Systemy wbudowane dla automatyki W02

dr inż. Krzysztof Urbański

Dodatkowe materiały do kursu

na prawach rękopisu

Źródła: materiały własne, publicznie dostępne dokumenty w dostępie otwartym oraz noty katalogowe i oficjalna dokumentacja prezentowanych rozwiązań.

```
int t[10];
t[10] = 7;

cout << t[20] << endl;

//zagadka: co strasznego się wydarzy?
```

Kompilowanie kodu:

```
g++ --stack-check -W -Wall main.cpp -o xxx
```

Testy:

```
valgrind ./xxx
```

Uwaga! Efekty tych błędów zarządzania pamięcią mogą być różne - zależnie od kompilatora, systemu, szczęścia.

Może być wymagane jest zainstalowanie pakietu valgrind:

```
sudo apt-get install valgrind
```

Przykład działania

```
#include <stdlib.h>
void f(void)
{
     int* x = malloc(10 * sizeof(int));
     x[10] = 0;       // problem 1: heap block overrun
}                     // problem 2: memory leak -- x not freed

int main(void)
{
     f();
     return 0;
}
```

```
int main()
{
   int* a = new int(33);
   *a = 7;
   return 0;
}
$ valgrind --leak-check=full ./Lab09a
==4060== HEAP SUMMARY:
==4060==     in use at exit: 4 bytes in 1 blocks
==4060==   total heap usage: 1 allocs, 0 frees, 4 bytes allocated
==4060==
==4060== 4 bytes in 1 blocks are definitely lost in loss record 1 of 1
==4060==    at 0x4C286E7: operator new(unsigned long) (vg_replace_malloc.c:287)
==4060==    by 0x400976: main (main.cpp:14)
==4060==
==4060== LEAK SUMMARY:
==4060==    definitely lost: 4 bytes in 1 blocks
==4060==    indirectly lost: 0 bytes in 0 blocks
==4060==      possibly lost: 0 bytes in 0 blocks
==4060==    still reachable: 0 bytes in 0 blocks
==4060==         suppressed: 0 bytes in 0 blocks
```

…I z tego wynikają główne problemy z C oraz C++ (tu w mniejszym stopniu).

➔ **MISRA** C:2023 (Motor Industry Software Reliability Association)

Sekwencje specjalne (*escape sequences*)

\a 07 BELL
\b 08 BS
\f 0C FF (drukarka)
\n 0A LF (uwaga na systemy Microsoftu!)
\r 0D CR
\t 09 HTAB
\v 0B VTAB (drukarka)
\\ 5C \
\' 27 '
\" 22 "
\? 3F ?
\0 00 NULL
\ooo ósemkowo
\xhh F szesnastkowo
\uhhhh,\xhhhh UNICODE (16 bitów, UTF-16)
\e 1B Esc (GCC)

Wskazówka. Zamiast 13 lub 10 używaj w kodzie terminala \r, \n, a do tego warto wiedzieć co znaczą b'\xhh' w Pythonie.

Zmienne

Podstawowe typy danych:

- nieokreślony: void
- wyliczeniowy: enum
- logiczny: bool (true/false albo TRUE/FALSE w C)
- całkowite: char, int
- zmiennoprzecinkowe: float, double
- wskaźnikowe: przez dodanie * (gwiazdki) po nazwie typu
- modyfikator: unsigned, signed, short, long,
- modyfikator: volatile

...lub lepiej: **int32_t, uint8_t**

Zadanie.

Janusz dostał 5 jabłek. 3 jabłka zjadł.

Pytanie: Ile jabłek zostało Januszowi?

Myślicie, że 2?
Guzik prawda! Nie wiadomo przecież, ile jabłek miał, zanim dostał 5 jabłek.
Morał: Zawsze zerujcie zmienne!

Nie do końca prawda – nie zawsze jest to potrzebne lub wskazane.

Zmienne w C/C++ ze względu na obszar pamięci i widoczność

- Globalne
  - Statyczne w module
  - Statyczne w funkcji
- Lokalne (w tym argumenty funkcji)
- Alokowane dynamicznie
- Rejestrowe
- Adresowane jawnie
- „Zmienne” const (!)

W MCU możliwe (i częste) są odstępstwa od standardu ANSI C.

```c
typedef enum nmbs_error {
    // Library errors
    NMBS_ERROR_INVALID_REQUEST = -8,
    NMBS_ERROR_INVALID_UNIT_ID = -7,
} nmbs_error;


const int NMBS_ERROR_INVALID_REQUEST = -8;
const int NMBS_ERROR_INVALID_UNIT_ID = -7;


#define NMBS_ERROR_INVALID_REQUEST -8
#define NMBS_ERROR_INVALID_UNIT_ID -7
```

```
extern int zmienna;

int zmienna;
```

→demonstracja: VS

UART: uniwersalny asynchroniczny odbiornik-nadajnik

(ang. universal asynchronous receiver-transmitter)
USART: uniwersalny synchroniczny/asynchroniczny odbiornik/nadajnik
...oraz rzadziej spotykany LPUART (ang. low-power)

Powszechny układ peryferyjny w mikrokontrolerach, ale w formie RS232 także w starszych komputerach klasy PC (8250, 16450, 16550A).

- Asynchroniczny: brak dedykowanej linii zegarowej (jak „zgrać” Tx i Rx?) ...chociaż w pewnych przypadkach dzięki dodatkowemu zegarowi bywa także synchroniczny.
- Dwukierunkowy *full-duplex*: możliwa transmisja w obu kierunkach jednocześnie (możliwe też half-duplex i simplex).
- Szeregowy: znak jest przesyłany bit po bicie.
- Interfejs zorientowany znakowo (to nie to samo co bajtowo!)
- Budowa ramki danych: bit startu, dane, pit parzystości, bit stopu
- Poziomy napięć

Historycznie: teletypewriter (tty) – zautomatyzowany, elektromechaniczny telegraf, używający ciągów 5,6,7 lub 8-bitowych → /dev/ttyX, /dev/ttyACM0

Urządzenie USB klasy CDC ACM:

CDC: Communications Device Class

ACM: Abstract Control Model

Strony wcześniej uzgadniają:

- Poziomy napięć
- Baud Rate (prędkość kodowania)
- Bit parzystości
- Liczbę bitów na znak
- Długość bitu STOP
- Kontrolę przepływu (sprzętowa, programowa, brak)

Budowa **ramki** UART

- Idle (linia nieaktywna, stan logiczny 1).
- Start bit (linia zmienia stan na 0, bit startu).
- Data bits: 5..9 bitów 0/1 reprezentujących kod przesyłanego znaku, LSB first.
- Parity bit: opcjonalny bit parzystości. None=brak, albo Even/Odd
- Stop (linia przechodzi w stan 1): trwa to 1/1,5/2 bity, I dopiero po tym czasie można przesłać kolejny znak. Jeśli linia zmieni stan przed tym czasem, to błąd ramki (framing error).

8N1

START	D0	D1	D2	D3	D4	D5	D6	D7	STOP

5E2 lub 5O2

START	D0	D1	D2	D3	D4	D5	parity	STOP

Błędy transmisji i stany interfejsu UART:

- Overrun – odbiornik nie zdążył przetworzyć znaku przed odebraniem kolejnego.
- Framing error (błąd ramki) – niewłaściwy stan linii w momencie spodziewanego odebrania bitu STOP.
- Parity error (błąd parzystości) – liczba jedynek w przesyłanych danych jest niezgodna z bitem parzystości (opcjonalnie, tylko jeśli występuje kontrola parzystości).
- Underrun – nadajnik wysłał wszystkie znaki (pusty bufor nadajnika).
- Break condition (przerwa w transmisji) – odbiornik nie odebrał znaku przez pewien czas (tzn. nie pojawiło się nic, co zawiera poprawne bity START i STOP) .

Kontrola przepływu (ang. flow control)

Sprzętowa:

- DTE sygnalizuje na linii RTS, że jest gotowy do odebrania danych z DCE.
- DCE sygnalizuje na linii CTS, że jest gotowy do odebrania danych z DTE.

Programowa:

- XON / XOFF

Kod	Funkcja	DEC	HEX	Skrót klawiszowy
XOFF	Pauza	19	0x13	Ctrl+S
XON	Wznowienie	17	0x17	Ctrl+Q

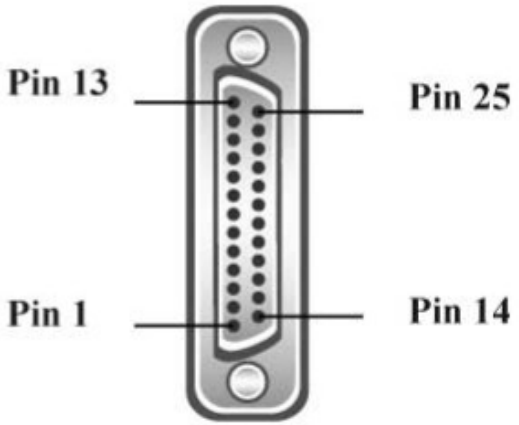
RS-232 (TIA/EIA-232)

ang. *Recommended Standard; Electronic Industries Alliance*

Oryginalnie 20 kbps, do 15 m, wersja asynchronicza lub synchroniczna

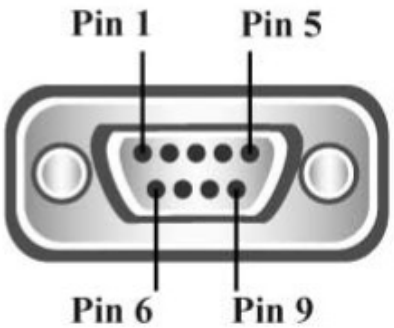
Pin 2	TXD
Pin 3	RXD
Pin 4	RTS
Pin 5	CTS
Pin 6	DSR
Pin 7	GND
Pin 8	DCD
Pin 20	DTR
Pin 22	RI

