napięcie na tym oporniku U wyrażone jest wzorem:

$$U = I \cdot R$$

Ponieważ rezystancja opornika jest stała (nie zależy od napięcia ani prądu), stąd wynika, że napięcie na tym oporniku jest wprost proporcjonalne do płynącego przez niego prądu. Natężenie prądu, które chcemy zmierzyć przyjmuje wartości od 4 do 20 mA. Mikrokontroler, którym się posługujemy jest w stanie mierzyć napięcie z zakresu 0 do 3.3V. Aby więc prądowi 20 mA odpowiadało napięcie 3.3V należy zastosować opornik o wartości:

$$R = \frac{U^{\max}}{I^{\max}} = \frac{3.3V}{0.02A} = 165\Omega$$

Dla prądu o wartości 4 mA uzyskane zostanie napięcie:

$$U = 0.004A \cdot 165\Omega = 0.66V$$

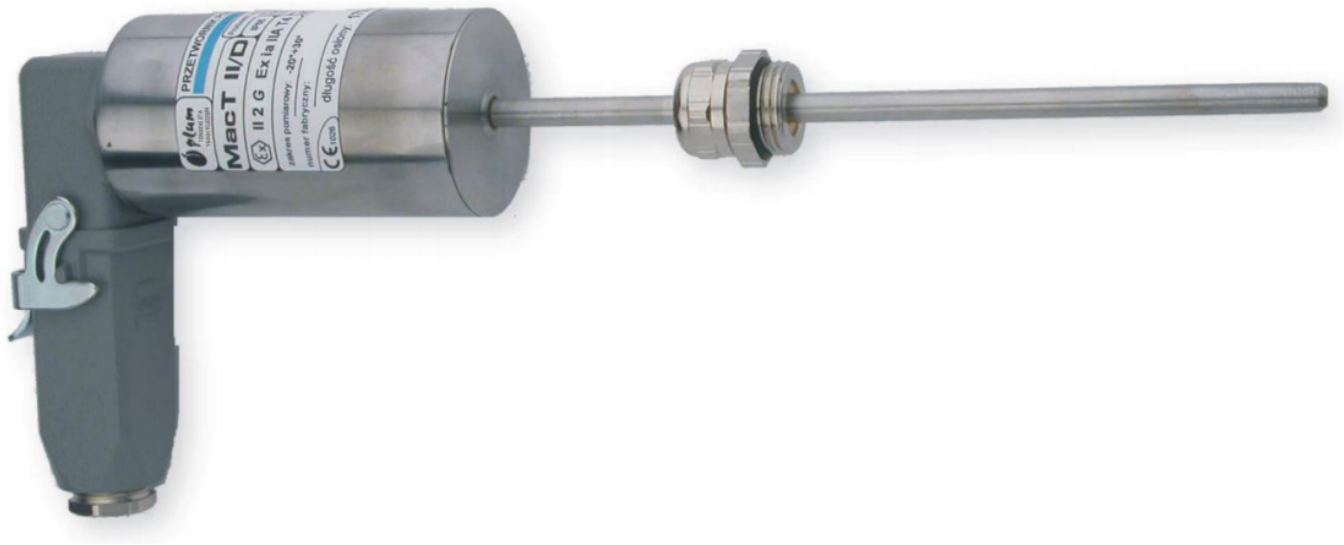
Tak dobrany rezystor posłuży do „zamiany" wartości prądu na napięcie. W razie braku opornika o stosownej wartości należy dobrać rezystor o **mniejszej** wartości, aby nie przekroczyć napięcia zasilania mikrokontrolera. W przypadku tego ćwiczenia wykorzystany zostanie rezystor o wartości 160Ω.

Zastosowanie prądu do reprezentacji pomiaru posiada wiele pożytecznych (szczególnie w przemyśle) cech. Między innymi:

- odczyt wartości prądu poniżej 4 mA oznacza uszkodzenie czujnika lub instalacji,
- podłączenie czujnika jest tanie i łatwe w realizacji,
- wpływ rezystancji linii pomiarowej na odczyt jest niewielki,
- zasilanie i odczyt pomiaru realizowany może być przy użyciu 2 kabli,
- takie podłączenie cechuje się bardzo wysoką odpornością na zakłócenia.

W związku z powyższym przemysłowe czujniki mogą być podłączone bardzo długimi kablami do urządzeń odczytujących pomiary. Pomiar z wykorzystaniem napięcia w takiej sytuacji byłby obarczony znaczącym błędem.

Implementacja pomiaru z użyciem pętli prądowej 4-20mA nie zostanie przytoczona, gdyż jest ona identyczna jak pomiar napięcia.

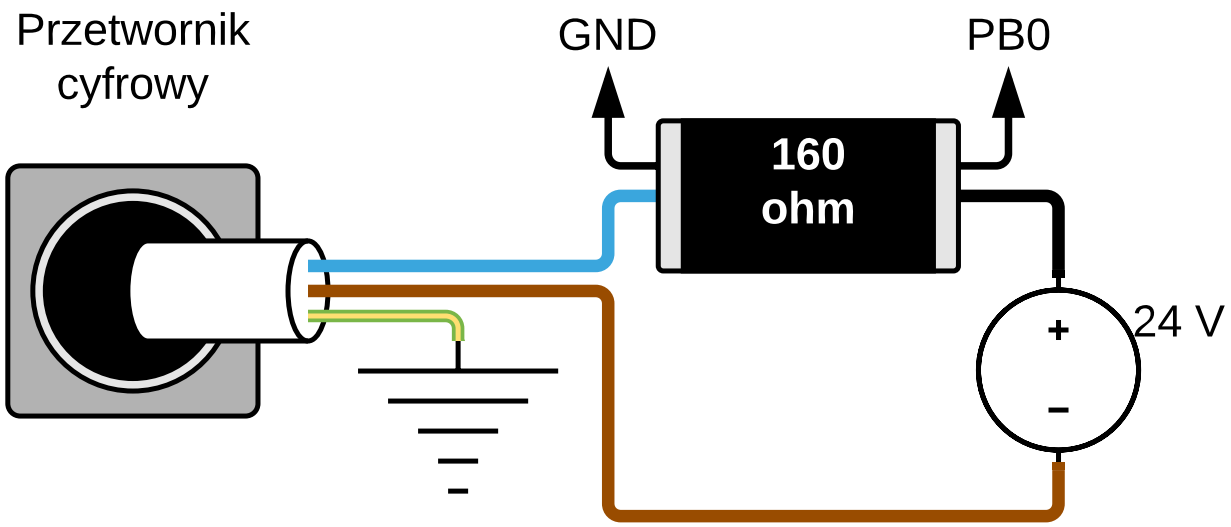


Rysunek 30: Cyfrowy przetwornik temperatury typu MacT II firmy Plum.

