JĘZYK C W PROGRAMOWANIU URZĄDZEŃ

PRAKTYCZNA NAUKA TWORZENIA KODU DLA SYSTEMÓW WBUDOWANYCH

STEPHEN OUALLINE

no starch press

Helion

Tytuł oryginału: Bare Metal C: Embedded Programming for the Real World

Tłumaczenie: Krzysztof Bąbol (wprowadzenie, rozdz. 1 – 14),
Andrzej Watrak (rozdz. 15 – 17, posłowie, dodatek).

ISBN: 978-83-8322-085-7

Drogi Czytelniku!
Jeżeli chcesz ocenić tę książkę, zajrzyj pod adres
https://helion.pl/user/opinie/japrsy
Możesz tam wpisać swoje uwagi, spostrzeżenia, recenzję.

Pliki z przykładami omawianymi w książce można znaleźć pod adresem:
https://ftp.helion.pl/przyklady/japrsy.zip

Helion S.A.
ul. Kościuszki 1c, 44-100 Gliwice
tel. 32 230 98 63
e-mail: *helion@helion.pl*
WWW: *https://helion.pl* (księgarnia internetowa, katalog książek)

Printed in Poland.

Spis treści

9

Wyjście szeregowe w mikrokontrolerze STM

Powracamy do programu „Witaj, świecie”. Tym razem jednak użyjemy płytki Nucleo, co stawia przed nami kilka wyzwań. Po pierwsze, gdzie należy wypisać komunikat, skoro nie ma wyświetlacza? Na szczęście mikroukład ma port szeregowy, ładnie połączony z portem USB/szeregowym w górnej części płytki.

Następnym wyzwaniem jest samo pisanie kodu. Musimy zainicjować urządzenie i utworzyć procedurę faktycznie wypisującą znak. Urządzenie zaprojektowano tak, że przyjmuje naraz tylko jeden znak, i musimy pamiętać o tym ograniczeniu podczas pisania programu.

Zanim zaczniemy pracować na urządzeniu, zasymulujemy ten proces. Język C ma wiele funkcji standardowych, takich jak `puts`, ułatwiających wyprowadzanie danych. Płytka Nucleo nie ma takich przyjemnych funkcji, więc musimy napisać je sami. Aby przejść do programowania niskopoziomowego, które w przypadku płytki Nucleo jest niezbędne, wypiszemy najpierw łańcuch „Hello World” znak po znaku.

Wypisywanie znaków łańcucha jeden po drugim

Gdy w programie w języku C wywoływana jest standardowa funkcja `puts`, zaczyna się długi proces programowy, który obejmuje wywołania kodu jądra, wewnętrzne buforowanie, planowanie przerwań i odwołania do sterowników urządzeń (przeczytasz o nich więcej w następnym rozdziale). W końcu dochodzimy do miejsca, w którym znaki, jeden po drugim, są wysyłane do urządzenia. Aby to zasymulować, będziemy wysyłać do systemu operacyjnego znak po znaku. Innymi słowy, ograniczymy się do wypisywania danych wyjściowych tylko przy użyciu standardowej funkcji `putchar`.

Listing 9.1 zawiera program wypisujący tym trudnym sposobem łańcuch znaków `"Hello World\n"`. Podkreślam, utrudniamy sobie sprawę dlatego, że później, na płytce Nucleo, będziemy musieli to zrobić w *naprawdę* trudny sposób.

Listing 9.1. Wypisywanie łańcucha po jednym znaku

putchar.c

```
/*
 * Wypisuje łańcuch znak po znaku.
 */
#include <stdio.h>

char hello[] = "Hello World\n"; // Znaki do wyświetlenia.
int curChar;     // Numer wypisywanego znaku.

int main()
{
    for (curChar = 0; hello[curChar] != '\0'; ++curChar) ❶
        putchar(hello[curChar]);
    return (0);
}
```

Jedynym interesującym miejscem tego programu jest pętla `for` ❶, która nie zatrzymuje się po określonej liczbie znaków, ale wtedy, gdy program napotka symbol końca łańcucha (ang. *end-of-string*, `'\0'`). Dzięki temu program może wyprowadzić na wyjście łańcuch o dowolnej długości.

Definiowanie własnej funkcji putchar

Aby ulepszyć ten program, najpierw uczyńmy `curChar` zmienną lokalną. Potem zdefiniujemy funkcję o nazwie `myPutchar`, która wysyła na wyjście standardowe jeden znak (listing 9.2).

Listing 9.2. Znak po znaku przy użyciu własnej funkcji wyprowadzającej

my_putchar.c

```
/**
 * Wypisz łańcuch znak po znaku
 * za pomocą naszej własnej funkcji.
```

```
 */
#include <stdio.h>

char hello[] = "Hello World\n"; // Znaki do wyświetlenia.

/** ❶
 * Ponowna implementacja funkcji putchar.
 *
 * @param ch Znak do wysłania.
 *
 * @note Nie tak całkiem bezużyteczna, jak by się mogło wydawać.
 */
void myPutchar(const char ch) ❷
{
    putchar(ch); ❸
}

int main()
{
    int curChar;         // Indeks aktualnie wypisywanego
                         // znaku.
    for (curChar = 0; hello[curChar] != '\0'; ++curChar)
      myPutchar(hello[curChar]); ❹
    return (0);
}
```

Na początku funkcji `myPutchar` dodaliśmy w bloku komentarza kilka dodatkowych elementów ❶. Słowo kluczowe `@param` wskazuje parametr, a po słowie `@note` widoczna jest uwaga. W komentarzach w stylu Doxygen można stosować wiele innych słów kluczowych, ale na razie użyjemy tylko tych podstawowych, by zachować zgodność z istniejącym kodem kontrolera STM.

Sama funkcja zaczyna się od deklaracji `void myPutchar(const char ch)` ❷, wskazującej, że procedura `myPutchar` nic nie zwraca, a przyjmuje jeden parametr typu `char`. Modyfikator `const` wskazuje zaś, że wewnątrz procedury nie będziemy zmieniać parametru. (W istocie nie możemy go zmienić, bo gdybyśmy spróbowali, kompilator wygenerowałby błąd).

Podczas wywoływania procedury ❹ program wykonuje następujące kroki:

1. Oblicza wartość wyrażenia `hello[curChar]`.
2. Umieszcza tę wartość w miejscu, gdzie funkcja `myPutchar` może ją znaleźć.
3. Rejestruje adres następnej instrukcji (koniec pętli `for`).
4. Zaczyna wykonywać funkcję `myPutchar`. (Zmienna `ch` została zainicjowana w kroku nr 2).

Podobna seria kroków jest wykonywana, gdy wywołujemy funkcję `putchar` ❸. Jedyna różnica polega na tym, że procedurę `myPutchar` musieliśmy napisać sami, a funkcję `putchar` dostarczyli ludzie, którzy napisali bibliotekę standardową języka C.

Tworzenie funkcji (`myPutchar`), która nic nie robi, tylko wywołuje drugą (`putchar`), nie jest zbyt przydatne. Płytka Nucleo nie ma funkcji `putchar`, więc w dalszej części rozdziału napiszemy ją sami. Zanim jednak to zrobimy, przyjrzyjmy się szczegółom dotyczącym urządzenia szeregowego.