RS232 Pinout (25 Pin Male)



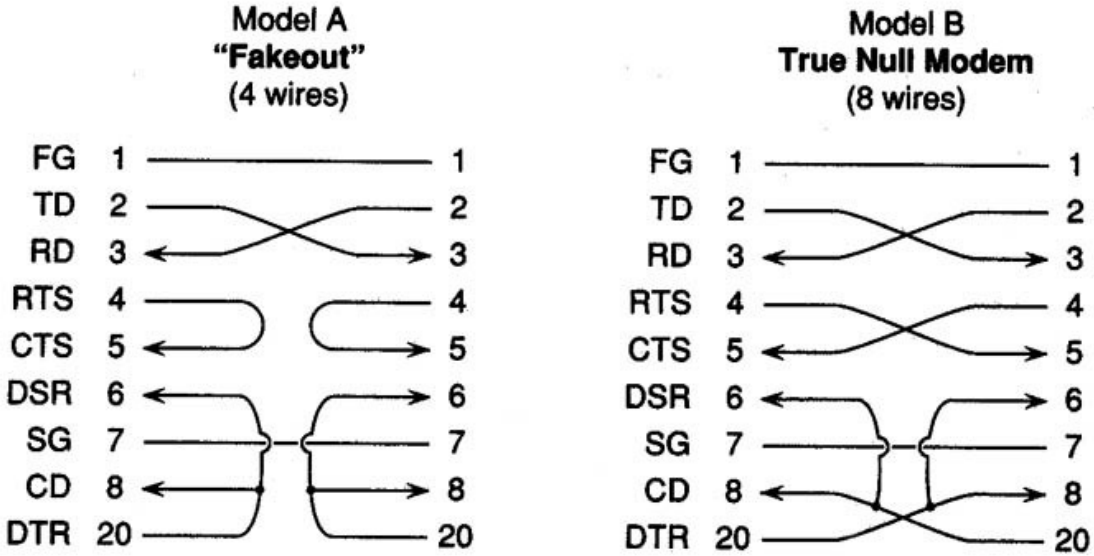
Pin 1	N/C
Pin 2	RXD
Pin 3	TXD
Pin 4	DTR
Pin 5	GND
Pin 6	DSR
Pin 7	RTS
Pin 8	CTS
Pin 9	Power Input

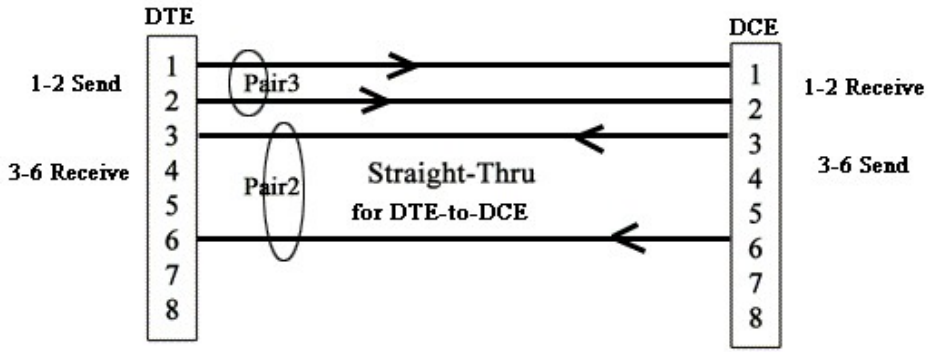
RS232 Pinout (9 Pin Male)

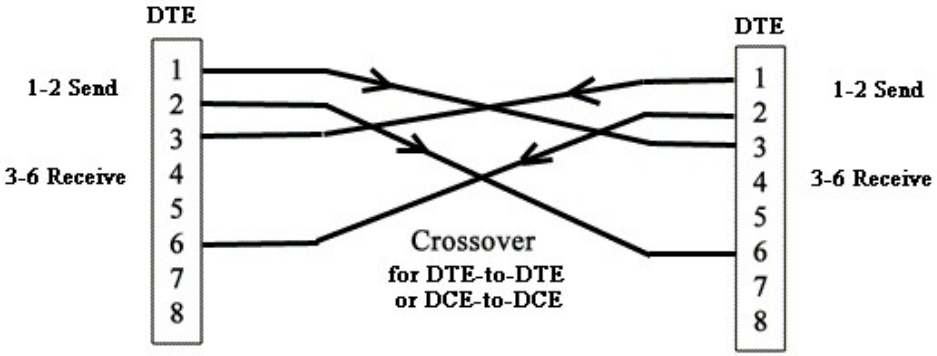


DCE: Data Communication Equipment
DTE: Data Terminal Equipment
DTR: Data Terminal Ready
DCD: Data Carrier Detect
DSR: Data Set Ready
RTS: Request To Send
CTS: Clear To Send
TxD: Transmitted Data
RxD: Received Data

Kabel połączeniowy RS232



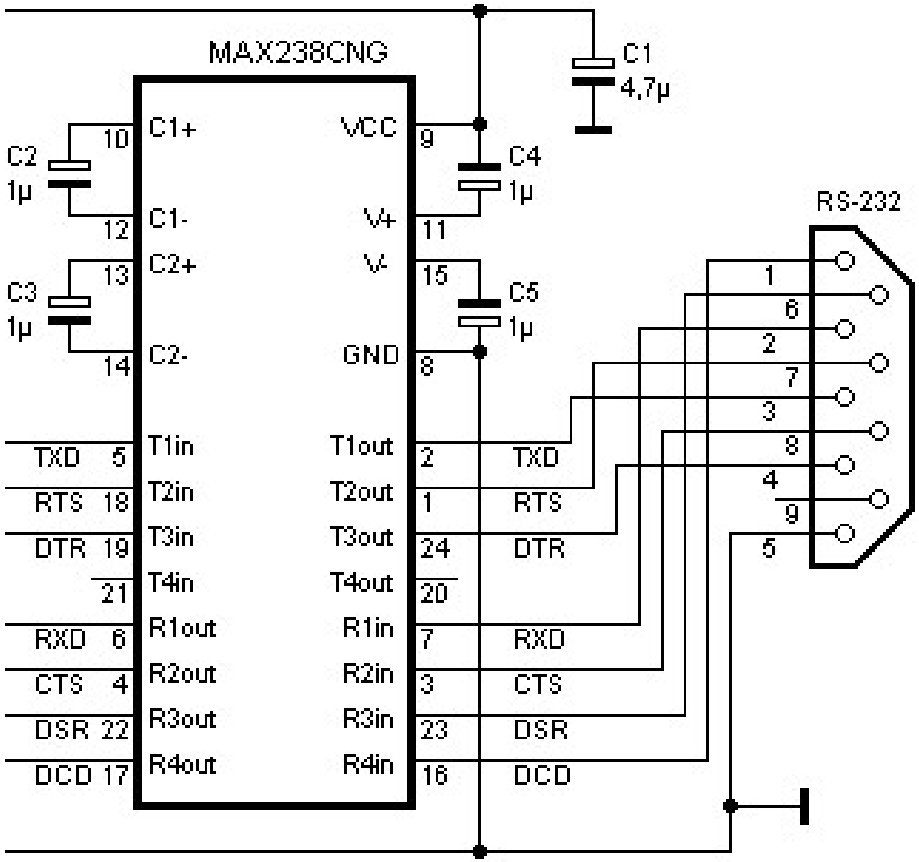


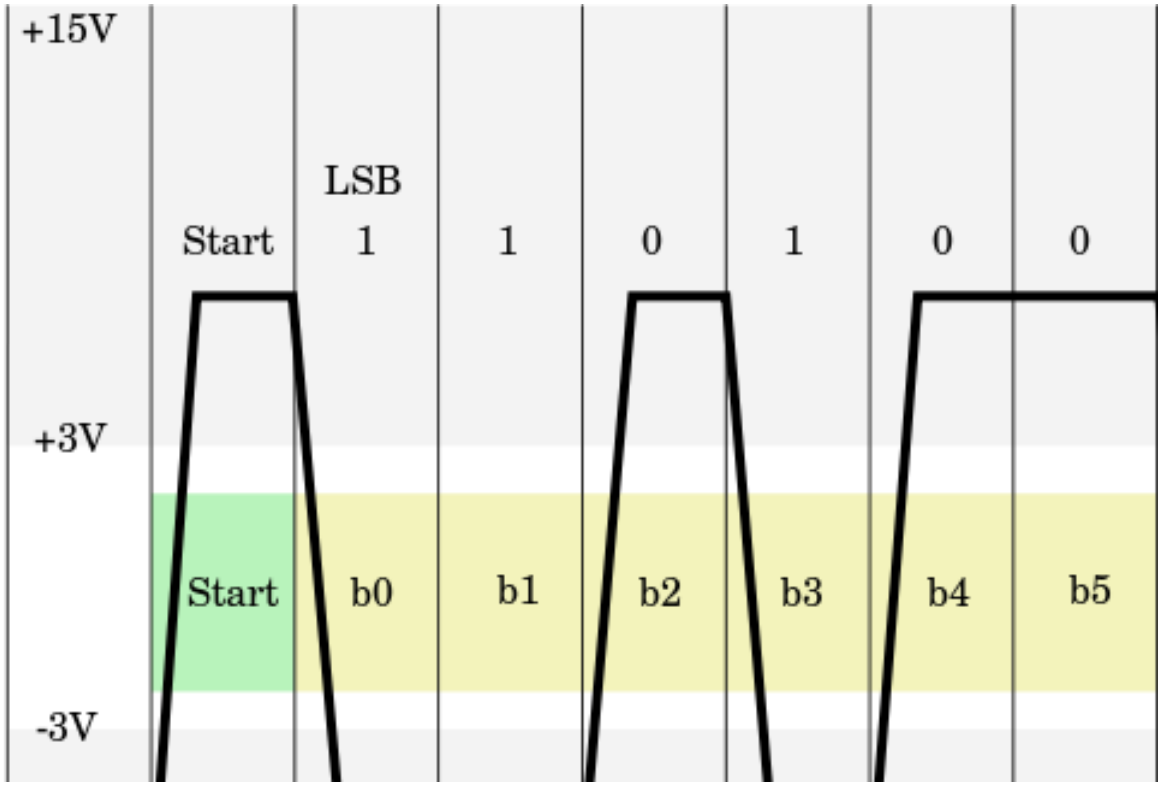


Konwersja poziomów napięć RS232-TTL (UART)