Cyfrowy przetwornik temperatury z termometrem rezystancyjnym. W tym ćwiczeniu jako przykład urządzenia komunikującego przy użyciu pętli prądowej 4-20mA posłuży cyfrowy przetwornik temperatury typu MacT II firmy Plum, z termometrem rezystancyjnym Pt100 (Rys. 30). Jest on wyposażony w mikrokontroler, który steruje pomiarami oraz kompensuje charakterystyki układu pomiarowego oraz pętli prądowej, dzięki temu została osiągnięta wysoka dokładność przetwornika. Jest to sprzęt wykorzystywany w przemyśle, przeznaczony do pomiaru temperatury dowolnych mediów, na które odporna jest obudowa przetwornika wykonana ze stali kwasoodpornej. Został wykonany jako urządzenie iskrobezpiecznie i może pracować w strefach zagrożenia wybuchem 1 i 2.

Powyższy przetwornik należy zasilić stałym napięciem od 9 do 30V, przy czym należy mieć na uwadze, iż w zależności od napięcia zasilania dopuszalne są różne wartości rezystancji w linii zasilającej. Oznacza to, że w szereg z termometrem można wstawić tylko odpowiednio małe rezystory, które następnie mogą być wykorzystane do pomiarów. Wcześniej ustalone zostało, że użyty zostanie opornik o wartości 160Ω, co (zgodnie z dokumentacją) oznacza, że przetwornik należy zasilić napięciem trochę ponad 13V. Z tego powodu do zasilania zostanie wykorzystany zasilacz o napięciu 24V, mogący dostarczyć maksymalnie 0.6A. Połączenie przetwornika do zasilacza przedstawione jest na Rys. 31. Należy zwrócić szczególną uwagę na podłączenie do mikrokontrolera – zamiana masy płytki z pinem wejściowym może spowodować uszkodzenie płytki rozwojowej. Dla ułatwienia podłączeń została zrealizowana płytka ze złączami śrubowymi, które tutaj posłużyły do połączenia kabli od przetwornika do płytki i od płytki do zasilacza, przy czym jedno z połączeń posiada w szeregu wlutowany opornik widoczny na schemacie.

Ostatnim ważnym aspektem jest kwestia translacji wartości zmierzonego prądu/napięcia na faktyczną temperaturę. Termometr potrafi zmierzyć wartość temperatury od -30°C do 60°C. Minimalnej wartości prądu wytwarzanej przez przetwornik (4mA) odpowiadać będzie więc -30°C, natomiast maksymalnej



Rysunek 31: Schemat podłączenia przetwornika do zasilacza w celu pomiaru temperatury.

(20mA) 60°C. Wartość prądu w funkcji temperatury przedstawia się więc następującym wzorem:

$$\begin{aligned} I(temp) &= \frac{temp°C - (-30°C)}{60°C - (-30°C)}(20mA - 4mA) + 4mA \\ &= \frac{temp + 30}{90}16mA + 4mA \end{aligned}$$

Oczywiście natychmiastowo można wyznaczyć również wzór na napięcie odłożone na oporniku w funkcji temperatury:

$$\begin{aligned} U(temp) &= \left(\frac{temp + 30}{90}16mA + 4mA\right)160\Omega \\ &= \frac{temp + 30}{90}2.56V + 0.64V \end{aligned}$$

Warto jednak zauważyć, że w związku z użyciem mniejszego opornika niż było oryginalnie planowane, wartość napięcia dla maksymalnej mierzonej temperatury nie będzie równa 3.3V, lecz $U(60) = 2.56V + 0.64V = 3.2V$.

Na koniec pozostaje kwestia reprezetancji cyfrowej takiego pomiaru. Ponieważ w rozważanym mikrokontrolerze wykorzystany jest 12bitowy przetwornik analogowo cyfrowy, pomiar napięcia może być reprezentowany przez wartości od 0 (0V) do 4095 (3.3V). A więc wartość otrzymana na mikrokontrolerze w

funkcji temperatury mierzonej wygląda następująco:

$$\begin{aligned} U^{\mathrm{c}}(temp) &= \left(\frac{temp+30}{90}2.56V + 0.64V\right)\frac{4095}{3.3V} \\ &= \left(\frac{temp+30}{90}2.56 + 0.64\right)\frac{4095}{3.3} \end{aligned}$$

Ponieważ jednak wartość U^{c} (oznaczenie $^{\mathrm{c}}$ podkreśla cyfrową reprezentację napięcia) jest z założenia liczbą całkowitą, należy wynik zaokrąglić. Wynik uzyskany na mikrokontrolerze może się nieznacznie różnić w stosunku do uzyskanego przy użyciu powyższego wyliczenia, lecz jest to związane z odpornością układu na zakłócenia, który to temat nie będzie poruszany.

Wszystkie powyższe wyprowadzenia służą do tego, aby zamienić temperaturę na 12 bitową wartość cyfrową. W praktyce przyda się jednak funkcja odwrotna, pozwalająca na podstawie odczytanej wartości 12 bitowej określić jaką to reprezentuje temperaturę.

$$temp\,(U^{\mathrm{c}}) = \left(U^{\mathrm{c}}\frac{3.3}{4095} - 0.64\right)\frac{90}{2.56} - 30$$

Oczywiście przed implementacją warto uprościć tę funkcję, tak aby była wygodniejsza w implementacji i nie wymagała tak dużej liczby obliczeń.

