Programowanie mikrokontrolerów 2.0

Uniwersalny interfejs szeregowy, standardowe tekstowe wejście i wyjście

Marcin Engel Marcin Peczarski

Instytut Informatyki Uniwersytetu Warszawskiego

17 października 2021

Uniwersalny interfejs szeregowy

- USART – Universal Synchronous and Asynchronous Receiver and Transmitter
- UART – Universal Asynchronous Receiver and Transmitter

- RS-232
- Smartcard, ISO 7816-3
- IrDA
- LIN
- MIDI

RS-232

- Jeden z najstarszych interfejsów szeregowych
- Pierwotne przeznaczenie to łączenie terminali znakowych z komputerem, często z wykorzystaniem modemów
- Dwa typy urządzeń
  - DTE (ang. *Data Terminal Equipment*) – terminal, komputer
  - DCE (ang. *Data Communication Equipment*, *Data Circuit-terminating Equipment*) – zwykle modem albo inne urządzenie peryferyjne podłączane do komputera
- Prędkości transmisji od kilkudziesięciu b/s do kilkuset kb/s, typowe wartości 1200, 2400, 4800, 9600, 19200, 38400 b/s
- Zasięg do kilkunastu metrów
- Wersja asynchroniczna (bardzo popularna) i synchroniczna (obecnie już bardzo rzadko spotykana)

RS-232, złącza i kable

- Wersja asynchroniczna definiuje 9 drutów: RXD, TXD, CTS, RTS, DSR, DTR, DCD, RI, GND
- Głównie stosuje się złącza D-Sub 9-pinowe lub 25-pinowe, żeńskie lub męskie, . . .

- . . . ale spotyka się też inne, np. RJ
- Kabel prosty łączy DTE z DCE
- Kabel skrzyżowany (ang. *null modem*) łączy DTE z DTE
- Żeby móc połączyć dowolne dwa urządzenia, trzeba mieć przynajmniej 8 różnych typów kabli, . . .
- . . . ale i to może być za mało, bo te 9 drutów można połączyć ze sobą na wiele sposobów

RS-232, wersja uproszczona

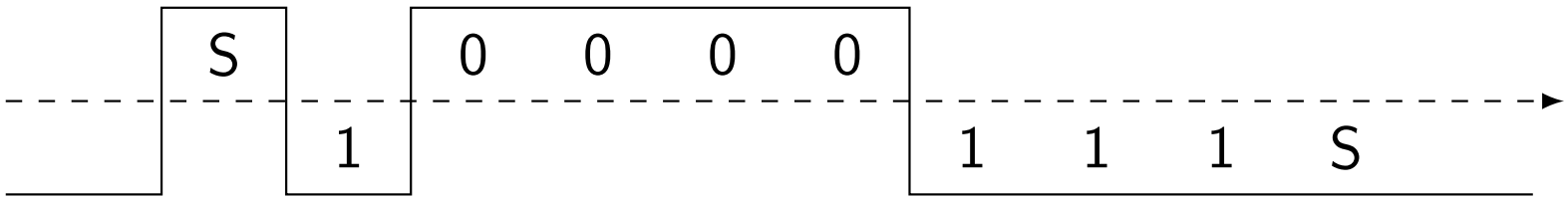
- Prawie każdy mikrokontroler obsługuje RS-232 w wersji uproszczonej, asynchronicznej, 3-drutowej:
  - RXD – odbiór w DTE, nadawanie w DCE
  - TXD – nadawanie w DTE, odbiór w DCE
  - GND – masa