Sygnał	Wartość logiczna	Napięcie UART	Napięcie RS232	Napięcie RS485
mark	1	+3.3…+5 V	-3…-15 V	A+ / B–
space	0	0V (GND)	+3…+15 V	A– / B+

Najpopularniejszy konwerter to MAX232, ale można spotkać wiele układów o podobnej funkcjonalności.

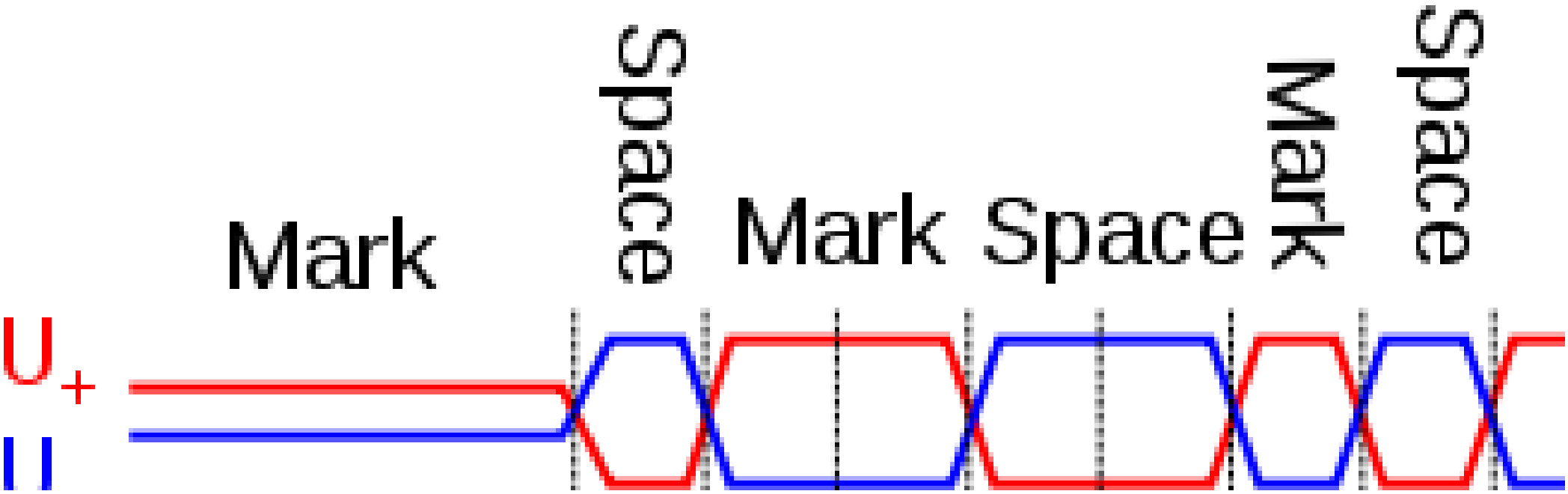


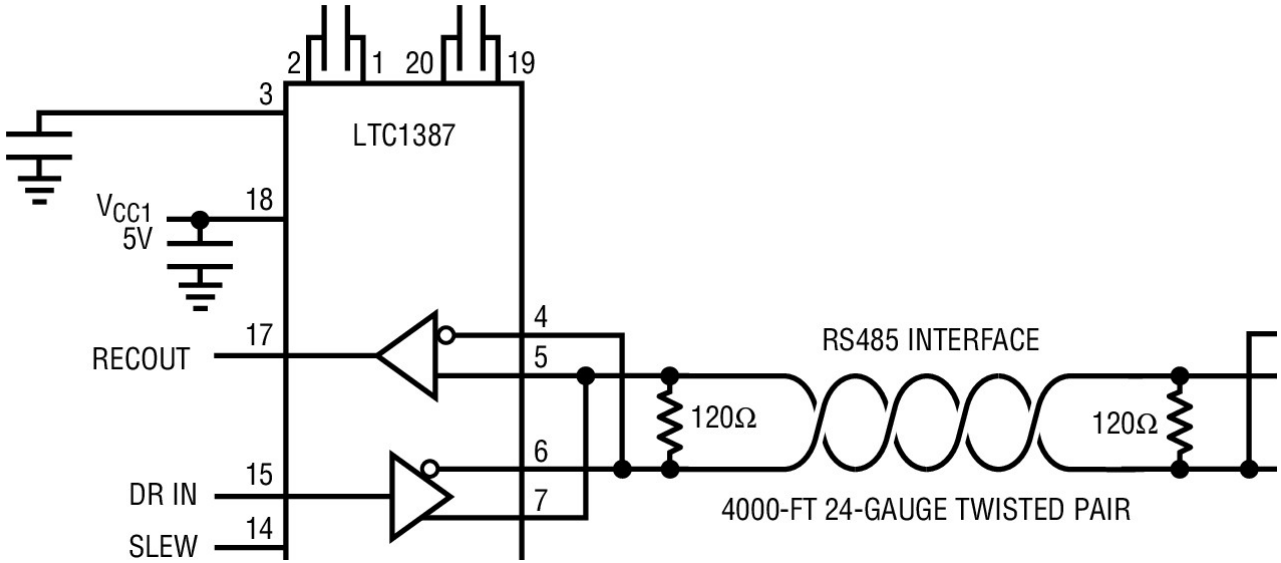


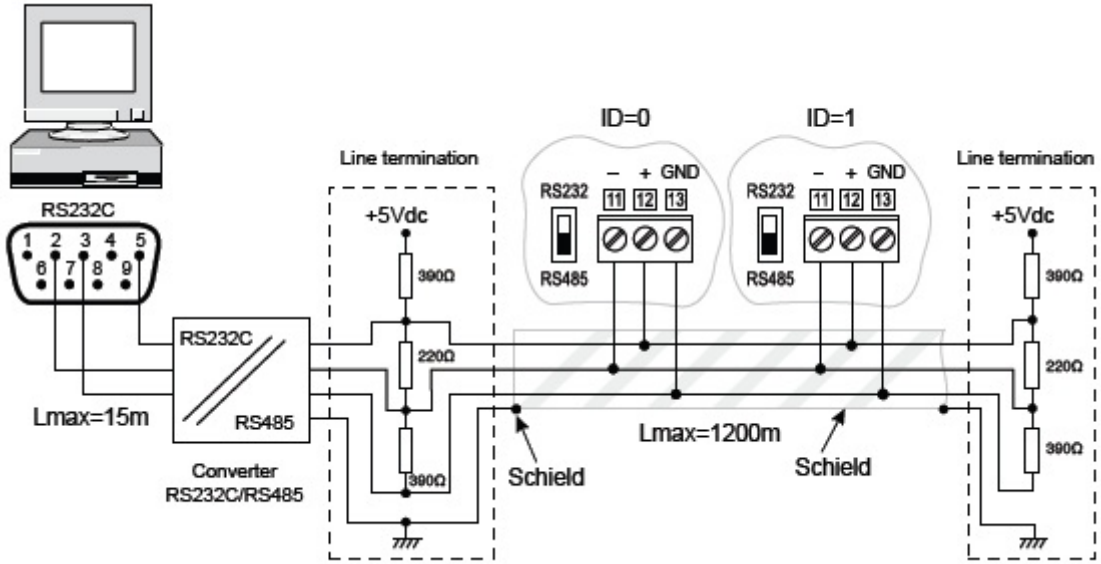
Transmisja znaku „K”

RS485 (aka: TIA-485-A, ANSI/TIA/EIA-485, TIA/EIA-485, EIA-485)

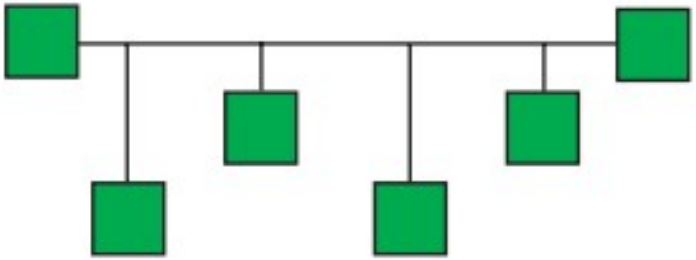
- Topologia magistrali, wg. specyfikacji do 1200 m
- Linia symetryczna (różnicowa) →dlaczego?
- Half-duplex: 2 przewody (opcjonalnie GND)
- Full-duplex: 4 przewody (opcjonalnie GND, VDD)