Transmisja szeregowa. Transmisja szeregowa (w przeciwieństwie do transmisji równoległej) polega na sekwencyjnym przesyłaniu kolejnych bitów danych. Oznacza to, że bity nadchodzą jeden za drugim w ustalonej kolejności przy użyciu jednego połaczenia. W przypadku transmisji równoległej, jednocześnie przesyłanych jest wiele bitów poprzez wykorzystanie wielu połączeń (tak jest w przypadku komunikacji z wyświetlaczem LCD znajdującym się na płytce ZL27ARM).

Oczywiście poprzez „przesył danych" należy rozumieć, że urządzenie nadawcze ustala napięcie na linii służącej do transmisji, a urzadzenie odbiorcze dokonuje pomiaru tego napięcia. Transmisja cyfrowa oznacza, że dane mogą być reprezentowane wyłacznie jako 0 lub 1 logiczne (tj. napięcie poniżej lub powyżej pewnego, wcześniej ustalonego poziomu napięcia). Niewielkie zakłócenia nie powodują problemów z taką transmisją, gdyż wspomniany próg napięcia rozróżniający 0 i 1 logiczną często znajduje się w połowie przedziału dozwolonego napięcia.

Transmisja szeregowa może być realizowana w trybie synchronicznym lub asynchronicznym. W pierwszym przypadku, wykorzystując dodatkowe połączenie, przesyłany jest sygnał zegarowy. Sygnał ten służy do wyznaczania chwil, w których transmisja danych jest w stanie gotowości do odczytu/zapisu. Odbiorca i nadawca są zobowiązani do synchronizacji zegarów tak, aby odbiorca odbierał dane wyłącznie wtedy gdy nadawca te dane wysyła.

W dalszej części jednak skupienie padnie na komunikację asynchroniczną, tj. pozbawioną dodatkowego zegara taktującego. Komunikacja w tym przypadku odbywa się na podstawie założenia, że obiorca i nadawca mają tak samo skonfigurowane zegary, na podstawie których będą wyznaczane chwile służące do nadawania/odbierania kolejnych bitów. Do określenia częstotliwości tych zegarów określa się wartość *baudrate*, która oznacza „liczbę zmian medium transmisyjnego na sekundę". W przypadku przesyłu binarnych wartości (bitów) można tę wartość utożsamić z bitami na sekundę. Popularne wartości, które przyjmuje się jako *baudrate* są następujące: 1200, 2400, 4800, 9600, 19200, 38400, 57600 i 115200.

Istnieją trzy możliwości zestawienia transmisji szeregowej przy użyciu trybu synchronicznego: *simplex*, *half-duplex* oraz *full-duplex*. Pierwszy z nich oznacza, że transmisja odbywa się przy użyciu jednego połaczenia i jest jednokierunkowa (jedno z urządzeń zawsze wyłącznie nadaje). Transmisja half-duplex oznacza transmisję dwukierunkową przy użyciu jednego, dzielonego połączenia. Stąd też jednoczesne nadawanie i odbieranie jest niemożliwe. Ostatni tryb pozwala na jednoczesne nadawanie i odbieranie danych ze względu na wykorzystanie dwóch osobnych połączeń przeznaczonych na komunikację w każdą ze stron. W dalszej części skupimy się na komunikacji *full-duplex*.

Kolejnymi parametrami transmisji szeregowej są długość porcji danych, konfiguracja bitów parzystości i liczba bitów stopu. Długość porcji danych może być równa od 5 do 8 bitów. Przesył znaków ASCII często realizuje się na 7 bitach, gdyż tyle właśnie zajmuje jeden taki znak.

Bit parzystości służy jako prosty mechanizm sprawdzania poprawności danych. Są trzy (podstawowe) możliwości konfiguracji tego bitu:

- brak – do danych nie będzie dodawana informacja o ich poprawności,
- parzystość – do danych będzie dodawana informacja o tym, czy liczba jedynek w części danych jest parzysta (0 jeśli jest, 1 w przeciwnym wypadku),
- nieparzystość – do danych będzie dodawana informacja o tym, czy liczba jedynek w części danych jest nieparzysta (0 jeśli jest, 1 w przeciwnym wypadku).

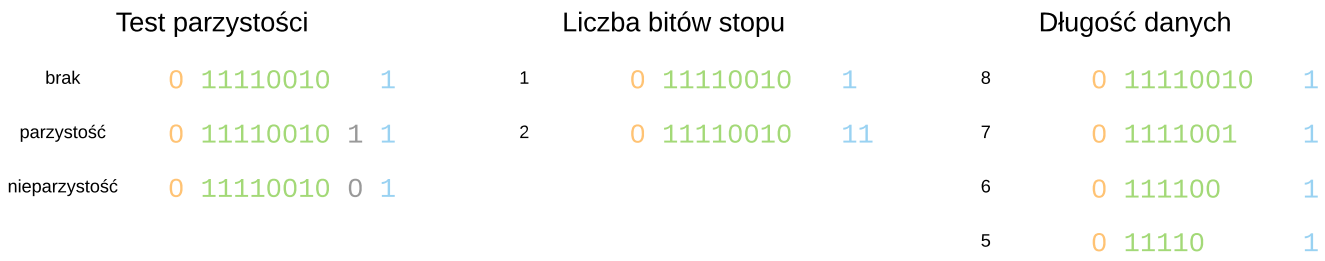
Ostatnim parametrem jest liczba bitów stopu. Tutaj są tylko dwie opcje: może ich być 1 lub 2.

Transmisja szeregowa rozpoczyna się od bitu startu – zera logicznego. Następnie przesyłane są kolejne bity danych, ewentualny bit parzystości i bit(y) stopu – jedynka(-i) logiczne. Dla przykładu rozważmy konfigurację 8N1 (najpopularniejsza konfiguracja, tj. 8 bitów na dane, brak testu parzystości, 1 bit stopu). Pojedyncza wiadomość będzie składała się z:

- 1 bitu startu,
- 8 bitów danych,
- 0 bitów parzystości,
- 1 bitu stopu.