Wyjście szeregowe

Wyjście szeregowe jest jednym z najprostszych sposobów wyprowadzania danych z systemu wbudowanego. Interfejs elektryczny składa się z linii wysyłania (ang. *send line*, TX), odbioru (ang. *receive line*, RX) i masy (ang. *ground*, GND). W większości systemów wbudowanych są one ukryte, dostępne tylko dla programistów chętnych do otworzenia obudowy i połączenia się z portem szeregowym.

Nasz mikroukład zawiera urządzenie szeregowe, na którym możemy zapisywać dane. Wystarczy połączyć linie TX, RX i GND między mikrokontrolerem (dolną połową płytki rozwojowej) a urządzeniem USB/szeregowym na górnej połowie płytki.

W poniższej tabeli przedstawione są konieczne połączenia:

Mikrokontroler		**Moduł USB/szeregowy i inne urządzenia wspomagające**	
RX	CN9-1	TX	CN3-1
TX	CN9-2	RX	CN3-2
GND	CN6-5	GND	CN4-3

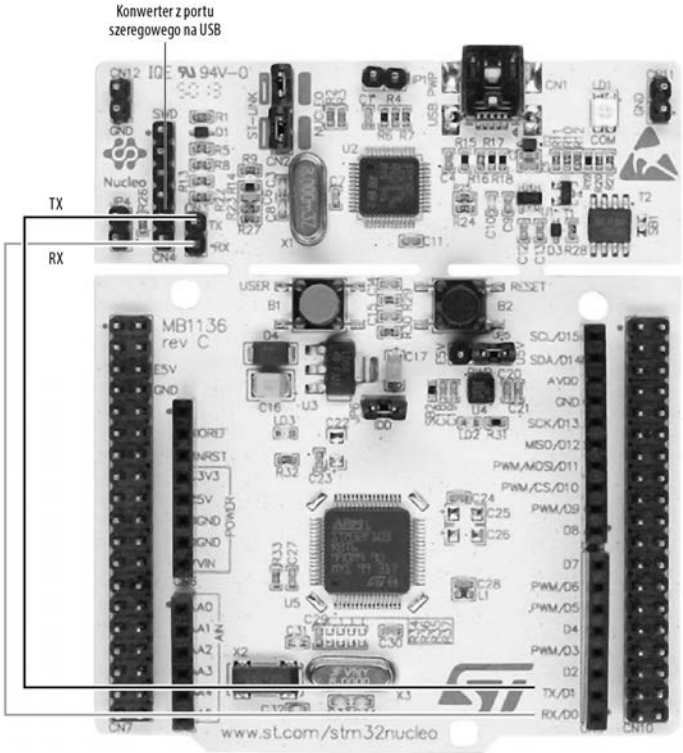
Gdybyśmy mieli Raspberry Pi albo inny system wbudowany bez zintegrowanego kontrolera szeregowego, sami musielibyśmy poprowadzić między nimi przewody. Na rysunku 9.1 przedstawiono rozkład tych komponentów i ich wewnętrzne połączenia wykonane przez STM.

Firma STM zrobiła już wszystko za nas. Nie potrzeba żadnych zworek.

Krótka historia komunikacji szeregowej

Komunikacja szeregowa sięga dawnych czasów, jeszcze epoki *przedkomputerowej*. Rolę internetu odgrywał swego czasu telegraf, umożliwiający długodystansową transmisję komunikatów za pomocą kabli. Moduł nadawczy składał się z klucza telegraficznego, który po naciśnięciu powodował „kliknięcie" u odbiorcy. Kliknięcia były kodowane przy użyciu systemu zwanego *kodem Morse'a* (który stosowany jest do dziś). Wynalazek ten zrewolucjonizował komunikację. Można było wysłać wiadomość do sąsiedniego miasta i tego samego dnia otrzymać odpowiedź.

Istniał tylko jeden problem; na obu końcach telegrafu trzeba było mieć wykwalifikowanych operatorów znających kod Morse'a. Laicy nie potrafili wysyłać ani odbierać komunikatów, a szkolenie operatorów było kosztowne. Jednym z rozwiązań było użycie dwóch zegarów: jednego u nadawcy, a drugiego u odbiorcy. Na tarczy zegara znajdowały się litery od „A" do „Z". Aby wysłać na przykład „S", nadawca



Rysunek 9.1. Komunikacja szeregowa na płytce Nucleo

musiał poczekać do chwili, aż wskazówka zegara pokazała „S", i przycisnąć klucz telegrafu. Odbiorca widział wtedy, że wskazówka znajduje się na „S", i zapisywał tę literę.

Utrzymanie synchronizacji zegarów było jednak prawie niemożliwe, więc bardzo sprytny wynalazca zdecydował, że wskazówka każdego zegara musi się zatrzymywać w górnej pozycji. Gdy nadawca zamierzał wysłać literę, naciskał klawisz telegrafu, nadając **sygnał startu** (ang. *start signal*). Zegary miały wtedy wystarczająco dużo czasu, by ich wskazówki mogły należycie wykonać jeden obrót wokół tarcz. Wtedy nadawca wysyłał sygnał litery. Gdy wskazówka wracała do góry, krótka przerwa zwana **czasem zatrzymania** (ang. *stop time*) dawała wolniejszemu zegarowi szansę na to, by dogonić szybszy. Sekwencja zdarzeń wyglądała następująco: sygnał startu, sygnał litery, czas zatrzymania.

Przenieśmy się teraz do chwili wynalezienia dalekopisu, pozwalającego wysyłać teksty poprzez odpowiednik linii telegraficznych. Dalekopis nie nadawał jednego impulsu dla każdej litery, ale kodował znaki jako serię ośmiu impulsów (siedmiu na dane i jednego do prostej kontroli błędów). Stosowano koder klawiaturowy zbudowany z dźwigni przekształcających naciśnięcie klawisza na 8-bitowy kod, który trafiał do mechanicznego rejestru przesuwającego (ang. *shift register*) przypominającego wyglądem kopułkę rozdzielacza zapłonu (ang. *distributor cap*).

Urządzenie to wysyłało impulsy przewodem, a inny dalekopis zamieniał je na pojedynczą drukowaną literę.

W przypadku dalekopisu sekwencja zdarzeń wyglądała następująco: nadawca naciskał klawisz, a mechanizm nadawczy wysyłał 10-bitowy sygnał (1 bit startu, 8 bitów danych i 1 bit stopu). Gdy odbiornik otrzymywał bit startu, włączał rejestr przesuwający (kolejny silnik z kopułką) i na podstawie nadchodzących impulsów obracał głowicę drukującą tak, by została wydrukowana właściwa litera. Po wysłaniu 8 bitów danych obie maszyny odczekiwały przynajmniej czas 1 bitu (bitu stopu), aby utrzymać synchronizację.

Większość dalekopisów potrafiła przesyłać znaki z prędkością 110 bodów (bitów na sekundę), czyli 10 znaków na sekundę. W czasach megabitowych połączeń internetowych nie wygląda to imponująco, ale mimo wszystko doszło do rewolucyjnego usprawnienia komunikacji.