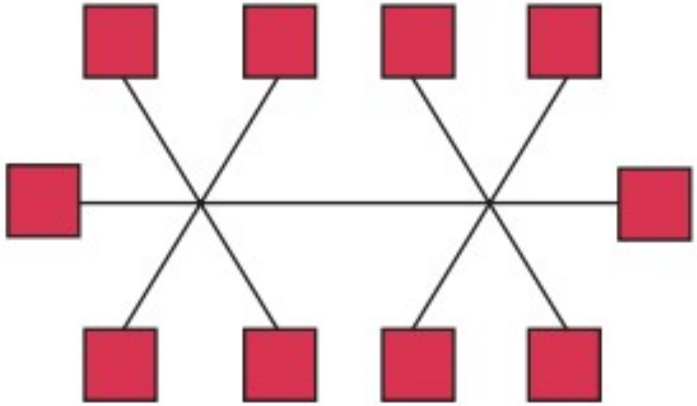

LTC1387
2
1
20
19
3
18
V_{CC1}
5V
17
RECOUT
15
DR IN
14
SLEW
4
5
6
7
120Ω
RS485 INTERFACE
120Ω
4000-FT 24-GAUGE TWISTED PAIR


RS232C
1 2 3 4 5
6 7 8 9
Lmax=15m
RS232C
RS485
Converter
RS232C/RS485
Line termination
+5Vdc
390Ω
220Ω
390Ω
ID=0
ID=1
RS232
RS485
– + GND
11 12 13
Schield
Lmax=1200m
Schield
Line termination
+5Vdc
390Ω
220Ω
390Ω

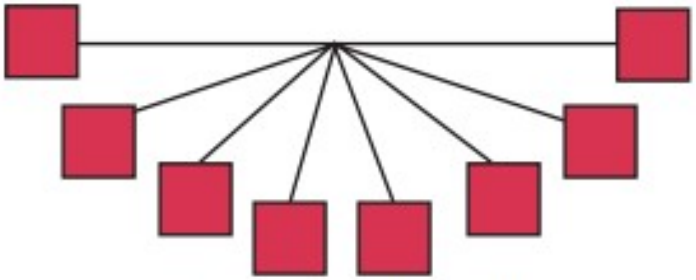
RS485 a topologie magistrali



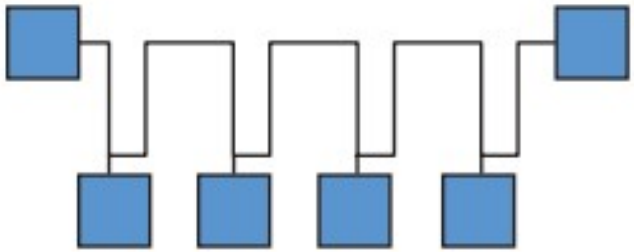
Backbone with stubs (workable)



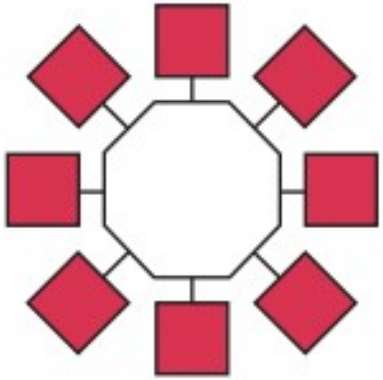
Backbone with stars or clusters (avoid)



Star network (avoid)



Daisy chain (best)



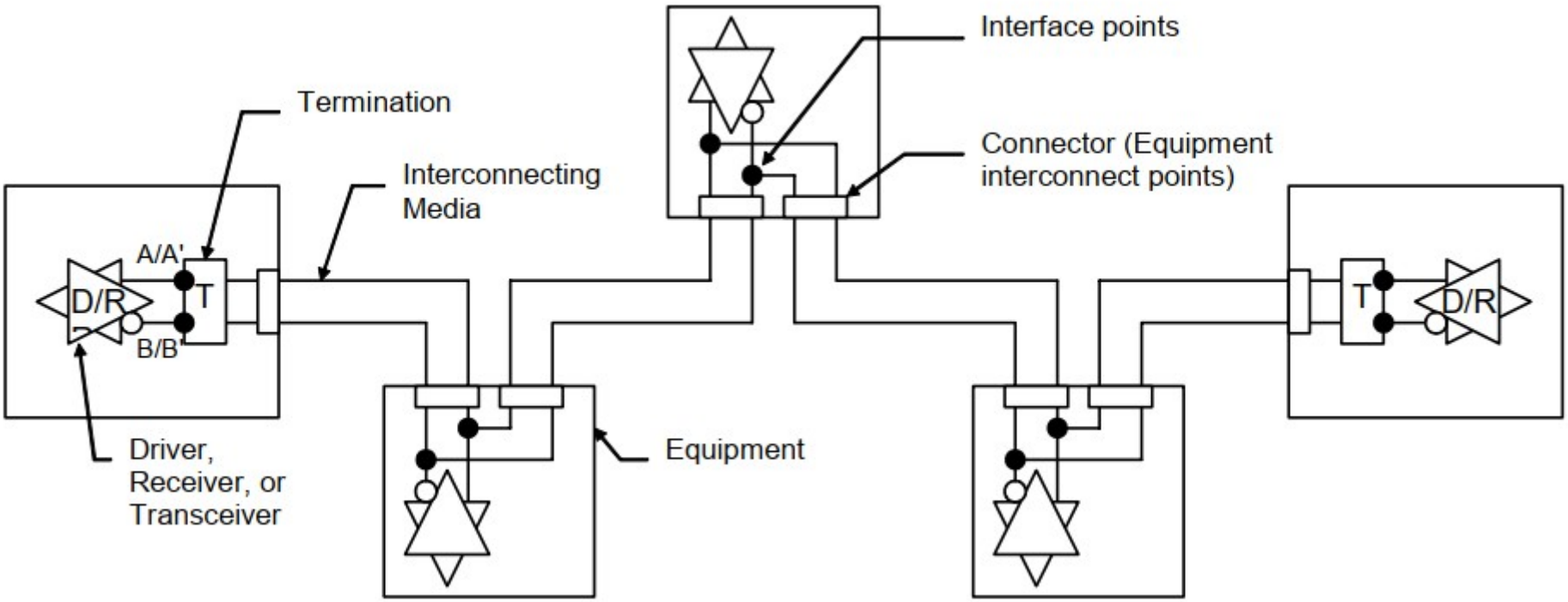
Ring (avoid)

Terminatory 120 Ω

Przy niewielkich odległościach i niewielkich prędkościach transmisji nawet bez R=120 komunikacja zadziała.

Mimo to proszę co do zasady terminować stacje końcowe (TYLKO końcowe!) na magistrali (zworka TERM J26 na BeamKit).

Na laborce będziemy używać celowo przewodu, który się do tego niezbyt nadaje, aby zwiększyć szanse wystąpienia błędów, z którymi powinniście sobie poradzić w dobrze wykonanym kodzie.



[TIA TELECOMMUNICATIONS SYSTEMS BULLETIN]

Inne rodzaje terminacji [https://www.ti.com/lit/an/snla034b/snla034b.pdf]

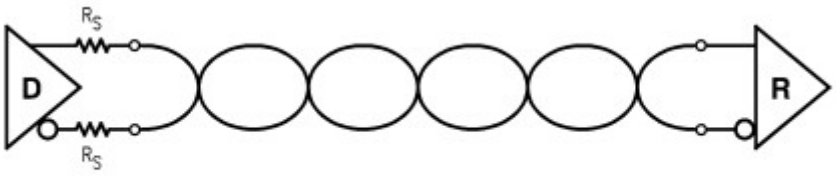


Figure 3. Series Termination Configuration

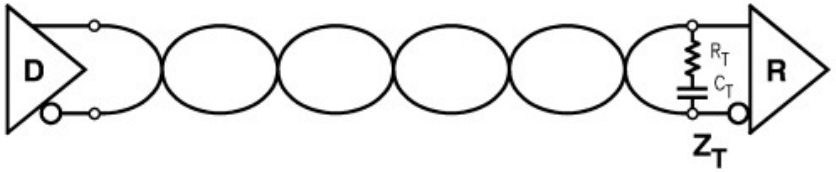


Figure 7. AC Termination Configuration

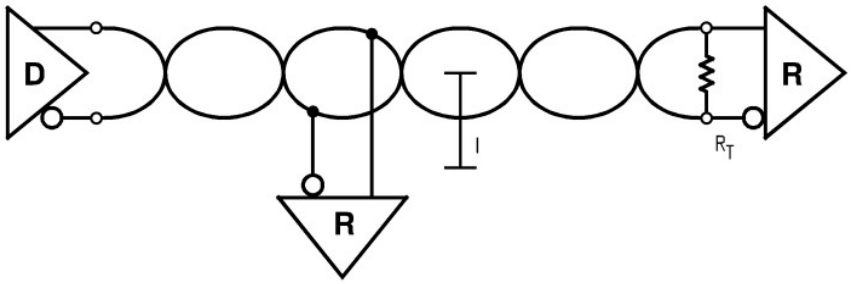


Figure 5. Parallel Termination Configuration

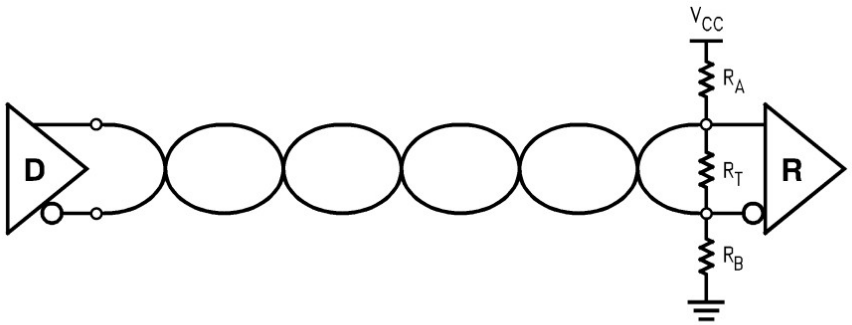
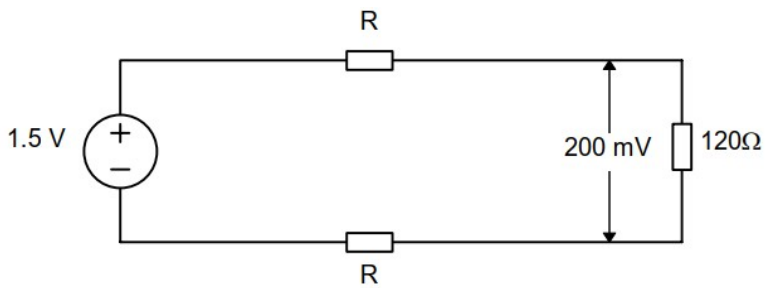