RS-232, protokół komunikacyjny

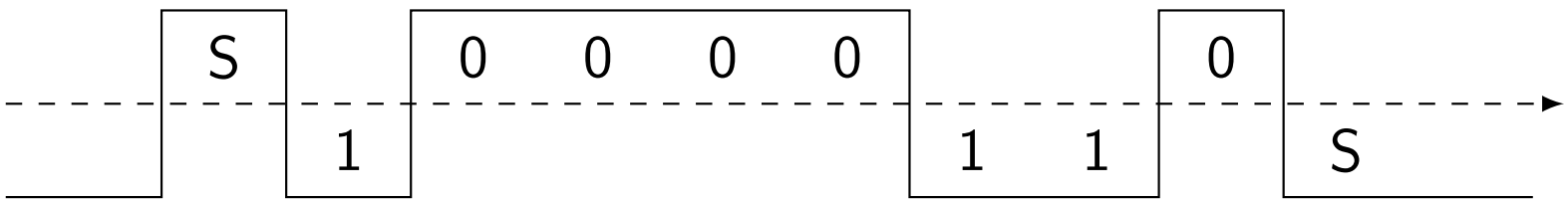
- Sygnalizacja napięciem o wartości bezwzględnej od 3 do 25 V, typowe wartości 5, 10, 12, 15 V
- Dwa poziomy napięcia
  - niski, napięcie ujemne – sygnał mark, logiczna 1, stan off
  - wysoki, napięcie dodatnie – sygnał space, logiczne 0, stan on
- Pojedyncza transmisja asynchroniczna
  - 1 bit startowy, space, logiczne 0
  - 5 do 9 bitów danych, typowo 7 lub 8, najpierw najmniej znaczący (LSB)
  - opcjonalny bit parzystości lub nieparzystości
  - 1 bit lub 1,5 bita lub 2 bity stopu, mark, logiczna 1
- Najczęściej spotykane kombinacje to
  - 7E1 – 7 bitów danych, bit parzystości, 1 bit stopu
  - 8N1 – 8 bitów danych, brak kontroli parzystości, 1 bit stopu
- Dowolnej długości przerwa między kolejnymi transmisjami

RS-232, przebiegi czasowe

- 7E1, litera 'a', kod 0x61

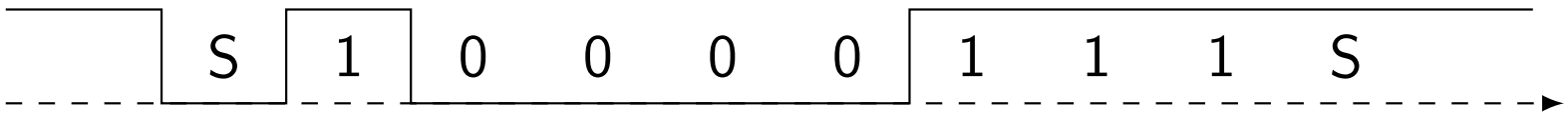


- 8N1, litera 'a', kod 0x61

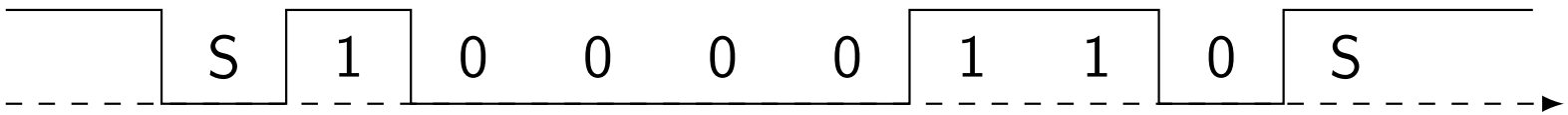


UART, przebiegi czasowe

- Na wyjściu mikrokontrolera logiczne 0 to stan niski, czyli napięcie 0 V, a logiczna 1 to stan wysoki, czyli napięcie zasilania
- 7E1, litera 'a', kod 0x61



- 8N1, litera 'a', kod 0x61



- Uzyskanie poziomów napięć wg standardu RS-232 wymaga podłączenia konwertera poziomów
- Gdy łączymy dwa mikrokontrolery, konwerter nie jest potrzebny

U(S)ART w STM32

- Mikrokontrolery STM32 mają zwykle kilka układów USART i UART
- USART może pracować jako asynchroniczny UART
- Obsługa poprzez rejestry: `BRR`, `CR1`, `CR2`, `CR3`, `DR`, `SR`, ...
- Patrz Data sheet STM32F411xC/E, rozdział 4, tabela 9: Alternate function mapping
- Patrz Reference manual RM0383, rozdział 19: Universal synchronous asynchronous receiver transmitter (USART)