W dzisiejszych komputerach nadal stosuje się komunikację szeregową, taką samą jak w dalekopisach. Zwiększyła się prędkość, ale protokół zasadniczo nie uległ zmianie.

Zakończenia wierszy

W rzeczy samej wciąż mamy do czynienia z inną zaszłością po dalekopisach: zakończeniami wierszy. Po wpisaniu 80 znaków można było wysłać do urządzenia znak **powrotu karetki** (ang. *carriage return*), aby wróciło ono na pozycję 1. Problem leżał w tym, że przesunięcie głowicy drukującej zajmowało 0,2 sekundy. Jeśli natychmiast po rozkazie powrotu karetki został wysłany kolejny znak, pośrodku wiersza pojawiał się rozmyty kleks, ponieważ głowica drukująca próbowała drukować w trakcie przesuwania.

Konstruktorzy dalekopisów rozwiązali ten problem przez wprowadzenie dwuznakowego końca wiersza. Pierwszy znak, powrotu karetki, przesuwał głowicę drukującą na pozycję 1. Drugi, wysuw wiersza (ang. *line feed*), przesuwał papier o jeden wiersz do góry. Ponieważ w czasie wysuwu wiersza na papierze nic się nie drukowało, fakt, że odbywało się to w czasie, gdy głowica biegła w lewo, nie miał znaczenia.

Gdy jednak pojawiły się komputery, przechowywanie danych pochłaniało mnóstwo pieniędzy (setki dolarów *za bajt*), więc oznaczanie końca wiersza dwoma znakami stało się kosztowne. Twórcy systemu Unix, który był inspiracją dla Linuksa, postanowili używać wyłącznie znaku wysuwu wiersza (`\n`). W firmie Apple zdecydowano się korzystać tylko ze znaku powrotu karetki (`\r`), a w koncernie Microsoft — z obu (`\r\n`).

Język C automatycznie obsługuje różne typy zakończeń wierszy w bibliotece systemowej, ale tylko wtedy, gdy się z niej korzysta. Jeśli programista zajmuje się tym samodzielnie, tak jak my będziemy to robić, musi wypisać pełną sekwencję końca wiersza (`\r\n`).

Komunikacja szeregowa dzisiaj

Dzisiaj prawie każdy procesor wbudowany ma w sobie interfejs szeregowy. Urządzenia szeregowe są proste i tanie w produkcji. Jedyną różnicą między interfejsem dzisiejszym a XIX-wiecznym jest to, że wzrosła szybkość (ze 110 do 115 200 bitów na sekundę) oraz że zmieniły się napięcia. W XIX w. stosowano od –15 do –3 woltów dla bitu „zero" i od +3 do +15 woltów dla bitu „jeden". Wciąż stanowi to „standard", jednak większość komputerów korzysta z napięcia wynoszącego 0 woltów (oznaczającego zero) i 3 wolty (oznaczającego jeden).

Urządzenie obsługujące szeregowe operacje we/wy nosi nazwę **uniwersalnego asynchronicznego nadajnika-odbiornika** (ang. *universal asynchronous receiver-transmitter*, UART). Istnieją dwa główne typy komunikacji szeregowej: *asynchroniczna* i *synchroniczna*. Podczas komunikacji synchronicznej zegary nadajnika i odbiornika muszą być zsynchronizowane. W tym celu nadajnik ciągle wysyła znaki. Odbiornik analizuje je i na ich podstawie ustala chronometraż zegara. Nadajnik musi ciągle wysyłać znaki, nawet jeśli są to znaki „nieaktywności" (ang. *idle characters*, oznaczające brak danych). W komunikacji asynchronicznej nie ma wspólnego zegara. Odebranie bit startu powoduje, że odbiornik uruchamia swój zegar i oczekuje kolejnego znaku. W komunikacji asynchronicznej zakłada się, że na czas przesłania jednego znaku nadajnik i odbiornik potrafią utrzymać zbliżone taktowanie swoich zegarów. Ponieważ do ich synchronizowania nie jest potrzebna ciągła transmisja, nie występuje znak nieaktywności. W stanie bezczynności nadajnik po prostu nic nie wysyła.

Mikroukład STM ma jeden port umożliwiający zarówno komunikację synchroniczną, jak i asynchroniczną, więc w dokumentacji mikrokontrolerów STM określany jest on jako **uniwersalny synchroniczny/asynchroniczny nadajnik-odbiornik** (ang. *universal synchronous/asynchronous receiver-transmitter*, USART). W tym programie użyjemy terminu *UART* w celu zachowania zgodności z biblioteką STM HAL.

Szeregowe „Witaj, świecie!"

Utwórzmy nowy projekt z plikiem *main.c*, w którym znajdzie się program „Witaj, świecie". Będzie on dość długi, ale trzeba w nim zrobić wszystko to, co ukrywa przed nami system operacyjny. Najpierw dołączymy pliki nagłówkowe ze zdefiniowaną informacją na temat UART (a także wielu innych urządzeń):

```
#include "stm32f0xx.h"
#include "stm32f0xx_nucleo.h"
```

Co do kodu, zaczniemy od funkcji `main`:

```
int main(void)
{
    HAL_Init(); // Zainicjuj sprzęt. ❶
    led2_Init();
```

```
    uart2_Init();

    // Kontynuuj wysyłanie komunikatu przez długi czas.
    for (;;) { ❷
        // Wysyłaj znak po znaku.
        for(current = 0; hello[current] != '\0'; ++current) {
            myPutchar(hello[current]); ❸
        }
        HAL_Delay(500); ❹
    }
}
```

Funkcja `main` wygląda całkiem podobnie do tej z listingów 9.1 i 9.2. Jedynym dodatkiem jest to, że musimy zainicjować wszystkie urządzenia, których zamierzamy użyć ❶, w tym bibliotekę sprzętową (`HAL_Init`), czerwoną diodę LED (`led2_Init`) oraz interfejs UART (`uart2_Init`). Nasz program wbudowany nie może się zatrzymać, więc mamy nieskończoną pętlę ❷, w której wysyłany jest łańcuch znaków ❸, a potem następuje 0,5 sekundy przerwy ❹.

Jedną z pierwszych rzeczy do zrobienia w następnej kolejności jest utworzenie funkcji `ErrorHandler`, którą biblioteka HAL wywołuje wtedy, gdy coś pójdzie nie tak. Nie możemy wypisać komunikatu o błędzie, ponieważ nasz kod wypisywania właśnie się popsuł, więc uciekniemy się do migania czerwonym światełkiem. Co prawda stanowi to bardzo skąpy wskaźnik błędu, ale tak samo jest z kontrolką silnika (ang. *check engine*) w samochodzie. W obu przypadkach projektanci zrobili wszystko, co mogli. Nie będziemy tu analizować funkcji `Error_Handler`; jest to „miganie diodą" z rozdziału 3. pod nową nazwą.