W sumie będzie to wiadomość o długości 10 bitów. Przykładowa wiadomość wygląda jak na Rys. 32 (pierwsza wiadomość w każdej kolumnie). Zakładając, że przesyłamy dane z prędkością 9600 (*baudrate*), możemy stwierdzić, że pojedynczy bajt wiadomości przesyłamy z prędkością $9600/10 = 960$ bajtów na sekundę, a więc jeden bajt wysyłany jest co $1/960 \approx 1.04$ms.

Implementacja na ZL27ARM Implementację transmisji szeregowej z użyciem modułu USART (*Universal Synchronus and Asynchronous Receiver and Transmitter*) należy rozpocząć od wyboru wolnych pinów, które posiadają możliwość pracy jako pin nadawczy i odbiorczy modułu USART. Do takich pinów należą między innymi piny PA9 (nadawczy) i PA10 (odbiorczy) – są one częścią USART1. Ponieważ komunikacja szeregowa jest funkcją alternatywną tych pinów należy je odpowiednio skonfigurować. Przedtem jednak należy zadbać o dołączenie do projektu plików do obsługi USART (tj. `stm32f10x_usart.*`) oraz włączenie odpowiednich modułów, tj.:



Rysunek 32: Możliwości konfiguracji wiadomości w transmisji szeregowej. Oznaczenia kolorystyczne: pomarańczowy – bit startu, zielony – dane, szary – bit parzystości, niebieski – bity stopu.

```
RCC_APB2PeriphClockCmd(RCC_APB2Periph_AFIO , ENABLE);  // wlacz taktowanie AFIO
RCC_APB2PeriphClockCmd(RCC_APB2Periph_GPIOA, ENABLE);  // wlacz taktowanie GPIOA
RCC_APB2PeriphClockCmd(RCC_APB2Periph_USART1, ENABLE); // wlacz taktowanie USART1
```

Następnie należy przejść do właściwej konfiguracji pinów do komunikacji z użyciem USART1:

```
GPIO_InitTypeDef GPIO_InitStruct;

// Pin nadawczy należy skonfigurować jako "alternative function, push-pull"
GPIO_InitStruct.GPIO_Pin = GPIO_Pin_9;
GPIO_InitStruct.GPIO_Mode = GPIO_Mode_AF_PP;
GPIO_InitStruct.GPIO_Speed = GPIO_Speed_50MHz;
GPIO_Init(GPIOA, &GPIO_InitStruct);

// Pin odbiorczy należy skonfigurować jako wejście "pływające"
GPIO_InitStruct.GPIO_Pin = GPIO_Pin_10;
GPIO_InitStruct.GPIO_Mode = GPIO_Mode_IN_FLOATING;
GPIO_InitStruct.GPIO_Speed = GPIO_Speed_50MHz;
GPIO_Init(GPIOA, &GPIO_InitStruct);
```

Dalej następuje konfiguracja samej transmisji szeregowej:

```
USART_InitTypeDef USART_InitStruct;
USART_InitStruct.USART_BaudRate = 19200;
USART_InitStruct.USART_HardwareFlowControl = USART_HardwareFlowControl_None;
USART_InitStruct.USART_WordLength = USART_WordLength_9b;
USART_InitStruct.USART_Parity = USART_Parity_Even;
USART_InitStruct.USART_StopBits = USART_StopBits_1;
USART_InitStruct.USART_Mode = USART_Mode_Tx | USART_Mode_Rx;

USART_Init(USART1, &USART_InitStruct);
USART_ITConfig(USART1, USART_IT_RXNE, DISABLE);
USART_ITConfig(USART1, USART_IT_TXE, DISABLE);
```

Kolejne linie oznaczają prędkość transmisji, tj. *baudrate* 19200, brak sprzętowej kontroli przepływu danych, 8 bitów na dane, test parzystości, 1 bit stopu, transmisja w obie strony. Następnie następuje przypisanie

powyższej konfiguracji do USART1 i wyłączenie przerwań związanych z odbiorem (`RXNE` – *RX buffer Not Empty*) i nadawaniem (`TXE` – *TX buffer Empty*). Przerwania te są wyzwalane odpowiednio w chwili gdy bufor odbiorczy przestanie być pusty, bufor nadawczy zostanie opróżniony (tj. wszystkie dane zostaną wysłane). Warto zwrócić uwagę na konfigurację długości danych – wartość ta w mikrokontrolerach rodziny STM32 oznacza długość danych wraz z bitem parzystości. A więc jeśli bit parzystości jest wykorzystywany, należy pamiętać o dodaniu tego bitu do długości słowa (jak to jest nazwane w standardowej bibliotece peryferiali).

W dalszej implementacji wykorzystane zostaną oczywiście przerwania związane z komunikacją, lecz muszą one zostać użyte w taki sposób, aby nie wysyłać gdy nie ma nic do wysłania i nie odbierać gdy niczego się nie spodziewamy. Ponieważ mowa o przerwaniach to konieczne jest również nadanie priorytetu przerwaniu:

```
NVIC_InitStructure.NVIC_IRQChannel = USART1_IRQn;
NVIC_InitStructure.NVIC_IRQChannelPreemptionPriority = 0;
NVIC_InitStructure.NVIC_IRQChannelSubPriority = 0;
NVIC_InitStructure.NVIC_IRQChannelCmd = ENABLE;
NVIC_Init(&NVIC_InitStructure);
```

Dopiero po tym etapie należy włączyć USART:

```
USART_Cmd(USART1, ENABLE);
```

Transmisja szeregowa – standard MODBUS RTU. Implementacja protokołu MODBUS RTU została opracowana na podstawie dokumentów: *MODBUS over Serial Line: Specification & Implementation guide V1.0* (obecnie dostępna jest nowsza wersja) oraz *MODBUS Application Protocol Specification V1.1b3*. Oba dokumenty są dostępne na stronie `www.modbus.org`. Dodatkową inspiracją przy opracowywaniu użytej niżej implementacji była implementacja o nazwie FreeMODBUS (`www.freemodbus.org`) – nie zawierała ona jednak wygodnego mechanizmu wysyłania żądań (tj. pracy w trybie *master*). Omawiana implementacja jest jednocześnie uproszczona na tyle, aby można było bez większych trudności zrozumieć zasadę działania tego protokołu. Warto dodatkowo zwrócić uwagę na fakt, że implementacja FreeMODBUS nie uwzględnia wykorzystania przerwy między poszczególnymi znakami oznaczonej w dokumentacji jako t1,5. Protokół MODBUS pozwala na dużą elastyczność w implementacji, dlatego niestety należy mieć na uwadze, że dwie różne implementacje tego protokołu mogą nie współpracować ze sobą w pełni. Wciaż jednak jest to jeden z najbardziej popularnych protokołów w przemyśle.