Konfigurowanie

- Będziemy korzystać z `USART2`, którego linia TXD jest wyprowadzona na `PA2`, a linia RXD – na `PA3`
- Włączamy potrzebne pliki nagłówkowe

```
#include <stm32.h>
#include <gpio.h>
```

- Włączamy taktowanie odpowiednich układów peryferyjnych

```
RCC->AHB1ENR |= RCC_AHB1ENR_GPIOAEN;
RCC->APB1ENR |= RCC_APB1ENR_USART2EN;
```

Konfigurowanie

- Konfigurujemy linię TXD

```
GPIOafConfigure(GPIOA,
                2,
                GPIO_OType_PP,
                GPIO_Fast_Speed,
                GPIO_PuPd_NOPULL,
                GPIO_AF_USART2);
```

- Konfigurujemy linię RXD

```
GPIOafConfigure(GPIOA,
                3,
                GPIO_OType_PP,
                GPIO_Fast_Speed,
                GPIO_PuPd_UP,
                GPIO_AF_USART2);
```

Rejestr CR1

- Tryb pracy

```
#define USART_Mode_Rx_Tx (USART_CR1_RE | \
                          USART_CR1_TE)
#define USART_Enable     USART_CR1_UE
```

- Przesyłane słowo to dane łącznie z ewentualnym bitem parzystości

```
#define USART_WordLength_8b 0x0000
#define USART_WordLength_9b USART_CR1_M
```

- Bit parzystości

```
#define USART_Parity_No   0x0000
#define USART_Parity_Even USART_CR1_PCE
#define USART_Parity_Odd  (USART_CR1_PCE | \
                           USART_CR1_PS)
```

Rejestr CR1

- Przykładowa konfiguracja (układ pozostaje nieaktywny – nie ustawiamy bitu `USART_Enable`)

```
USART2->CR1 = USART_Mode_Rx_Tx |
              USART_WordLength_8b |
              USART_Parity_No;
```

Rejestr CR2

- Bit(y) stopu

```
#define USART_StopBits_1   0x0000
#define USART_StopBits_0_5 0x1000
#define USART_StopBits_2   0x2000
#define USART_StopBits_1_5 0x3000
```

- Przykładowa konfiguracja

```
USART2->CR2 = USART_StopBits_1;
```

Rejestr CR3

- Sterowanie przepływem

```
#define USART_FlowControl_None 0x0000
#define USART_FlowControl_RTS  USART_CR3_RTSE
#define USART_FlowControl_CTS  USART_CR3_CTSE
```

- Przykładowa konfiguracja

```
USART2->CR3 = USART_FlowControl_None;
```

Rejestr BRR

- Po włączeniu mikrokontroler STM32F411 jest taktowany wewnętrznym generatorem RC HSI (ang. *High Speed Internal*) o częstotliwości 16 MHz

```
#define HSI_HZ 16000000U
```

- Układ `UART2` jest taktowany zegarem PCLK1, który po włączeniu mikrokontrolera jest zegarem HSI

```
#define PCLK1_HZ HSI_HZ
```

- Przykładowa konfiguracja

```
#define BAUD_RATE 9600U
USART2->BRR = (PCLK1_HZ + (BAUD_RATE / 2U)) /
              BAUD_RATE;
```

Uaktywnienie interfejsu

- Ustawiamy bit `UE` w rejestrze `CR1`

  ```
  USART2->CR1 |= USART_Enable;
  ```

- Teraz można już przesyłać dane

Odbieranie znaku

- Na razie aktywne oczekiwanie, bez sprawdzania błędów
- Po odebraniu znaku w rejestrze `SR` ustawiany jest bit `RXNE` (ang. *rx data register not empty*), czyli spełniony jest warunek (przykład dla `USART2`)

```
USART2->SR & USART_SR_RXNE
```

- Teraz można odczytać odebrany znak, czytając rejestr `DR`

```
char c;
c = USART2->DR;
```