Inicjacja interfejsu UART

Programowanie urządzenia szeregowego powinno być proste, jednak inżynierowie firmy STMicroelectronics zdecydowali się ulepszyć prosty interfejs UART za pomocą dodatkowych funkcji. W rezultacie opis prostego urządzenia szeregowego zajmuje teraz w podręczniku 45 stron. A my po prostu chcemy je wykorzystać do wysyłania znaków. Nie musi ono ich nawet odbierać.

Na szczęście biblioteka HAL zawiera funkcję `HAL_UART_Init`, która ukrywa wiele kłopotliwych detali. Nie da się niestety ukryć szczegółów związanych z jej wywoływaniem, ale nie można mieć wszystkiego. W funkcji `uart2_Init` musimy skonfigurować strukturę inicjacyjną, a potem wywołać funkcję `HAL_UART_Init`:

```
void uart2_Init(void)
{
  // Inicjacja interfejsu UART.
  // UART2 -- ten połączony z interfejsem USB programatora ST-LINK.
  uartHandle.Instance = USART2; ❶
  uartHandle.Init.BaudRate = 9600;                  // Prędkość 9600 bodów. ❷
  uartHandle.Init.WordLength = UART_WORDLENGTH_8B;  // 8 bitów na znak. ❸
  uartHandle.Init.StopBits = UART_STOPBITS_1;       // 1 bit stopu. ❹
```

```
  uartHandle.Init.Parity = UART_PARITY_NONE;          // Brak kontroli parzystości. ❺
  uartHandle.Init.Mode = UART_MODE_TX_RX;             // Wysyłanie i odbiór. ❻
  uartHandle.Init.HwFlowCtl = UART_HWCONTROL_NONE;    // Brak sprzętowego sterowania
                                                      // przepływem. ❼

  // Poddaj strumień przychodzący nadpróbkowaniu.
    uartHandle.Init.OverSampling = UART_OVERSAMPLING_16;

  // Nie stosuj jednej próbki na bit.
  uartHandle.Init.OneBitSampling = UART_ONE_BIT_SAMPLE_DISABLE; ❽

  // Żadnych zaawansowanych funkcji.
  uartHandle.AdvancedInit.AdvFeatureInit = UART_ADVFEATURE_NO_INIT; ❾
  /*
   * Dla tych z Was, którzy łączą się przez emulator terminala: powyższe parametry
   * przekładają się na format transmisji 9600,8,N,1.
   */

  if (HAL_UART_Init(&uartHandle) != HAL_OK)
  {
      Error_Handler();
  }
}
```

Najpierw informujemy system, który interfejs UART ma zostać użyty ❶. Nasz mikroukład ma ich kilka, a z interfejsem szeregowym/USB połączony jest drugi z nich. Następnie ustawiamy prędkość na 9600 bodów (bitów na sekundę) ❷, czyli 960 znaków na sekundę. Dlaczego stosunek ten wynosi 10 do 1? Mamy 1 bit startu, 8 bitów danych i 1 bit stopu. Liczba bitów na znak wynosi 8, bo w języku C znaki przechowywane są w jednostkach 8-bitowych. Możliwe jest istnienie systemu z 5, 6, 7 albo 9 bitami na znak, ale prawie wszędzie używa się 8, poza urządzeniami telekomunikacyjnymi dla niesłyszących (TDD), w których stosuje się 5 bitów. Musimy powiedzieć systemowi, że użyjemy 8 bitów ❸.

W następnym wierszu konfigurowana jest liczba bitów stopu ❹, czyli upływ czasu (w bitach) pomiędzy kolejnymi znakami. Większość ludzi stosuje 1 bit stopu. (Jeśli w nadajniku używane są 2, a w odbiorniku 1, wciąż będzie to działać. Dodatkowy bit zostanie zinterpretowany jako czas nieaktywności między znakami).

Początkowo w urządzeniach szeregowych stosowano 7-bitowe znaki i 1 bit parzystości. Ten ostatni stanowił prostą, prymitywną metodę kontroli błędów. Nie korzystamy z tej funkcji, więc wyłączamy bit parzystości ❺. Następnie włączamy nadajnik i odbiornik ❻.

Oryginalny interfejs szeregowy (według standardu RS-232) ma kilka linii sprzętowego sterowania przepływem danych. Na płytce nie są one połączone ścieżkami i nie będziemy z nich korzystać ❼.

Jedną z zaawansowanych funkcji tego interfejsu UART jest **nadpróbkowanie** (ang. *oversampling*). Pozwala ono odbiornikowi wielokrotnie sprawdzić stan bitu przychodzącego, zanim zadecyduje on, czy jest to jedynka, czy zero. Funkcja ta przydaje się czasami w środowisku o dużych szumach elektrycznych albo wtedy,

gdy kable szeregowe biegną na dużych odległościach. Nasz „kabel" szeregowy składa się z dwóch ścieżek biegnących od dołu do góry płytki, o długości około 7,5 centymetra. Nie potrzebujemy nadpróbkowania, ale musimy je wyłączyć ❽[1]. I wreszcie: nie korzystamy z żadnych zaawansowanych funkcji.

Potem, w celu zainicjowania interfejsu UART, wywołujemy funkcję `HAL_UART_Init` ❾, która wymaga pomocy, by mogła wykonać swoje zadanie. Należące do procesora piny **wejścia/wyjścia ogólnego przeznaczenia** (ang. *general-purpose input/output*, GPIO) mogą nie tylko działać zgodnie ze swoją nazwą, ale też robić wiele różnych rzeczy. Większość z nich ma „funkcje alternatywne", co oznacza, że można zaprogramować je tak, by działały jako różne urządzenia (pin GPIO, urządzenie USART, magistrala SPI, magistrala I2C, pin do modulacji szerokości impulsów (PWM) i tak dalej). Warto zauważyć, że nie każdy pin wszystko obsługuje. Na koniec funkcja `HAL_UART_Init` w celu skonfigurowania pinów jako interfejsu UART wywołuje funkcję `HAL_UART_MspInit`:

```
HAL_StatusTypeDef HAL_UART_Init(UART_HandleTypeDef *huart)
{
  /* Sprawdź alokację uchwytu do interfejsu UART. */
  if(huart == NULL)
  {
    return HAL_ERROR;
  }
  // ...
  if(huart->gState == HAL_UART_STATE_RESET)
  {
    /* Zaalokuj zasób blokowania i zainicjuj go. */
    huart->Lock = HAL_UNLOCKED;

    /* Zainicjuj niskopoziomowy sprzęt: GPIO, CLOCK. */
    HAL_UART_MspInit(huart);
  }
```

Funkcję `HAL_UART_MspInit` musimy dostarczyć sami. Pamiętaj, że piny są kosztowne, a sterujące nimi tranzystory — tanie. Domyślnie dwa piny sterujące urządzeniem szeregowym, noszące nazwy PA2 i PA3, są pinami GPIO. Trzeba nakazać systemowi, by użył ich funkcji alternatywnej i zmienił je w piny urządzenia szeregowego.