Implementacja protokołu MODBUS RTU (nie została zaimplementowana wersja ASCII ani TCP/IP) jest tak naprawdę implementacją maszyny stanów opisanej w dokumentacji tego protokołu (Rys. 14 z *MODBUS over serial line...*). W zależności od stanu i kontekstu, wykonywane są poszczególne operacje przejścia między stanami, co pozwala na wygodne i niemal jednoznaczne ustalenie przyczyny błędu w razie jego występowania. Aby aktualizować stan wspomnianej maszyny stanów należy regularnie wywoływać `void MB(void)`. Wywoływanie tej funkcji powinno odbywać się nie rzadziej niż odświeżany jest timer odpowiedzialny za wyznaczanie czasu t1,5 oraz t3,5 (omówione zostaną za chwilę). Z poziomu urządzenia typu *master* (gdyż taki będzie nas interesował) nie można wysyłać jednocześnie dwóch wiadomości. Co więcej należy uważać, aby nie podjąć takiej próby, gdyż obecna implementacja nie jest na to odporna – obowiązkiem użytkownika i programisty jest zachowanie ostrożności lub zapewnienie sobie takiego środo-

wiska, w którym nie dojdzie do wysłania dwóch osobnych zapytań w jednym momencie (dokładniej: nie kolejno). Wynika to z prostego mechanizmu tworzenia wiadomości – istnieje bufor, w którym przechowywane są zarówno bajty wychodzące jak i przychodzące, w zależności od stanu maszyny stanów. Próba wielokrotnego wysłania danych (lub wysłania danych w trakcie odbierania odpowiedzi) może spowodować nadpisanie niewysłanej lub nieprzetworzonej ramki danych. W związku z tym należy pamiętać, że protokół MODBUS jest protokołem typu *master-slave*, a co za tym idzie, *master* wysyła zapytanie i oczekuje na odpowiedź od *slave'a*. Dopiero taka para zdarzeń powoduje, że można ponownie wysłać zapytanie.

Wspomniane zostało, że funkcja aktualizująca maszynę stanów powinna być wywoływana nie rzadziej niż odświeżany jest pewien timer. Wymieniony timer służy od wyznaczania czasów t1,5 oraz t3,5, które (zgodnie z dokumentacją) oznaczają czas potrzebny na przesłanie pojedynczego znaku przemnożony przez kolejno 1.5 oraz 3.5. Czasy te wyznaczają moment, kiedy zakończona została transmisja pojedynczej wiadomości (t1,5) oraz czas między poszczególnymi wiadomościami (t3,5). Mimo bardzo zbliżonego znaczenia (dlatego prawdopodobnie FreeMODBUS nie wykorzystuje t1,5), ich rozróżnienie jest wyraźne w dokumentacji. Aby odmierzyć ten czas można wykorzystać albo dwa timery (dla każdego z czasów osobny) lub jednego, który będzie zliczał małe kwanty czasu, co pozwoli na ustalenie kiedy minęło t1,5 oraz t3,5. Drugie rozwiązanie ma dwie kluczowe zalety – oszczędność timerów oraz elastyczność. Ponieważ MODBUS może pracować przy różnych prędkościach przesyłu danych, także i czasy t1,5 i t3,5 są zmienne. Zamiast komplikować procedurę inicjalizacji timerów można modyfikować jedynie wartość liczników. Stąd bierze się zalecenie dotyczące częstotliwości wywoływania funkcji `MB()` – jeśli będzie ona wywoływana rzadziej niż pojedynczy kwant zliczany przez timer, możemy odczekać więcej czasu niż było w założeniu. Może to nie być zauważalne, lecz dla zwiększenia niezawodności implementacji warto przestrzegać możliwie dokładnie limitów czasowych. W tej implementacji wykorzystany zostanie timer, który odliczać będzie kwanty 50µs – taka sama lub wyższa częstotliwość jest zalecana do wywoływania funkcji `MB()`.

Konstrukcja wiadomości w protokole MODBUS RTU przedstawiona jest w pliku *MODBUS Application Protocol Specification...* w rozdziale *6 Function codes descriptions*. Opis jest wyjątkowo prosty, co pozwala na bezproblemową ręczną konstrukcję wiadomości.

Do wysyłania wiadomości w trybie *master* wykorzystana jest funkcja:

```
void MB_SendRequest(uint8_t addr, MB_FUNCTION f, uint8_t* datain, uint16_t lenin)|
```

która jako kolejne parametry przyjmuje:

- adres urzadzenia typu *slave*,
- identyfikator funkcji protokołu MODBUS,
- adres tablicy zawierającej treść wiadomości,
- długość treści wiadomości.

Bajty CRC są dodawane automatycznie. Identyfikatory funkcji zostały zaimplementowane w postaci zmiennej wyliczeniowej (`enum`) w pliku `main.h`. Podobnie wygląda funkcja służąca do odebrania odpowiedzi:

```
MB_RESPONSE_STATE MB_GetResponse(uint8_t addr, MB_FUNCTION f,
                                 uint8_t** dataout, uint16_t* lenout, uint32_t timeout)
```

która także przyjmuje jako pierwsze parametry adres oraz identyfikator funkcji protokołu MODBUS – wykorzystane jest to do sprawdzenia poprawności odpowiedzi. Pozostałymi parametrami są:

- adres na zmienną, do której ma być wpisany adres tablicy, w której znajduje się treść odpowiedzi,
- adres na zmienną, do której ma być wpisana długość treści odpowiedzi,
- wartość w milisekundach określająca ile czasu mikrokontroler ma oczekiwać na odpowiedź od urządzenia typu *slave*.