Figure 9. Power Termination Configuration

Ograniczenie zasięgu



[TIA TELECOMMUNICATIONS SYSTEMS BULLETIN]

Poniżej 1 MHz można przyjąć, że dominującym czynnikiem jest rezystancja przewodów. Aby zapewnić minimalną wartość napięcia 200 mV rozpoznawaną przez wejście różnicowe odbiornika, rezystancja linii (typowa skrętka) nie może przekroczyć 390 Ω, co w teorii (brak zakłóceń i negatywnego wpływu innych czynników) daje kilka kilometrów, ale w praktyce są to setki metrów (standard mówi 1200 m).

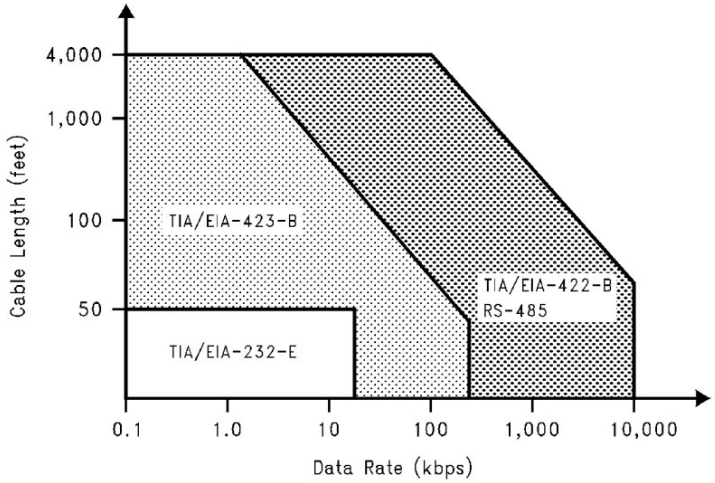
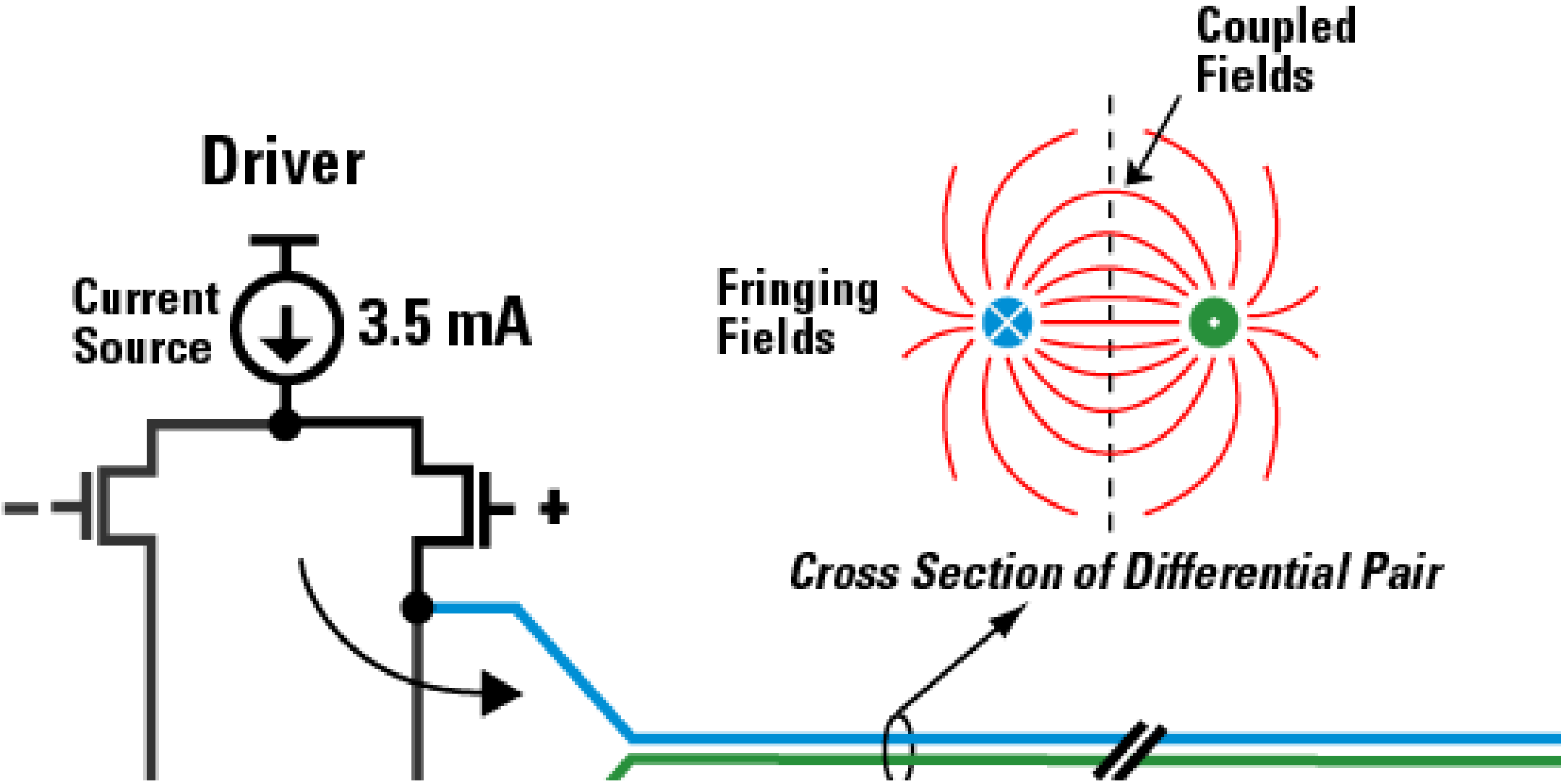
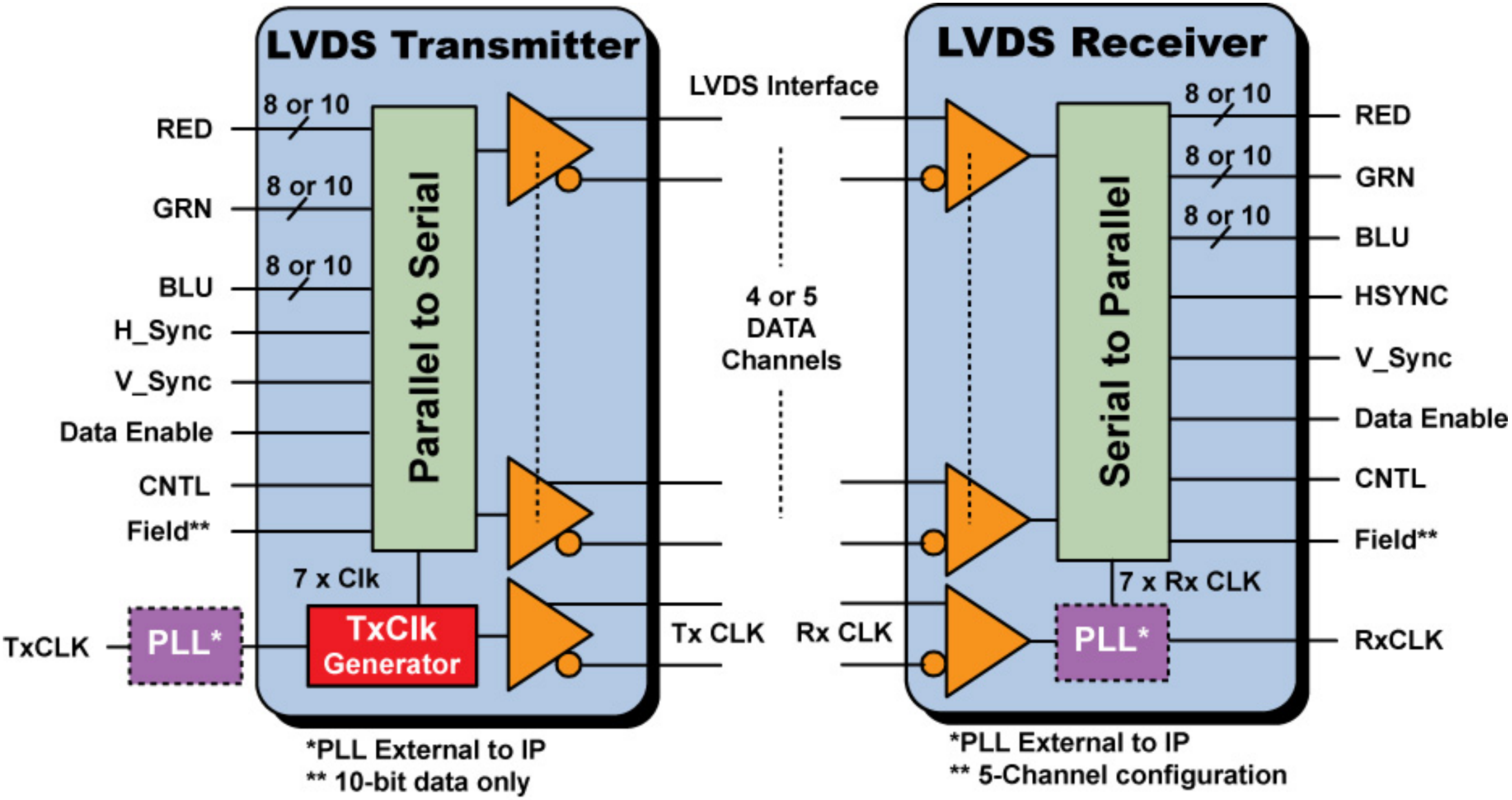