- Bit `RXNE` zeruje się automatycznie po odczytaniu rejestru `DR`

Wysyłanie znaku

- Na razie aktywne oczekiwanie, bez sprawdzania błędów
- Znak można wstawić do wysłania, gdy w rejestrze `SR` ustawiony jest bit `TXE` (ang. *tx data register empty*), czyli gdy spełniony jest warunek (przykład dla `USART2`)

  ```
  USART2->SR & USART_SR_TXE
  ```

- Znak wstawia się do wysłania, pisząc do rejestru `DR`

  ```
  char c = 'a';
  USART2->DR = c;
  ```

- Bit `TXE` zeruje się automatycznie po zapisie do rejestru `DR`, a jest ponownie ustawiany, gdy znak został skopiowany do wysłania do rejestru przesuwającego (jego wysyłanie nie musiało się zakończyć)

Minicom

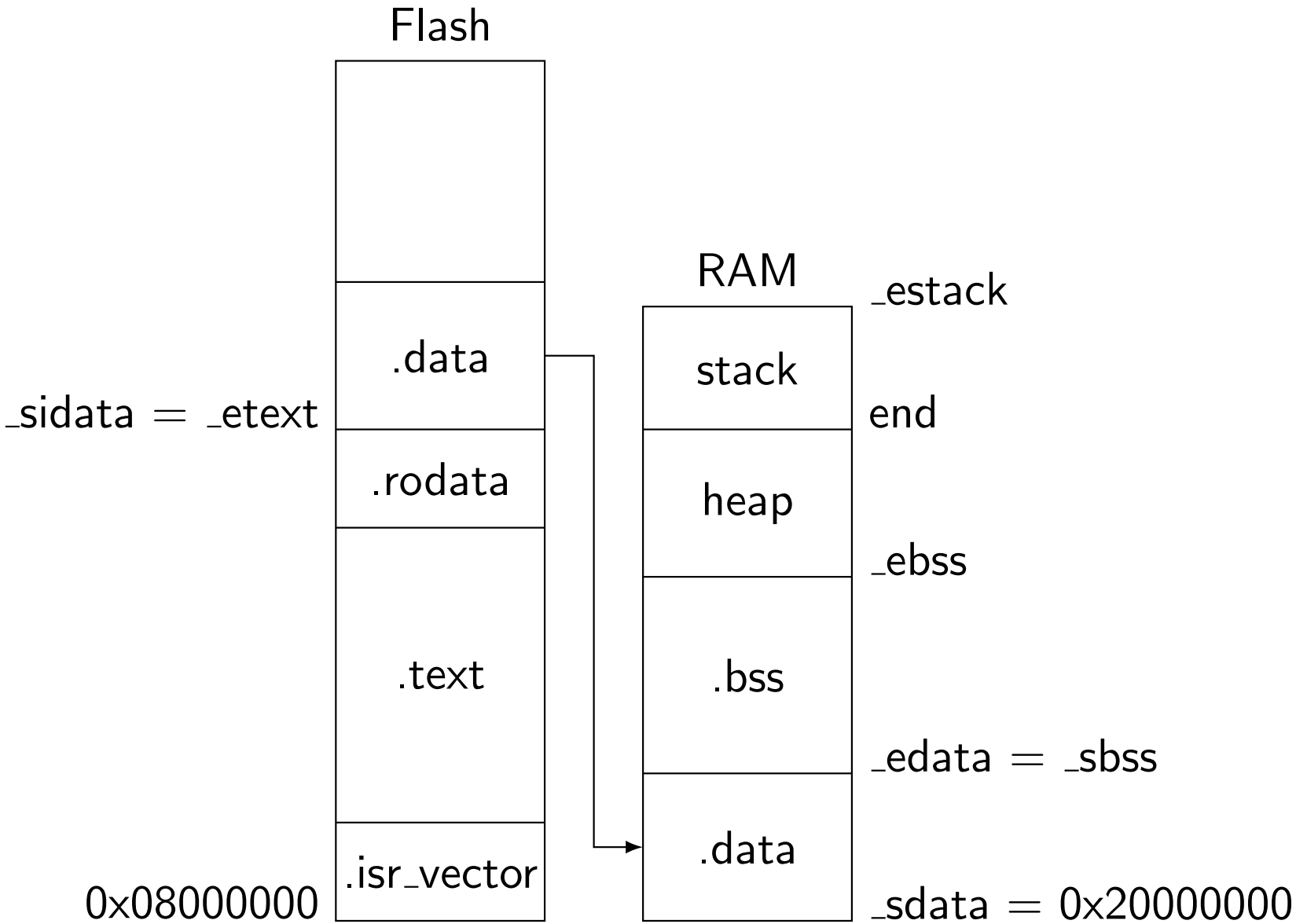
- USART2 mikrokontrolera podłączony jest do ST-LINK/V2-1 i widoczny jako wirtualny port szeregowy: w Linuksie urządzenie /dev/ttyACM0
- Pod Linuksem do komunikacji możemy skorzystać z programu minicom
- Przykładowa konfiguracja w pliku .minirc.dfl

```
pu port             /dev/ttyACM0
pu baudrate         9600
pu bits             8
pu parity           N
pu stopbits         1
pu rtscts           No
pu minit
pu mreset
```