Funkcja ta zwraca stan odpowiedzi `MB_RESPONSE_STATE`, który może przyjmowac jedną z następujących wartości:

- `RESPONSE_OK` – odpowiedź poprawna,
- `RESPONSE_TIMEOUT` – czas oczekiwania na odpowiedź został przekroczony,
- `RESPONSE_WRONG_ADDRESS` – odpowiedź zawiera inny adres urządzenia typu *slave* niż oczekiwany,
- `RESPONSE_WRONG_FUNCTION` – odpowiedź zawiera inny identyfikator funkcji niż oczekiwany,
- `RESPONSE_ERROR` – urządzenie typu *slave* zgłosiło błąd (typ błędu zawarty jest w treści wiadomości).

Wartość argumentu określającego czas oczekiwania na wiadomość powininna wynosić około 1s, lecz dokładny czas oczekiwania nie został zdefiniowany (w dokumentacji można znaleźć jedynie sugestie – rozdział 2.4.1 dokumentu *MODBUS over serial line...*)

Dla przykładu, gdyby chcieć z urządzenia *master* wysłać do urządzenia *slave* o adresie 103 wiadomość ustawienia wartości cewki 3. na 1, należałoby wywołać następujący kod:

```
MB_SendReqest(103, FUN_WRITE_SINGLE_COIL, write_single_coil_3, 4);
```

gdzie `write_single_coil_3` zdefiniowane jest jako:

```
uint8_t write_single_coil_3[] = {0x00, 0x03, 0xFF, 0x00};
```

W odpowiedzi otrzymane zostanie echo wiadomości nadanej:

```
uint8_t *resp;
uint16_t resplen;
MB_RESPONSE_STATE respstate;
respstate = MB_GetResponse(103, FUN_WRITE_SINGLE_COIL, &resp, &resplen, 1000);
```

a więc wartości znajdujące się w tablicy `resp` powinny pokrywać się z zawartością tablicy `write_single_coil_3`.

Z drugiej strony, gdyby chcieć odczytać z tego urządzenia *slave* wartości przez niego zmierzone (np. dyskretne wejście 4), należałoby wysłać następującą wiadomość:

```
MB_SendRequest(103, FUN_READ_DISCRETE_INPUTS, read_discrete_input_4, 4);
```

gdzie

```
uint8_t read_discrete_input_4[] = {0x00, 0x04, 0x00, 0x01};
```

Odpowiedź urządzenia *slave* odbierana jest przy użyciu:

```
uint8_t *resp;
uint16_t resplen;
MB_RESPONSE_STATE respstate;
respstate = MB_GetResponse(103, FUN_READ_DISCRETE_INPUTS, &resp, &resplen, 1000);
```

gdzie wartość cewki znajduje się na najmniej znaczącym bicie w tablicy spod adresu `resp`. Analogicznie można przeprowadzić zapis i odczyt wartości 16-bitowych – informacje o strukturze wiadomości znajdują się w dokumencie *MODBUS Application Protocol Specification....* Z powyższego wynika pewna niedogodność – to użytkownik jest odpowiedzialny za uzupełnienie ramki zgodnej z protokołem MODBUS RTU w treść odpowiadającą wykorzystywanej funkcji protokołu, gdyż dostarczona jest jedynie funkcja do przesyłania ramek, a nie uzupełniania ich treści.

Proponowana implementacja protokołu MODBUS RTU zrealizowana została w postaci szablonu, który wymaga od użytkownika implementacji kilku funkcji. Funkcje te mają następujące deklaracje:

- `void __attribute__((weak)) Disable50usTimer(void)` – funkcja służąca do wyłączenia działania timera odliczającego kwanty 50µs,
- `void __attribute__((weak)) Enable50usTimer(void)` – funkcja służąca do włączenia działania timera odliczającego kwanty 50µs,
- `uint8_t __attribute__((weak)) Communication_Get(void)` – funkcja służąca do odczytania pojedynczego znaku,
- `void __attribute__((weak)) Communication_Mode(bool rx, bool tx)` – funkcja służąca do przełączania modułu wykorzystanego do komunikacji w tryb oczekiwania na znak (tj. tryb nasłuchiwania), wysyłania danych (tj. tryb transmisji) lub oczekiwania (wyłączenia) – jednoczesne włączenie transmisji i nasłuchiwania nie jest wykorzystane,
- `void __attribute__((weak)) Communication_Put(uint8_t c)` – funkcja służąca do wysłania pojedynczego znaku.

Nadanie funkcji atrybutu `weak` pozwala na zdefiniowanie funkcji, której domyślna postać jest pusta, natomiast użytkownik może ją „nadpisać". Jest to podobnie działający mechanizm jak przeciążanie znane z języka C++.

Dodatkowo użytkownik jest zobowiązany do wywołania kilku funkcji mających na celu sygnalizację pewnych zdarzeń lub odświeżenie wartości związanych z implementacją protokołu MODBUS RTU. Przykładowa implementacja podanych funkcji i wywołanie prezentowane są poniżej. Przyjęte zostały pewne założenia:

- zegar SysTick zgłasza przerwanie co 10µs,
- za odliczanie kwantów 50µs odpowiedzialny jest timer TIM4,
- do komunikacji zostanie wykorzystany USART1.

Konfiguracja zegara SysTick była już prezentowana – zostanie tutaj pominięta. Należy jednak zwrócić uwagę na fakt, iż do tej pory w obsłudze przerwania wynikającej z działania SysTick'a znajdowało się wywołanie funkcji `DelayTick()`, które powodowało dekrementację licznika milisekund. Ponieważ częstotliwość pracy zegara SysTick zmieniła się – należy wprowadzić odpowiednie zmiany także i w wywołaniu/definicji funkcji związanych z opóźnieniem.