Figure 3. Cable Length Versus Data Rate

Zasada działania linii różnicowej



Podobna zasada działania: LVDS



...lub Ethernet 802.3u

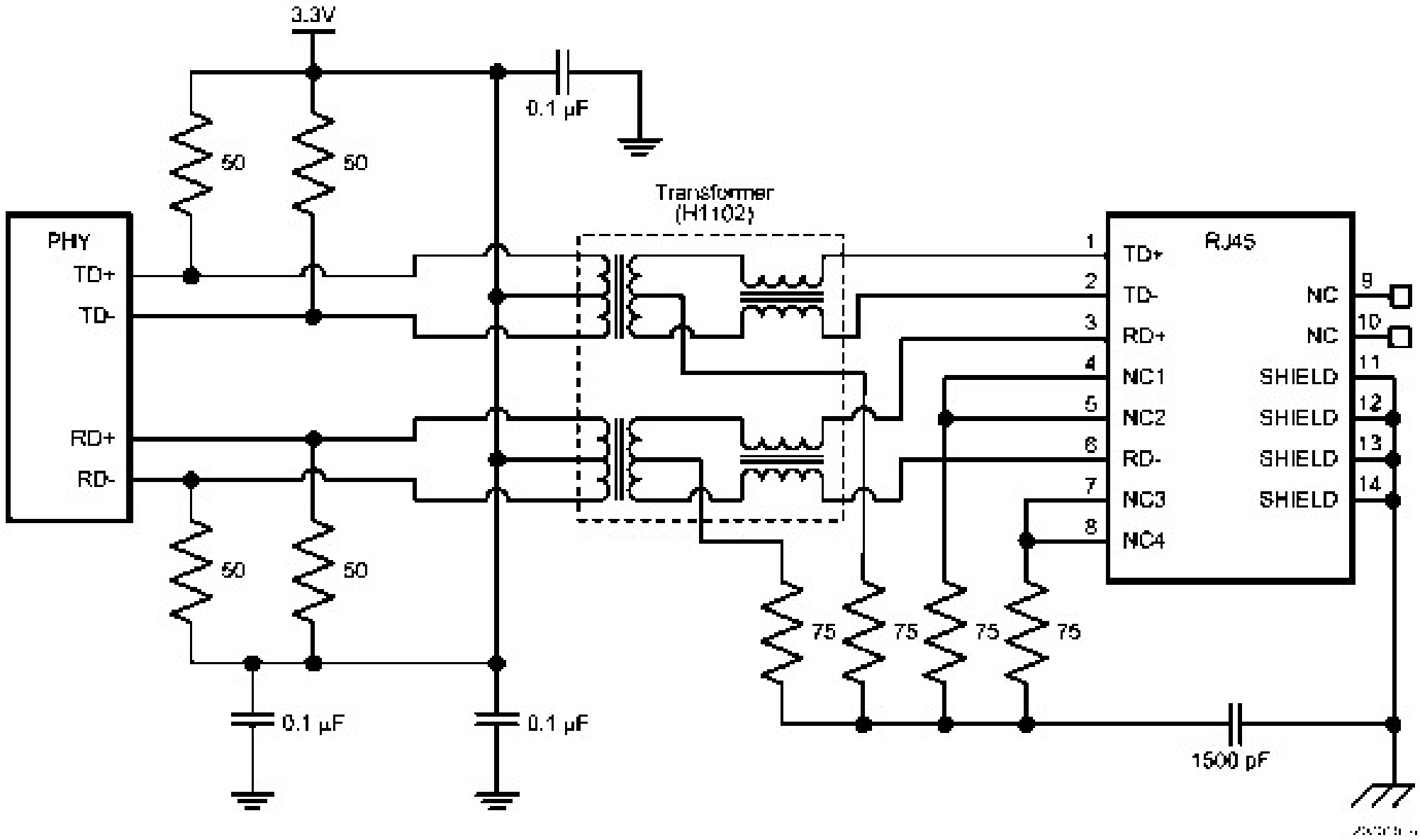


FIGURE 4. Typical 10/100 Mb/s Twisted Pair Interface

Czym są komendy AT?

MCU ← → modemem GSM/GPRS
MCU ← → Bluetooth SPP

→komendy Hayesa, czyli zbiór rozkazów opracowany na potrzeby komunikacji i sterowania urządzeniem *Hayes Smartmodem*.

→Technicznie: zbiór krótkich rozkazów zapisanych w postaci tekstowej, z opcjonalnymi argumentami (też kodowanymi tekstowo).

→AT pochodzi od „attention”, a najprostszą komendą AT jest… „AT”.

→różne znaczenie w różnych urządzeniach

→obecnie dopuszczalne jest używanie wielkich i małych liter

at

OK

AT

OK

Tryb komend a tryb danych

→współdzielenie jednego kanału transmisyjnego (linii TX/RX)

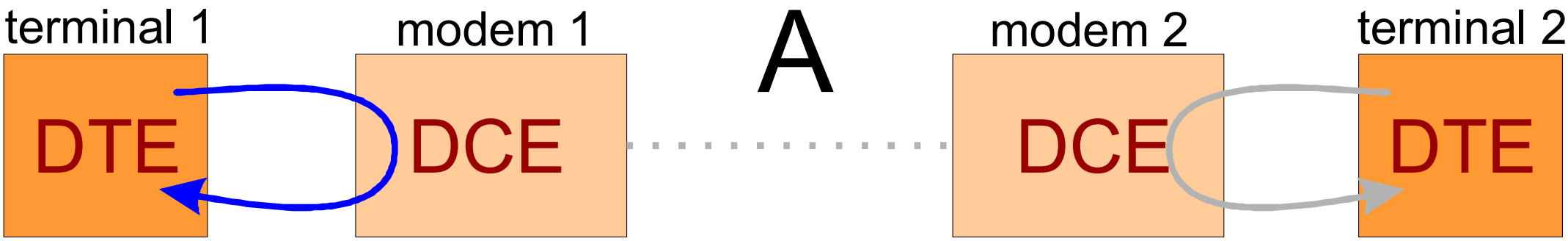
→wychodzenie z trybu danych w trakcie połączenia

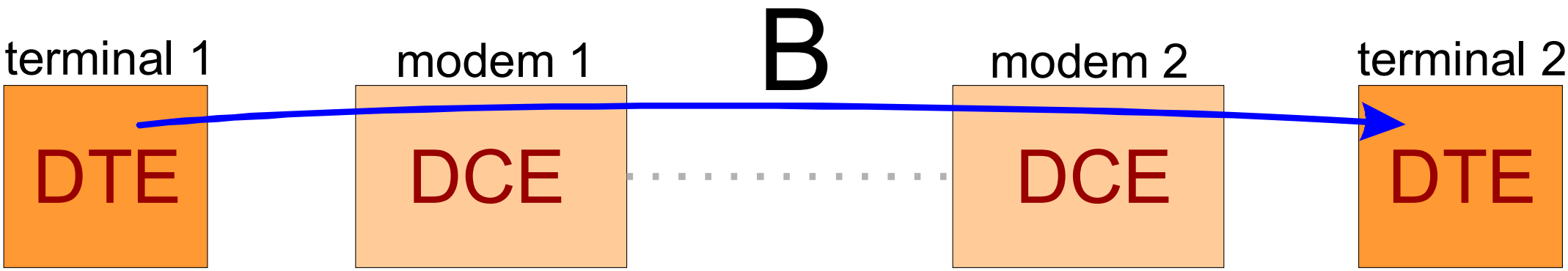
Aby urządzenie mogło rozpoznawać rozkazy AT, musi działać w trybie komend (w odróżnieniu od trybu transmisji danych).

Zarówno do przesyłania rozkazów jak i danych (już po nawiązaniu połączenia) są używane najczęściej te same linie TxD/RxD.

Skąd urządzenie „wie”, czy pojawiające się znaki należy przekazywać dalej (do odbiorcy), czy przetwarzać lokalnie (interpretować jako komendy AT)?