Nasza funkcja `HAL_UART_MspInit` bardzo przypomina kod inicjacji pinu GPIO użyty wcześniej w celu „migania diodą", ale istnieją drobne różnice:

```
void HAL_UART_MspInit(UART_HandleTypeDef* uart)
{
    GPIO_InitTypeDef GPIO_InitStruct;
    if(uart->Instance == USART2) ❶
    {
```

[1] Prawdopodobnie autor się w tym miejscu pomylił. Pole OneBitSampling nie dotyczy nadpróbkowania, ale zmniejszenia z trzech do jednej liczby próbek służących do odczytu bitu — *przyp. tłum.*

```
        /* Włączenie zegara urządzenia peryferyjnego. */
        __HAL_RCC_USART2_CLK_ENABLE(); ❷
        /*
         * Konfigurowanie bitów GPIO interfejsu USART2.
         * PA2    ------> USART2_TX.
         * PA3    ------> USART2_RX.
         */
        GPIO_InitStruct.Pin = GPIO_PIN_2|GPIO_PIN_3;
        GPIO_InitStruct.Mode = GPIO_MODE_AF_PP;
        GPIO_InitStruct.Pull = GPIO_NOPULL;
        GPIO_InitStruct.Speed = GPIO_SPEED_FREQ_LOW;
        // Funkcja alternatywna -- związana z UART.
        GPIO_InitStruct.Alternate = GPIO_AF1_USART2; ❸
        HAL_GPIO_Init(GPIOA, &GPIO_InitStruct);
    }
}
```

Na początku funkcja ta sprawdza, z którego interfejsu USART korzystamy. W jej kodzie konfigurujemy tylko USART2 ❶. Następnie włączamy zegar dla tego urządzenia ❷. Potem konfigurujemy piny GPIO ❸ (robiliśmy to wcześniej w programie migającym diodą). Informujemy w ten sposób mikroukład, że PA2/PA3 nie mają działać jako piny GPIO, ale powinny być połączone z interfejsem USART2.

Przesyłanie znaku

Do przesyłania znaków przez urządzenie szeregowe użyjemy funkcji `myPutchar`. USART to urządzenie we/wy mapowane w pamięci. Aby wysłać znak, musimy go przypisać (zapisać) do tajemniczego miejsca w pamięci (*rejestru*), z którego potem wychodzi on na zewnątrz:

```
uartHandle.Instance->TDR = ch;      // Wyślij znak do interfejsu UART.
```

Musimy też wybrać odpowiedni czas na wysłanie znaku, a to wymaga dodatkowego kodu:

```
void myPutchar(const char ch)
{
    // W wierszu tym pobiera się i zapisuje wartość bitu UART_FLAG_TXE w czasie
    // wywołania. Wartość ta zmienia się, więc jeśli zatrzymasz program na znajdującym się
    // poniżej wierszu z instrukcją "while", wartość będzie równa 0, gdyż znika
    // szybciej, niż można na nią spojrzeć.
    int result __attribute__((unused)) =
        (uartHandle.Instance->ISR & UART_FLAG_TXE);

    // Blokuj wykonanie, do chwili gdy ustawiona zostanie flaga pustego rejestru nadawczego (TXE).
    while ((uartHandle.Instance->ISR & UART_FLAG_TXE) == 0)
```

```
        continue;

    uartHandle.Instance->TDR = ch;      // Wyślij znak do interfejsu UART.
}
```

Rejestr, do którego piszemy, nosi nazwę **TDR** (ang. *transmit data register*, nadawczy rejestr danych). Gdybyśmy dokonywali tam zapisu podczas transmisji znaku, nowy znak nadpisałby stary, co doprowadziłoby do błędów i zamieszania. Aby wysłać znaki `a`, `b`, `c`, moglibyśmy napisać poniższy kod:

```
uartHandle.Instance->TDR = 'a';
sleep_1_960_second();
uartHandle.Instance->TDR = 'b';
sleep_1_960_second();
uartHandle.Instance->TDR = 'c';
sleep_1_960_second();
```

Tego rodzaju odmierzanie czasu jest problematyczne, szczególnie wtedy, gdy chcemy wykonywać w międzyczasie jakiś kod. Mikroukład STM32 ma bity służące do wszystkiego, w tym również ten, który mówi: „Rejestr TDR jest pusty, możesz już zapisać kolejny znak".

Bit ten znajduje się w rejestrze **ISR** (ang. *interrupt and status register*, rejestr przerwań i stanu), zawierającym bity pokazujące stan urządzenia. Na rysunku 9.2 przedstawiono diagram tego rejestru pochodzący z podręcznika mikrokontrolera STM32F030R8 (*RM0360 Reference manual/STM32F030x4/x6/x8/xC and STM32F070x6/xB*).

22.7.7 **Interrupt and status register (I2C_ISR)**
Przesunięcie adresu: 0x18
Wartość resetująca: 0x0000 0001
Dostęp: brak stanów oczekiwania

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	ADDCODE [6:0]							DIR
								r							r

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
BUSY	Res.	ALERT	TIME OUT	PEC ERR	OVR	ARLO	BERR	TCR	TC	STOPF	NACKF	ADDR	RXNE	TXIS	TXE
r		r	r	r	r	r	r	r	r	r	r	r	r	rs	rs

Rysunek 9.2. Zawartość rejestru przerwań i stanu (r — odczyt, rs — odczyt/ustawianie, Res. — zastrzeżony)

Interesuje nas bit o nazwie TXE (bit nr 0 na diagramie). W bibliotece HAL jest on zdefiniowany pod nazwą `UART_FLAG_TXE`. Abyśmy mogli wysłać dane do rejestru TDR bez obawy o nadpisanie przesyłanego znaku, musimy poczekać,

aż bit TXE zostanie wyczyszczony (przyjmie wartość 0). W naszym kodzie pole `uartHandle.Instance->ISR` nie jest nigdzie zmieniane.

Pole `uartHandle.Instance->ISR` to jednak tajemne miejsce w pamięci, które jest elektrycznie połączone z urządzeniem. Stan urządzenia ulega zmianie na przykład po zakończeniu transmisji znaków; zmienia się wówczas także zawartość pola `uartHandle.Instance->ISR`.

UWAGA *Pole `ISR` jest deklarowane z modyfikatorem `volatile`, aby przekazać językowi C, że może ono w każdej chwili ulec zmianie w sposób niepodlegający kontroli kompilatora.*

Jeśli spróbujesz zbadać pole `uartHandle.Instance->ISR` w debuggerze, zobaczysz, że flaga `UART_FLAG_TXE` wydaje się zawsze ustawiona. Dzieje się tak dlatego, że jest ona czyszczona w trakcie przesyłania znaku (na $^1/_{960}$ sekundy), co stanowi długi czas dla komputera, ale bardzo krótki dla człowieka wpisującego polecenia.

Aby lepiej pokazać, co się dzieje, dodaliśmy bezużyteczną instrukcję:

```
int result __attribute__((unused)) =
    (uartHandle.Instance->ISR & UART_FLAG_TXE);
```

Instrukcja ta sprawdza wartość bitu `UART_FLAG_TXE` i zapisuje ją w zmiennej `result`. Teraz wartość wyrażenia `(uartHandle.Instance->ISR & UART_FLAG_TXE)` może nagle zniknąć, ale wartość zmiennej `result` nie zmieni się aż do końca procedury.