Zgodnie z wcześniejszym opisem, w funkcji obsługi przerwania SysTick musi znaleźć się wywołanie funkcji `MB()`. Dodatkowo jednak umieszczone tutaj zostanie wywołanie funkcji `TimeoutTick()`, której zadaniem jest dekrementacja licznika tak, jak ma to miejsce przy funkcji opóźnienia. Pozwala ona za to na realizację nie opóźnienia, a odmierzenia pewnej ilości czasu jednocześnie nie blokując działania programu. Możnaby to oczywiście zrealizować z użyciem timer'a, lecz ponieważ dokładność pomiaru tutaj nie odgrywa kluczowej roli można zastosować właśnie tak prosty mechanizm. Na koniec warto pamiętać o nadaniu przerwaniu zgłaszanemu przez SysTick jeden z wyższych priorytetów, aby nie doprowadzić do zakleszczenia programu.

Ponieważ wykorzystany zostanie USART1 do komunikacji, potrzeba go skonfigurować. Podstawową konfiguracją protokołu MODBUS RTU jest 8E1, z prędkością 19200 baudrate – wykorzystana zostanie jednak prędkość 115200. Dodatkowo USART1 będzie pracował z użyciem przerwań `TXE`, `RXNE`, których funkcja obsługi zdefiniowana jest jako:

```
void USART1_IRQHandler(void){
    if( USART_GetITStatus(USART1, USART_IT_RXNE) ){
        USART_ClearITPendingBit(USART1, USART_IT_RXNE);
        SetCharacterReceived(true);
    }
    if( USART_GetITStatus(USART1, USART_IT_TXE) ){
        USART_ClearITPendingBit(USART1, USART_IT_TXE);
        SetCharacterReadyToTransmit();
    }
}
```

Widoczne tutaj wywołania funkcji `SetCharacterReceived(true)` oraz `SetCharacterReadyToTransmit()` mają na celu poinformowanie funkcji obsługującej protokół MODBUS o odpowiednio otrzymaniu znaku i osiągnięciu gotowości do transmisji znaku. Faktyczne wysłanie oraz odebranie znaku realizowane jest w osobnych funkcjach, które wywoływane są z poziomu funkcji `MB()`. Ich definicje wymagają zaimplementowania przez użytkownika – przykładowo w następujący sposób:

```
void Communication_Put(uint8_t ch){
    USART_SendData(USART1, ch);
}

uint8_t Communication_Get(void){
    uint8_t tmp = USART_ReceiveData(USART1);
    SetCharacterReceived(false);
    return tmp;
}
```

gdzie `SetCharacterRecieved` jest funkcją ustawiającą flagę świadczącą o gotowości do odczytu kolejnego znaku – po odbiorze tę gotowość należy oczywiście odwołać, co jest zrealizowane wyżej.

Funkcja służąca do przełączania modułu komunikacyjnego w tryb nasłuchiwania i transmisji jest również wyjątkowo prosta i może zostać zrealizowana jako funkcja włączająca i wyłączająca stosowne przerwania w module USART1:

```
void Communication_Mode(bool rx, bool tx){
    USART_ITConfig(USART1, USART_IT_RXNE, rx?ENABLE:DISABLE);
    USART_ITConfig(USART1, USART_IT_TXE , tx?ENABLE:DISABLE);
```

```
}
```

Kolejnymi funkcjami wymagającymi implementacji są:

```
void Enable50usTimer(void){
    TIM_ITConfig(TIM4, TIM_IT_Update, ENABLE);
}

void Disable50usTimer(void){
    TIM_ITConfig(TIM4, TIM_IT_Update, DISABLE);
}
```

które są wykorzystane do zatrzymywania i uruchamiania timera odmierzającego kwanty 50µs. Aby naliczanie tych kwantów miało faktycznie miejsce należy dodać do obsługi przerwania generowanego przez TIM4 wywołanie funkcji `Timer50usTick`, która nie przyjmuje argumentów. Konfiguracja samego timera nie będzie przytaczana gdyż była ona omawiana w poprzednim ćwiczeniu.

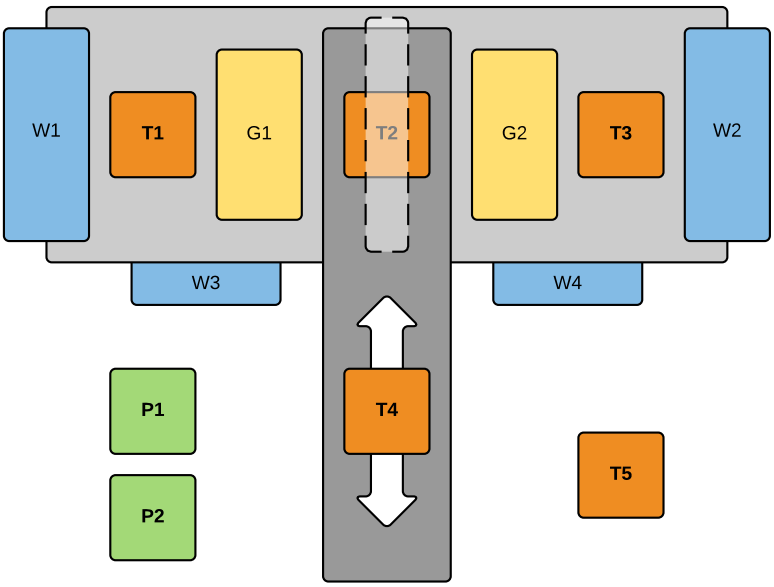
Procedura inicjalizacji komunikacji z użyciem protokołu MODBUS składa się z następujacych etapów:

1. Konfigruacja USART – pinów PA9 i PA10, modułu USART1, przerwań,
2. Konfiguracja timera – modułu TIM4, przerwań,
3. Konfiguracja protokołu MODBUS – tj. wywołanie `MB_Config`, w argumentcie podając prędkość komunikacji (*baudrate*),
4. Wyłączenie przerwań USART'a – tj. stosowne wywołanie `Communication_Mode`,
5. Konfiguracja SysTick'a (wyzwalając od tej pory regularnie `DelayTick`, `TimeoutTick` i `MB`),-
6. Nadanie SysTick'owi wysokiego priorytetu.