DTE – Data Terminal Equipment (np. terminal znakowy)
DCE – Data Communications Equipment (np. modem telefoniczny)


terminal 1
modem 1
A
modem 2
terminal 2
DTE
DCE
DCE
DTE

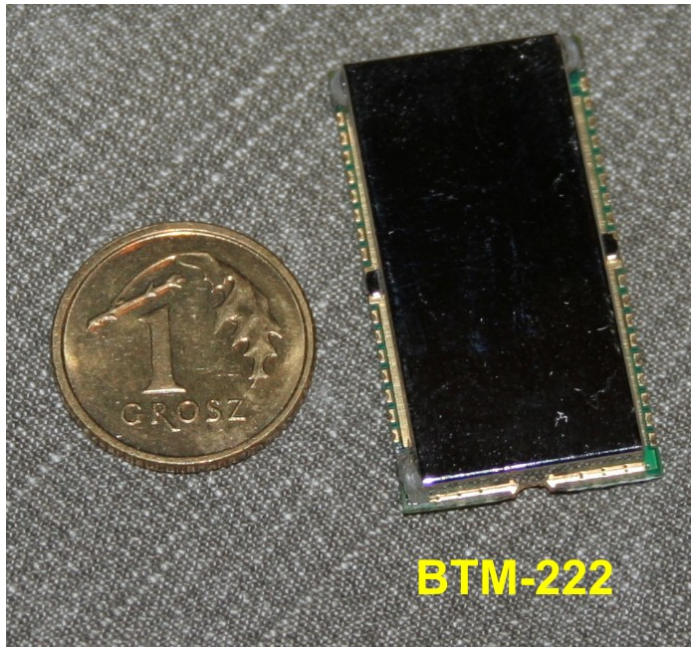

terminal 1
modem 1
B
modem 2
terminal 2
DTE
DCE
DCE
DTE

Typowy scenariusz użycia modemu

1) Włączamy zasilanie (urządzenie jest w trybie komend)
2) Konfigurujemy urządzenie i ustawiamy parametry wysyłając linie tekstu rozpoczynające się od AT, np. „ATN modul_BTM222<CR>”, przy czym <CR> oznacza znak powrotu karetki (ang. *carriage return*), Kodem dziesiętnym tego znaku jest 13 (szesnastkowo 0x0D). Jeśli używamy programu terminal, to jest to po prostu naciśnięcie klawisza Return.
3) Wydajemy rozkaz nawiązania połączenia (ATD) lub czekamy, aż druga strona nawiąże z nami połączenie
4) Pojawia się opcjonalnie informacja „CONNECT”, po czym urządzenie przechodzi w tryb transmisji danych. Od tego momentu każdy wysłany znak jest przesyłany do odbiorcy (tzn. do innego połączonego z nami urządzenia). W tym trybie modem ignoruje wszystkie polecenia (są one traktowane jak dane do wysłania), z jednym wyjątkiem.
5) Wyjście z trybu danych (i powrót do trybu poleceń) odbywa się za pomocą wysłania trzech znaków +++ i odczekania przez okres trwania jednej pauzy (domyślnie jest to 1 sekunda).

Komunikacja z miniaturowym modułem Bluetooth (BTM-222)

→ kompletny układ nadajnika/odbiornika zgodnego ze standardem Bluetooth, wyposażony w profil SPP.



Do uruchomienia układu potrzebne jest napięcie zasilające 3.3 V, antena nadawczo-odbiorcza (w zupełności wystarczy kawałek przewodu długości kilku cm) oraz sygnały TxD/RxD, zgodnie ze standardem UART.

Profil SPP + komendy AT → bezpośrednia (i uproszczona dla programisty) komunikacja mikrokontrolera z urządzeniami takimi jak np. laptopy, komputery stacjonarne, komputery kieszonkowe i inne urządzenia wyposażone w profil SPP.

Przykładowe komendy AT (BTM-222)

Komenda AT	Przykład
ATA1..ATA8 – w trybie Master służy do nawiązania połączenia z jednym z 8 urządzeń wykrytych w otoczeniu interfejsu Bluetooth. Listę urządzeń można uzyskać wydając rozkaz „ATF?”.	**ATA1** **OK** **CONNECT '001E-0B-019A28'**
ATB? – wyświetla adres modułu BT.	**ATB?** **0012-6f-02a493** **OK**
ATD=xxxxxxxxxxxx – dla większego bezpieczeństwa pozwala skonfigurować adres urządzenia, które jako jedyne będzie mogło się połączyć z modułem BTM-222. ATD? – odczytuje w/w adres.	**ATD=001E0B019A28** **OK** **ATD?** **001e-0b-019a28** **OK**
ATE0 – wyłącza ECHO dla rozkazów odbieranych lokalnie (ich kopia nie pojawia się na linii RxD). Wygodne rozwiązanie w przypadku bezpośredniej komunikacji między mikrokontrolerem a modułem (upraszcza odbieranie danych z modułu, nie potrzeba dodatkowo filtrować kopii znaków wysłanych przez mikrokontroler). **ATE1** – włącza ECHO dla rozkazów wysyłanych na linię TxD, znaki są kopiowane na linię RxD. Bardzo	**ATE?** **1** **OK**

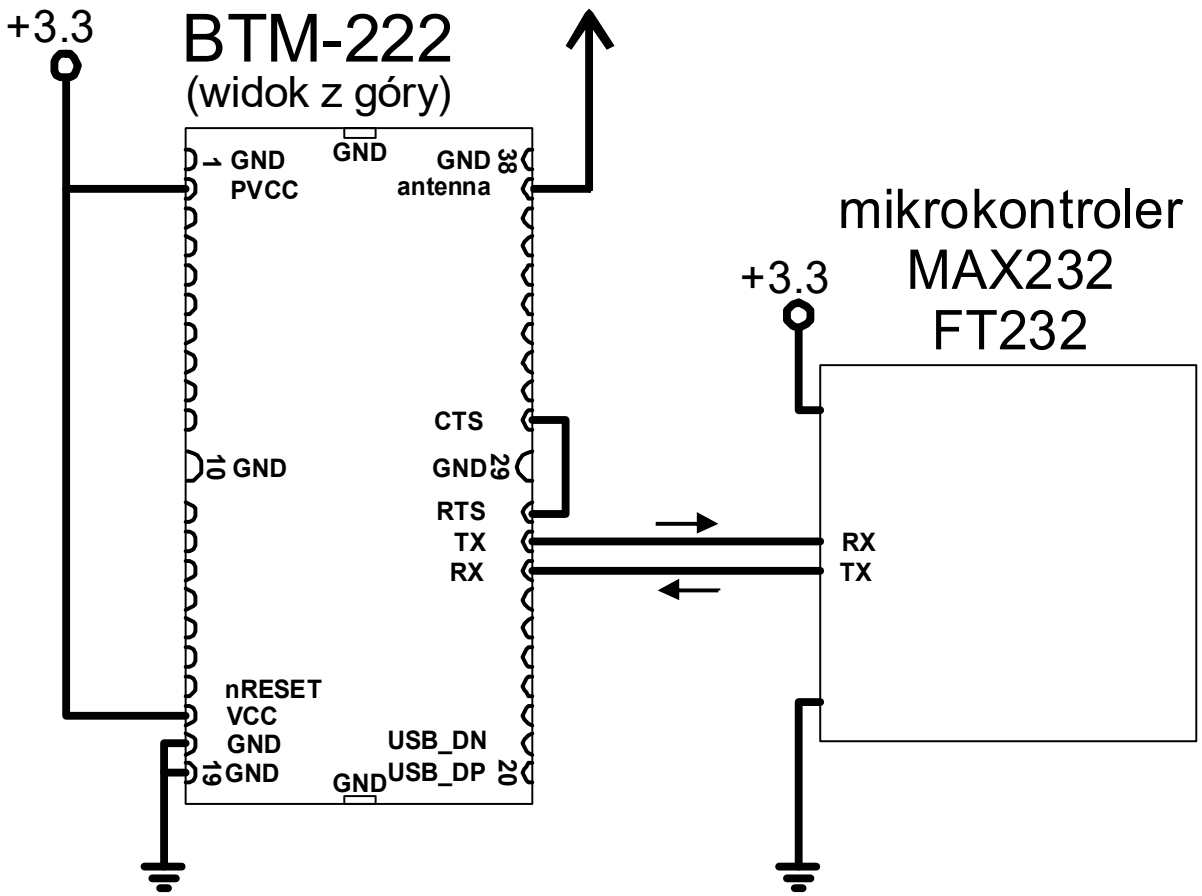
wygodne wtedy, gdy bezpośrednio z okienka terminala komunikujemy się z urządzeniem.	
ATF? – w trybie Master listuje urządzenia w otoczeniu Bluetooth. Skanowanie trwa 60 sekund. Pozycja na liście (1..8) jest używana jako argument komendy ATA, służącej do nawiązania połączenia.	**ATF?** **OK** **Inquiry Results:** **1 WM_root 001E-0B-019A28** **2 Nokia 001E-A3-E11481** **3 0021-D2-39678E** **Inquiry End. 3 Device(s) Found.**
ATH1 – urządzenie jest wykrywalne **ATH0** – urządzenie jest ukryte Nawet w przypadku ukrycia urządzenia możliwe jest nawiązanie z nim połączenia, o ile wcześniej miało miejsce sparowanie urządzeń.	**ATH?** **0** **OK**
ATI? – wyświetla wersję firmware'u	**ATI?** **F/W VERSION: v4.22**
ATK0 – jeden bit stopu **ATK1** – dwa bity stopu	**ATK?** **0** **OK** **0 (1 bit stopu) jest domyślną wartością.**
ATL0 – 4800bps **ATL1** – 9600bps **ATL2** – 19200bps	**ATL?** **2**