Można przyjrzeć się tej zmiennej w debuggerze i zobaczyć, jaką wartość miał wspomniany bit na początku pętli. W kodzie widać dziwne wyrażenie:

```
__attribute__((unused))
```

Jest to rozszerzenie kompilatora GCC dla języka C. Wyrażenie to informuje kompilator, iż zdajemy sobie sprawę, że dana zmienna jest nieużywana, więc nie musi generować ostrzeżenia. (Zmienna faktycznie nie jest używana w programie, ale może zostać wykorzystana w debuggerze. Kompilator nie widzi niczego poza samym programem).

Łańcuch znaków z wiadomością „Hello World" kończy się znakami `\r\n` (powrót karetki, wysuw wiersza). W pierwotnym programie „Witaj, świecie" z rozdziału 1. system operacyjny edytował strumień wyjściowy i zmieniał za nas znak `\n` na `\r\n`. Tu nie mamy systemu operacyjnego, więc wszystko musimy robić sami.

Listing 9.3 zawiera pełną wersję programu „Witaj, świecie" przy użyciu komunikacji szeregowej.

Listing 9.3. Program 09.serial

```
/**
 * @brief Wypisz "hello world" do portu szeregowego.
```

```
 */
#include <stdbool.h>
#include "stm32f0xx_nucleo.h"
#include "stm32f0xx.h"

const char hello[] = "Hello World!\r\n";    // Wysyłany komunikat.
int current; // Znak w wysyłanym komunikacie.

UART_HandleTypeDef uartHandle;        // Inicjacja interfejsu UART.

/**
  * @brief Funkcja ta jest wykonywana w razie wystąpienia błędu.
  *
  * Jedyne, co robi, to miganie diodą LED.
  */
void Error_Handler(void)
{
    /* Włącz diodę LED2. */
    HAL_GPIO_WritePin(LED2_GPIO_PORT, LED2_PIN, GPIO_PIN_SET);

    while (true)
    {
        // Przełącz stan diody LED2.
        HAL_GPIO_TogglePin(LED2_GPIO_PORT, LED2_PIN);
        HAL_Delay(1000);          // Poczekaj 1 sekundę.
    }
}
/**
 * Wyślij znak do interfejsu UART.
 *
 * @param ch Wysyłany znak.
 */
void myPutchar(const char ch)
{
    // W wierszu tym pobiera się i zapisuje wartość bitu UART_FLAG_TXE w czasie
    // wywołania. Wartość ta zmienia się, więc jeśli zatrzymasz program na znajdującym się
    // poniżej wierszu z instrukcją "while", wartość będzie równa 0, bo znika
    // szybciej, niż można na nią spojrzeć.
    int result __attribute__((unused)) =
        (uartHandle.Instance->ISR & UART_FLAG_TXE);

    // Blokuj wykonanie do chwili, gdy ustawiona zostanie flaga pustego rejestru nadawczego (TXE).
    while ((uartHandle.Instance->ISR & UART_FLAG_TXE) == 0)
        continue;

    uartHandle.Instance->TDR = ch;       // Wyślij znak do interfejsu UART.
}

/**
 * Zainicjuj LED2 (po to, by można było zasygnalizować błąd miganiem na czerwono).
 */
void led2_Init(void)
```

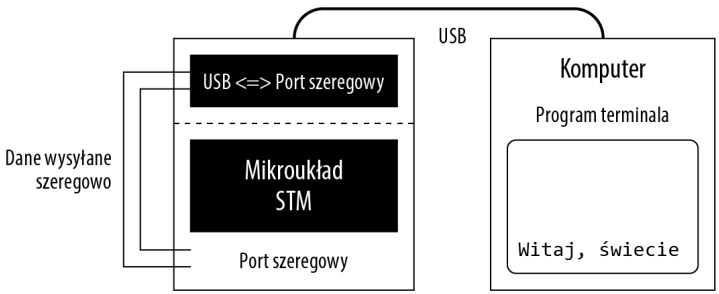
```
{
    // Inicjacja zegara LED.
    LED2_GPIO_CLK_ENABLE();

    GPIO_InitTypeDef GPIO_LedInit;       // Inicjacja LED.
    // Zainicjuj LED.
    GPIO_LedInit.Pin = LED2_PIN;
    GPIO_LedInit.Mode = GPIO_MODE_OUTPUT_PP;
    GPIO_LedInit.Pull = GPIO_PULLUP;
    GPIO_LedInit.Speed = GPIO_SPEED_FREQ_HIGH;
    HAL_GPIO_Init(LED2_GPIO_PORT, &GPIO_LedInit);
}

/**
 * Zainicjuj interfejs UART2 w celu wyprowadzania danych.
 */
void uart2_Init(void)
{
    // Inicjacja interfejsu UART.
    // UART2 -- ten połączony z interfejsem USB programatora ST-LINK.
    uartHandle.Instance = USART2;
    uartHandle.Init.BaudRate = 9600;                        // Prędkość 9600 bodów.
    uartHandle.Init.WordLength = UART_WORDLENGTH_8B;        // 8 bitów na znak.
    uartHandle.Init.StopBits = UART_STOPBITS_1;             // 1 bit stopu.
    uartHandle.Init.Parity = UART_PARITY_NONE;              // Brak kontroli parzystości.
    uartHandle.Init.Mode = UART_MODE_TX_RX;                 // Wysyłanie i odbiór.
    uartHandle.Init.HwFlowCtl = UART_HWCONTROL_NONE;        // Brak sprzętowego sterowania
                                                            // przepływem.

    // Poddaj strumień przychodzący nadpróbkowaniu.
    uartHandle.Init.OverSampling = UART_OVERSAMPLING_16;

    // Nie stosuj jednej próbki na bit.
    uartHandle.Init.OneBitSampling = UART_ONE_BIT_SAMPLE_DISABLE;

    // Żadnych zaawansowanych funkcji.
    uartHandle.AdvancedInit.AdvFeatureInit = UART_ADVFEATURE_NO_INIT;
    /*
     * Dla tych z Was, którzy łączą się przez emulator terminala: powyższe parametry
     * przekładają się na format transmisji 9600,8,N,1.
     */

    if (HAL_UART_Init(&uartHandle) != HAL_OK)
    {
        Error_Handler();
    }
}

int main(void)
{
    HAL_Init(); // Zainicjuj sprzęt.
    led2_Init();
```

```
    uart2_Init();

    // Kontynuuj wysyłanie komunikatu przez długi czas.
    for (;;) {
        // Wysyłaj znak po znaku.
        for(current = 0; hello[current] != '\0'; ++current) {
            myPutchar(hello[current]);
        }
        HAL_Delay(500);
    }
}

/**
 * Tajemna funkcja wywoływana przez warstwę HAL w celu faktycznego
 * zainicjowania interfejsu UART. W tym przypadku musimy
 * przestawić piny UART na tryb alternatywny, by działały jako
 * linie UART, a nie GPIO.
 *
 * @note: Działa tylko z interfejsem UART2, tym połączonym z konwerterem
 * USB/port szeregowy.
 *
 * @param uart Dane UART.
 */
void HAL_UART_MspInit(UART_HandleTypeDef* uart)
{
    GPIO_InitTypeDef GPIO_InitStruct;
    if(uart->Instance == USART2)
    {
        /* Włączenie zegara urządzenia peryferyjnego. */
        __HAL_RCC_USART2_CLK_ENABLE();

        /*
         * Konfigurowanie bitów GPIO interfejsu USART2.
         * PA2    ------> USART2_TX.
         * PA3    ------> USART2_RX.
         */
        GPIO_InitStruct.Pin = GPIO_PIN_2|GPIO_PIN_3;
        GPIO_InitStruct.Mode = GPIO_MODE_AF_PP;
        GPIO_InitStruct.Pull = GPIO_NOPULL;
        GPIO_InitStruct.Speed = GPIO_SPEED_FREQ_LOW;
        // Funkcja alternatywna -- związana z UART.
        GPIO_InitStruct.Alternate = GPIO_AF1_USART2;
        HAL_GPIO_Init(GPIOA, &GPIO_InitStruct);
    }
}

/**
 * Tajemna funkcja wywoływana przez warstwę HAL w celu przywrócenia
 * urządzenia UART do stanu niezainicjowanego. Nigdy tego nie robimy, ale umieściliśmy tu tę
 * funkcję gwoli kompletności.
 *
```

```
 * @note: Działa tylko z interfejsem UART2, tym połączonym z konwerterem
 * USB/port szeregowy.
 *
 * @param uart  Dane UART.
 */
void HAL_UART_MspDeInit(UART_HandleTypeDef* uart)
{
    if(uart->Instance == USART2)
    {
        /* Wyłączenie zegara urządzenia peryferyjnego. */
        __HAL_RCC_USART2_CLK_DISABLE();

        /*
         * Konfigurowanie bitów GPIO interfejsu USART2.
         * PA2    ------> USART2_TX.
         * PA3    ------> USART2_RX.
         */
        HAL_GPIO_DeInit(GPIOA, GPIO_PIN_2|GPIO_PIN_3);
    }
}
```