Jak lepiej obsługiwać USART?

- Aktywne oczekiwanie na zdarzenie (odebranie znaku, przekazanie znaku do wysłania, zakończenie wysyłania znaku itp.) jest kiepskim rozwiązaniem
- Zdarzenia mogą zgłaszać przerwania
- Przesyłanie długich ciągów danych można usprawnić, korzystając z DMA
- O tym w przyszłości

Organizacja pamięci programu



Jak to działa w Cortex-M?

- Element 0 tablicy przerwań zawiera adres `_estack`
- Element 1 tablicy przerwań zawiera adres, od którego ma się rozpocząć wykonywanie programu – funkcja `Reset_Handler`
- Po włączeniu zasilania lub wyzerowaniu
  - rejestr `SP` jest inicjowany adresem `_estack`
  - rejestr `PC` jest inicjowany adresem `Reset_Handler`
- Zadaniem funkcji `Reset_Handler` jest
  - skopiowanie sekcji `.data` z Flash do RAM
  - wyzerowanie sekcji `.bss`
  - wywołanie funkcji `main`
  - obsłużenie wartości zwróconej przez funkcję `main`

Jak to działa w Cortex-M?

- Początkowe adresy Flash i RAM oraz ich rozmiary zdefiniowano w pliku `stm32f411re.lds`
- Rozmieszczenia sekcji w pamięci zdefiniowano w pliku `cortex-m.lds`
- Pliki te są dostępne w labie w katalogu `/opt/arm/stm32/lds`
- Tekst źródłowy funkcji `Reset_Handler` znajduje się w pliku `startup_stm32.c`
- Pliki ten jest dostępny w labie w katalogu `/opt/arm/stm32/src`
- Tablicę przerwań zdefiniowano w plikach `interrupt_vector_stm32.c` i `interrupt_vector_stm32f411xe.c`
- Pliki te są dostępne w labie w katalogu `/opt/arm/stm32/inc`

Jak wymusić rozmieszczenie danych z poziomu języka C?

- Wektor przerwań

```
__attribute__ ((section(".isr_vector")))
void (* const g_pfnVectors[])(void) = {
  (void*)&_estack,
  Reset_Handler,
  ...
  SPI5_IRQHandler
};
```

- Bufor DMA

```
#define __dma_aligned \
  __attribute__((section(".dmamem"), aligned(4)))

static packet_t packet __dma_aligned;
```

Biblioteka standardowa języka C

- W laboratorium korzystamy z `newlib` – implementacji standardowej biblioteki języka C przeznaczonej dla systemów wbudowanych
- Biblioteka języka C powinna implementować funkcje `printf`, `fprintf`, `sprintf`, `snprintf`, ...

Dla ciekawskich – wyjście formatowane

- Spróbujmy skompilować następujący program

```
#include <stdio.h>
#define SIZE 100

int main() {
  char buf[SIZE];
  int x = 23;
  snprintf(buf, SIZE, "Temperatura %d\n", x);
  return 0;
}
```

- Efekt

```
sbrkr.c:(.text._sbrk_r+0xc): undefined
reference to `_sbrk'
collect2: error: ld returned 1 exit status
```

- Trzeba zaimplementować funkcję `_sbrk`
- Patrz plik `sbrk.c` w katalogu `/opt/arm/stm32/src`

Wyjście formatowane, cd.