Ponieważ omawiane niżej stanowisko, będące adresatem wszelkich wiadomości wysyłanych z mikrokontrolera STM32, posiada interfejs komunikacyjny RS-485, a nie TTL czy RS-232 jak na rozważanej płycie rozwojowej – wykorzystany zostanie stosowny konwerter. Jego schemat jest nieistotny z punktu widzenia realizacji ćwiczenia, tak więc zostanie on pominięty.

Stanowisko grzejąco-chłodzące. W tym ćwiczeniu (oraz nastepnych) wykorzystane zostanie stanowisko grzejąco-chłodzące zrealizowane w Instytucie Automatyki i Informatyki Stosowanej Politechniki Warszawskiej. Pełna dokumentacja stanowiska znajduje się w podanym przez prowadzącego katalogu, natomiast poniżej przedstawione zostaną wyłącznie istotne, z punktu widzenia tego ćwiczenia, cechy.

Stanowisko składa się z 6 elementów wykonawczych: 4 wentylatorów (W1-W4) i 2 grzałek (G1, G2) oraz 7 czujników: 5 czujników temperatury (T1-T5), pomiar prądu (P1) i pomiar napięcia (P2). Rozmieszczenie elementów widoczne jest na Rys. 33. Czujnik T5 służy do pomiaru temperatury otoczenia – aby spełniał swoją rolę właściwie, nie powinien się znajdować w okolicy strumienia powietrza wytwarzanego przez wentylatory ani nie powinien znajdować się w pobliżu nagrzewających się elementów. W ramach tego ćwiczenia skupienie pada na elementy W1, T1, G1.



Rysunek 33: Schemat rozmieszczenia elementów wykonawczych i pomiarowych w stanowisku grzejąco-chłodzącym

Komunikacja ze stanowiskiem może odbywać się na trzy sposoby: przy użyciu opracowanego dla niego protokołu komunikacyjnego (z pośrednictwem przejściówki USB-UART), przy użyciu standardu napięciowego 0-10V lub korzystając z protokołu MODBUS RTU i standardu RS-485. Ta ostatnia opcja będzie głównie wykorzystana w tym i następnych ćwiczeniach. Konfiguracja protokołu MODBUS RTU dla stanowiska zakłada występowanie bitu parzystości (sprawdzanie czy liczba jedynek jest parzysta), danych o długości 8 bitów i 1 bitu stopu. Prędkość transmisji jest konfigurowalna, lecz na rzecz tego ćwiczenia wykorzystana będzie prędkość 115200. Adres stanowiska jest również konfigurowalny – każde stanowisko przygotowane zostało tak, aby mieć unikalny adres.

Stanowisko to odświeża wartości pomiarów i sterowania z częstotliwością 1Hz. Oznacza to, że zmiana sygnału sterującego może zostać zauważona maksymalnie po 1 sekundzie. Zmiana sterowania (tj. mocy z jaką pracują grzałki lub wentylatory) następuje poprzez zapisanie nowej wartości do rejestru typu *Holding Register*, o adresie od 0 do 6, natomiast odczyt pomiaru dokonywany jest poprzez odczyt zawartości rejestru typu *Input Register* o adresie od 0 do 7. Wspomniane elementy, które są interesujące z punktu widzenia tego ćwiczenia mają adresy: W1 – 0, T1 – 0, G1 – 4. Wartości sterowania wyrażane są w promilach pełnej mocy elementu wykonawczego. Oznacza to, że zapisanie wartości 200 do odpowiedniego rejestru typu *Holding Register* spowoduje, że powiązany z tym rejestrem element będzie pracował z mocą równą 20.0% pełnej mocy tego elementu. Podobnie należy traktować pomiary temperatury – wartość 2500 znajdująca się w rejestrze typu *Input Register* oznacza, że związany z nim czujnik temperatury zmierzył 25.00°C.

Aby zapewnić poprawność komunikacji ze stanowiskiem należy uważnie śledzić odpowiedzi na wysłane wiadomości. Brak odpowiedzi może sugerować problemy z komunikacją (niepoprawny adres, prędkość).

Odpowiedzi zawierające kody błędu należy przeanalizować w oparciu o dokumentację protokołu MODBUS RTU.

3.1 Przebieg laboratorium

Celem ćwiczenia jest wykorzystanie komunikacji przy użyciu protokołu MODBUS RTU do sterowania mikrokontrolerem oraz pętli prądowej 4-20 mA do odczytu wartości temperatury. Aby to osiągnąć należy:

- Skonfigurować piny PA9 oraz PA10 pod kątem komunikacji USART, odpowiednio jako pin nadawczy i odbiorczy. Prędkość komunikacji musi zgadzać się z ustawieniami w stanowisku, tj. baudrate 115200, 8 bitów na dane oraz bit parzystości.
- Skonfigurować timer tak, aby odmierzał 50µs kwanty, inkrementując w ten sposób licznik.
- Korzystając z dostarczonych funkcji należy zrealizować prosty regulator, który będzie regulował temperaturę T1 stanowiska grzejąco-chłodzącego zgodnie z następującymi regułami:
  - temperatura zbyt wysoka – włącza się wentylator W1,
  - temperatura za niska – włącza się grzałka G1,
  - temperatura w okolicach (należy je rozsądnie zdefiniować) temperatury zadanej – oba elementy są wyłączone.
- Obecną oraz zadaną temperaturę należy wyświetlić na wyświetlaczu LCD,
- Należy regularnie wykonywać pomiary temperatury z wykorzystaniem pętli pradowej 4-20mA i wynik pomiaru przedstawić na wyświetlaczu w °C (jako symbol ° można wykorzystać ∗).

Diody można skonfigurować dowolnie – w szczególności mogą służyć jako doskonałe narzędzie do zgrubnego badania stanu wykonania programu lub nawet jako prosty mechanizm debugowania.