Komunikacja z urządzeniem

Mamy już program, który wysyła komunikat „Hello World” przez łącze szeregowe. Ma ono połączenie ze znajdującym się na płytce urządzeniem USB/szeregowym, które podłącza się do komputera. Aby zobaczyć komunikat, należy uruchomić na komputerze emulator terminala. Całą konfigurację przedstawiono na rysunku 9.3.



Rysunek 9.3. Komunikacja szeregowa

W każdym systemie operacyjnym znajduje się inny program emulatora terminala, a niekiedy jest więcej niż jeden. Programy wymienione poniżej są często spotykane, darmowe i łatwe w użyciu.

Windows

W systemie Windows użyjemy programu PuTTY (*https://putty.org*). Pobierz i zainstaluj go w systemie, wybierając wszędzie opcje domyślne, a potem wykonaj następujące kroki:

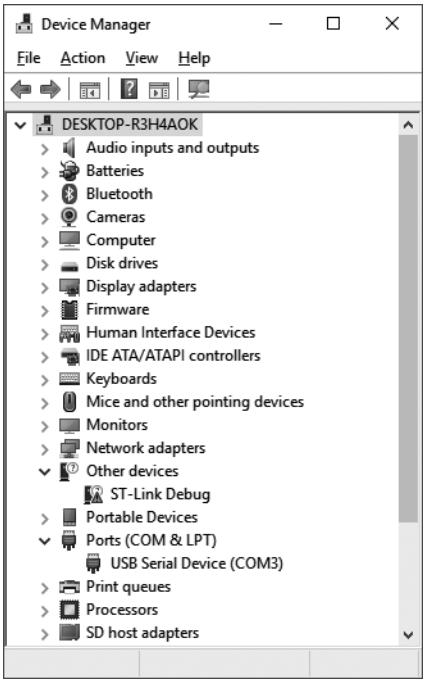
1. Upewnij się, że płytka Nucleo *nie jest* połączona z komputerem. Otwórz Panel sterowania i przejdź do ekranu *Device Manager* (*Menedżer urządzeń* — rysunek 9.4). Na liście nie ma urządzenia sieciowego, a zatem nie ma również sekcji *Ports* (*Porty*).

Rysunek 9.4. Urządzenie szeregowe nie jest zainstalowane

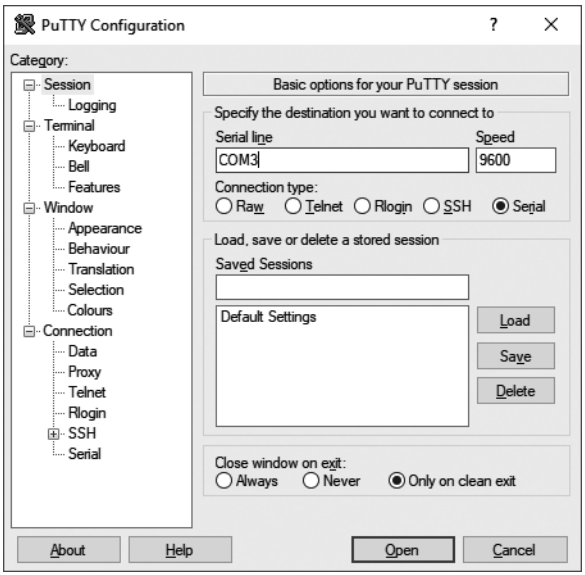
2. Podłącz płytkę Nucleo. Lista urządzeń zmieni się tak, jak to widać na rysunku 9.5.

   Powinno się pojawić nowe urządzenie USB/szeregowe (ang. *USB serial device*) o nazwie COM3. (Windows ma system dotyczący przypisywania portów COM urządzeniom szeregowym, ale nikt nie wie, na czym on polega. Być może na swoim komputerze zobaczysz inny port COM).

3. Uruchom program PuTTY. W głównym oknie, pokazanym na rysunku 9.6, zaznacz przycisk radiowy *Serial* (połączenie szeregowe). W polu pod etykietą *Serial Line* (łącze szeregowe) wybierz nowy port COM, który pokazał się właśnie w Menedżerze urządzeń. W polu *Speed* (prędkość) pozostaw wartość domyślną — 9600.



Rysunek 9.5. Nowe urządzenie USB/szeregowe



Rysunek 9.6. Uruchamianie programu PuTTY

4. Kliknij przycisk *Open* (otwórz). Powinno pojawić się okno terminala, w którym zacznie się wyświetlać komunikat powitalny urządzenia.

Linux i macOS

Na systemach opartych na Uniksie, takich jak Linux i macOS, dobrze działa program `screen`. (Zadanie spełni także program `minicom`). Aby skorzystać z programu `screen`, musisz poznać nazwę urządzenia szeregowego, która różni się w zależności od systemu operacyjnego. W Linuksie urządzenie najprawdopodobniej ma nazwę */dev/ttyACM0*, chociaż jeśli podłączone są inne urządzenia szeregowe, może nazywać się */dev/ttyACM1*, */dev/ttyACM2* lub podobnie. W systemie macOS nazwą urządzenia będzie prawdopodobnie */dev/tty.usbmodem001*, ale może to być też */dev/tty.usbmodem002*, */dev/tty.usbmodem003* czy coś podobnego.