- Chcemy wypisywać komunikaty na różnych urządzeniach peryferyjnych (UART, LCD, ...) za pomocą biblioteki standardowej
- Przyjmiemy upraszczające założenia:
  - z konkretnymi urządzeniami związujemy na stałe konkretne deskryptory
  - standardowe wejście, wyjście i strumień diagnostyczny będą związane z UART-em
  - kolejne deskryptory mogą oznaczać LCD, BT itp.

Formatowanie wyjście, cd.

- Spróbujmy skompilować program

```
#include <stdio.h>

int main() {
  printf("Hello world\n");

  for (;;);
}
```

- Linker zgłasza, że brakuje funkcji: `_read`, `_write`, `_close`, `_fstat`, `_isatty`, `_lseek`
- Trzeba je zaimplementować

Implementacja funkcji _read

- Funkcja _read blokuje się w oczekiwaniu na jeden znak

```
long _read(int fd, char *ptr, long len) {
  if (ptr == 0 || len <= 0) {
    errno = EINVAL; return -1;
  }
  else if (fd == STDIN_FILENO) {
    while (!(USART->SR & USART_SR_RXNE));
    *ptr = USART->DR;
    errno = 0; return 1;
  }
  else {
    errno = EBADF; return -1;
  }
}
```

Implementacja funkcji `_write`

- Funkcja `_write` blokuje się, zapisując po jednym znaku

```
long _write(int fd, char const *ptr, long len) {
  long tmp = len;
  if (ptr == 0 || len < 0) {
    errno = EINVAL; return -1;
  }
  else if (fd == STDOUT_FILENO ||
           fd == STDERR_FILENO) {
    while (tmp--) {
      while (!(USART->SR & USART_SR_TXE));
      USART->DR = *ptr++;
    }
    errno = 0; return len;
  } else {
    errno = EBADF; return -1;
  }
}
```

Implementacja pozostałych funkcji

- Deskryptory prowadzą do urządzeń znakowych:

```
int _fstat(int fd, struct stat *st) {
  if (st == 0) {
    errno = EINVAL; return -1;
  }
  else if (fd == STDIN_FILENO ||
           fd == STDOUT_FILENO ||
           fd == STDERR_FILENO) {
    memset(st, 0, sizeof(struct stat));
    st->st_mode = S_IFCHR | 0666;
    errno = 0; return 0;
  }
  else {
    errno = EBADF; return -1;
  }
}
```

Implementacja pozostałych funkcji

- Przewijanie urządzenia nie ma żadnego efektu:

```
long _lseek(int fd, long offset, int whence) {
  errno = 0; return 0;
}
```

- Deskryptory prowadzą do terminali:

```
int _isatty(int fd) {
  errno = 0;
  return fd == STDIN_FILENO  ||
         fd == STDOUT_FILENO ||
         fd == STDERR_FILENO;
}
```

- Zamykanie nie ma żadnego efektu:

```
int _close(int fd) {
  errno = 0; return 0;
}
```

Implementacja funkcji _open

- Teraz kompilacja przebiega poprawnie
- Ale urządzenie trzeba jeszcze otworzyć przed użyciem
- A do tego jest potrzebna funkcja _open

```
int _open(const char *path, int flags, ...) {
  if (strncmp(path, "tty", 3) == 0) {
    init_usart();
    errno = 0; return STDOUT_FILENO;
  }
  else {
    errno = EACCES; return -1;
  }
}
```

Wyjście formatowane

- Kompletny program

```
#include <stdio.h>

int main() {
  fopen("tty", "w");
  printf("Hello world\n");

  for (;;);
}
```

- Uwagi:
  - `iprintf` – „lekka”, ograniczona do formatowania liczb całkowitych, wersja `printf`
  - biblioteka standardowa buforuje wejście-wyjście