Aby znaleźć nazwę, upewnij się, że płytka Nucleo *nie jest* podłączona do komputera, a potem wykonaj w terminalu jedno z następujących poleceń:

```
$ ls /dev/ttyACM*          (Linux).
$ ls /dev/tty.usbmodem*    (macOS).
```

Podłącz urządzenie i wydaj ponownie to samo polecenie. Na liście powinno pojawić się jeszcze jedno urządzenie. Skorzystaj z niego. Wydaj teraz następujące polecenie:

```
$ screen /dev/ttyACM0 9600
```

Powinny pojawić się napisy „Hello World". Aby wyjść z programu, naciśnij klawisze *Ctrl*+*A*+\.

Podsumowanie

W tej książce na temat programowania przykład „Witaj, świecie" pojawia się dopiero w rozdziale 9. Dlaczego? Dlatego, że wszystko musieliśmy zrobić sami. Napisanie prostego programu wysyłającego wiadomość znak po znaku wymagało zainicjowania interfejsu UART (proces ten nie jest prosty), poinformowania pinu GPIO, że ma być odtąd linią szeregową, i sprawdzenia w rejestrze sprzętowym (tych tajemniczych miejscach w pamięci, które niespodziewanie się zmieniają w zależności od stanu urządzenia), czy interfejs UART jest gotowy do przesłania znaku.

Zrobiliśmy wielki krok do przodu. Po pierwsze, zaprogramowaliśmy średnio złożone urządzenie, a przy tym zdobyliśmy ogrom wiedzy na temat bezpośrednich, niskopoziomowych operacji we/wy. Po drugie, port szeregowy jest głównym narzędziem diagnostycznym i konserwacyjnym w wielu urządzeniach wbudowanych. Mimo całego postępu, jaki dokonał się w informatyce w ostatnich 60 latach, najczęściej stosowaną techniką debugowania wciąż jest wypisywanie danych za pomocą

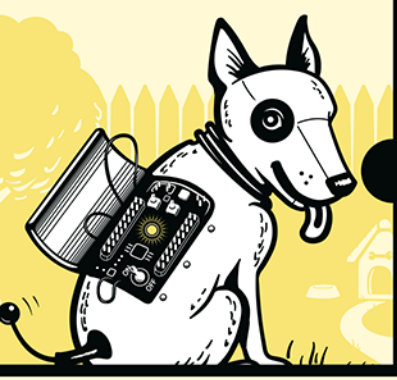
instrukcji `printf` do portu szeregowego. Jest to bardzo proste, niezawodne urządzenie, tanie w produkcji i łatwe do połączenia. Teraz wiemy już, jak przy jego użyciu debugować systemy wbudowane.

Problemy programistyczne

1. Dla ucznia: gdy sprawisz, że program zacznie działać, zobacz, co się stanie, jeśli usuniesz znak `\r`. Spróbuj to zrobić jeszcze raz po wstawieniu z powrotem sekwencji `\r`, a usunięciu `\n`.
2. Umiarkowanie trudna zagadka: spróbuj zmienić konfigurację tak, by wysyłać 7 bitów danych i bit parzystości (zamiast 8 bitów danych bez parzystości). Nie modyfikuj konfiguracji emulatora terminala. Niektóre znaki ulegną zmianie. Zbadaj ich ciągi bitowe i ustal, które znaki się zmieniły i dlaczego.
3. Zaawansowane: obecnie w programie nie ma możliwości sterowania przepływem danych. Otrzymujesz wiadomość „Hello World", czy tego chcesz, czy nie. Zmień kod inicjujący tak, by skorzystać z programowego sterowania przepływem. Wówczas po wpisaniu znaku XOFF (*Ctrl*+*S*) wyprowadzanie danych powinno się zatrzymać, a po wpisaniu znaku XON (*Ctrl*+*Q*) powinno zostać wznowione.
4. Zaawansowane: napisz dla płytki Nucleo funkcję odczytującą znak z portu szeregowego. Może ona wyglądać bardzo podobnie do funkcji `myPutchar`, ale powinna sprawdzać inny bit i odczytywać dane z portu we/wy, zamiast je zapisywać. Musisz przeczytać w dokumentacji mikrokontrolera, za co odpowiada bit RDR.

Notatki


PROGRAM
PARTNERSKI
GRUPY HELION
1. ZAREJESTRUJ SIĘ
2. PREZENTUJ KSIĄŻKI
3. ZBIERAJ PROWIZJĘ
Zmień swoją stronę WWW w działający bankomat!
Dowiedz się więcej i dołącz już dzisiaj!
http://program-partnerski.helion.pl
GRUPA
Helion



PROGRAMOWANIE SYSTEMÓW WBUDOWANYCH? JĘZYK C TO NAJLEPSZY WYBÓR!

Język C jest niezastąpiony w programowaniu niewielkich mikrokontrolerów o ograniczonych zasobach. Takie chipy sterują pracą wielu urządzeń, w tym lodówek, mikrofalówek czy kamer monitoringu. Tworzenie kodu dla systemów wbudowanych nie jest łatwe: brakuje udogodnień oferowanych przez języki wysokiego poziomu i trzeba zachować ścisłą kontrolę nad działaniem programu, gdyż dostępne zasoby pamięci nie są duże, a brak systemu operacyjnego wymusza bezpośrednią komunikację ze sprzętem.

Dzięki tej książce nauczysz się programowania systemów wbudowanych. Do ćwiczeń i eksperymentów posłuży Ci płytka rozwojowa Nucleo z niewielkim, tanim mikroukładem ARM. Dowiesz się, w jaki sposób pisać odpowiedni kod w C, a także zobaczysz, jak jest on tłumaczony na kod maszynowy układu ARM, jak działa kompilator i jakie procesy zachodzą podczas wykonywania kodu w mikroukładzie. Stopniowo, wykonując wiele praktycznych ćwiczeń, przyswoisz zasady pisania niedużych, efektywnych programów, które będą się dobrze sprawdzać w warunkach ograniczonej ilości pamięci, a równocześnie maksymalnie wykorzystywać dostępny potencjał.

W książce między innymi:

- podstawy języka C
- pisanie maksymalnie zoptymalizowanego kodu w C
- działanie kompilatora i przekształcenie kodu w program wykonywalny
- tworzenie kodu, który ma bezpośredni dostęp do operacji wejścia/wyjścia
- pisanie poprawnych procedur obsługi przerwań
- sterty, buforowane systemy wejścia/wyjścia, liczby zmiennoprzecinkowe

Stephen Oualline od ponad 25 lat programuje systemy wbudowane, jest orędownikiem tworzenia niezawodnego i bezbłędnego kodu. Autor wielu książek z dziedziny programowania. Pracuje też jako wolontariusz w Muzeum Kolejnictwa Południowej Kalifornii.

Helion
helion.pl
HELION SA
ul. Kościuszki 1c
44-100 Gliwice
tel.: 32 230 98 63
helion@helion.pl

KOD KORZYŚCI
Sięgnij po więcej! ▶

ISBN 978-83-8322-085-7
9 788383 220857

Cena: 87,00 zł

no starch press