ATL3 – 38400bps **ATL4** – 57600bps **ATL5** – 115200bps **ATL6** – 230.4Kbps **ATL7** – 460.8Kbps	**OK** **2 (19200 bps) jest domyślną wartością.**
ATM0 – parzystość „None” **ATM1** – parzystość „Odd” **ATM2** – parzystość „Even”	**ATM?** **0** **OK** **0 (brak kontroli parzystości) jest domyślną wartością.**
ATN=nazwa Nadaje nazwę urządzeniu	**ATN=M52233DEMO** **OK** **ATN?** **M52233DEMO** **OK** **Domyślne ustawienie: „Serial Adaptor”**
ATO0 – automatyczne nawiązywanie połączenia z dostępnym urządzeniem (lub tym, które określono w parametrze wywołania ATD) **ATO1** – użytkownik ręcznie nawiązuje połączenie używając komendy ATA	**ATO?** **0** **OK**

ATP=xxxx – nadaje numer PIN **ATP?** – wyświetla aktualnie obowiązujący PIN	**ATP=1234** **OK** **ATP?** **1234** **OK**
ATQ0 – komunikaty będą wysyłane interfejsem UART. Dotyczy to komunikatów OK, CONNECT, DISCONNECT, ERROR **ATQ1** – komunikaty nie będą wysyłane. Może to ułatwić obsługę modułu z poziomu mikrokontrolera, ponieważ zwalnia programistę z konieczności analizowania tych odpowiedzi. Należy jednak wtedy użyć innych metod sprawdzania statusu wykonania komendy lub połączenia/rozłączenia.	**ATQ?** **0** **OK**
ATR0 – urządzenie pełni rolę nadrzędną (Master) **ATR1** – urządzenie pełni rolę podrzędną (Slave)	**ATR?** **1** **OK** **Domyślne ustawienie: 1 (Slave)**
ATU – umożliwia załadowanie nowszej wersji firmware'u za pośrednictwem portu USB (wymaga wyprowadzenia odpowiednich sygnałów z modułu)	
ATZ0 – przywraca ustawienia domyślne (fabryczne), w tym konfigurację portu szeregowego. Po wykonaniu tego polecenia należy w mikrokontrolerze przywrócić ustawienia UART	

19200,8,N,1 aby była możliwa dalsza komunikacja z modułem.	

Łączymy mikrokontroler i moduł BTM-222

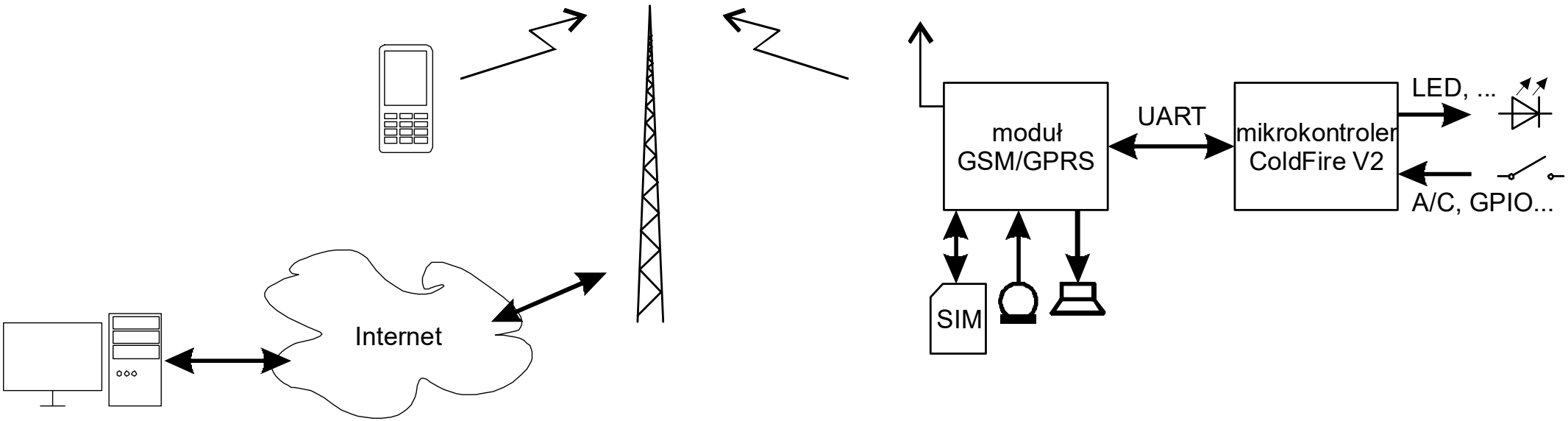


→najmniejsza możliwa liczba połączeń

→brak sterowania przepływem (lokalnie zwarte sygnały RTS/CTS)

→brak sterowania linią RESET

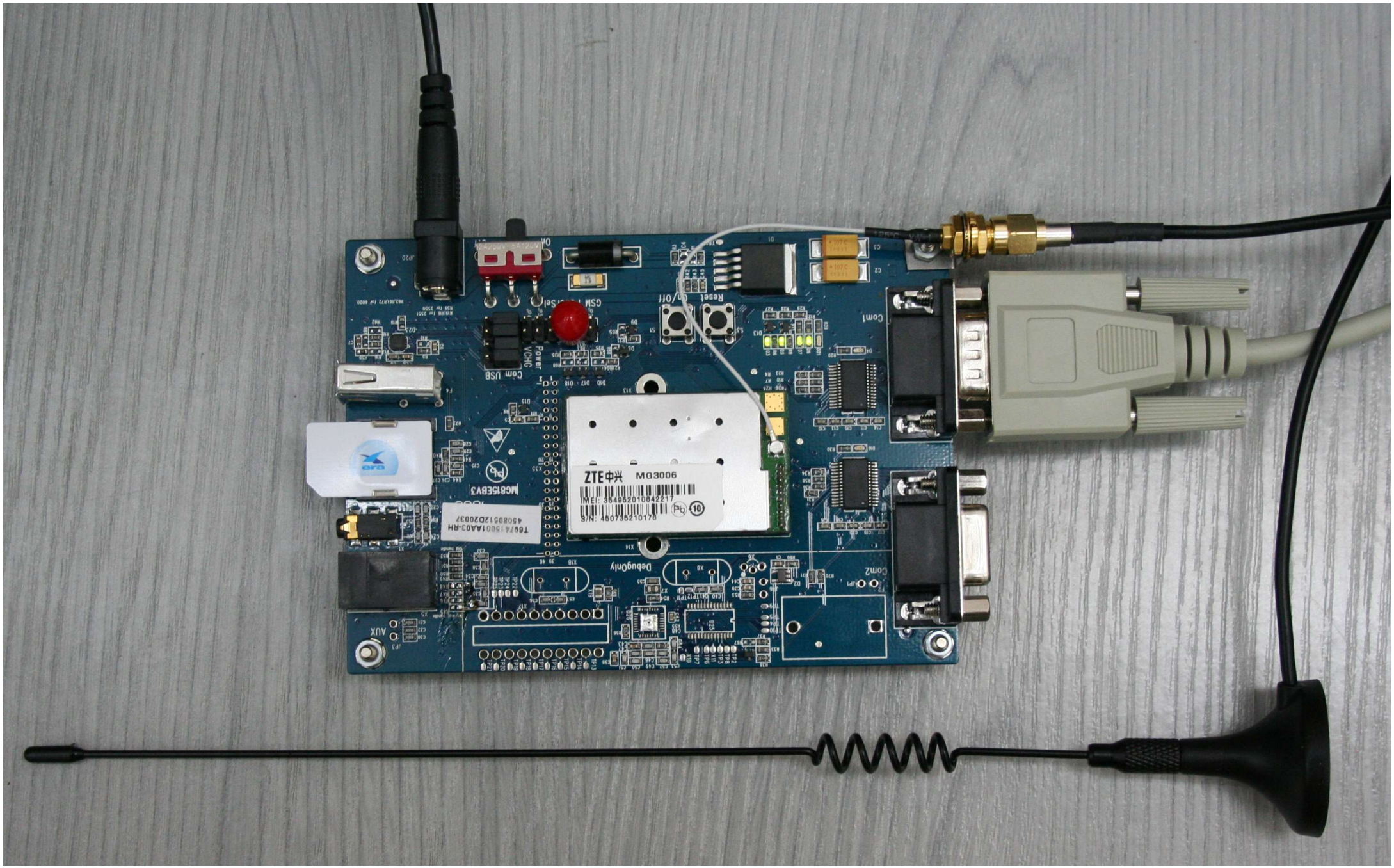
Poszerzamy horyzonty: komunikacja GSM/GPRS



→niewielkim kosztem można połączyć MCU z siecią GSM.

→w świecie mikrokontrolerów: głównie telemetria, zdalne monitorowanie, sterowanie, systemy alarmowe, śledzenie pojazdów…

→w ofercie operatorów m.in. taryfy telemetryczne



Zabytkowy zestaw, zwróć uwagę na 2 porty RS232

Podstawowe komendy związane z GSM

Wprowadzanie numeru PIN

```
AT+CPIN="1234"
+CPIN: READY
OK
```

Wysyłanie SMS

```
AT+CMGF=1
AT+CMGS="661280***",129
> QWERTY
>
+CMGS: 38
```

Odczyt SMS

```
AT+CMGL="ALL"
+CMGL: 1,"REC READ","+48509396***",,"08/07/09,22:49:59+08"
Test Twojego modulu
```

Nawiązanie, a następnie przerwanie połączenia głosowego

```
ATD+48661280***;
ATH
```

Książka telefoniczna (SIM)

```
AT+CPBR=1,9
```

+CPBR: 1,"+48602963000",145,"Saldo konta"
+CPBR: 2,"+48602909000",145,"TeleEra"
+CPBR: 3,"+48602900000",145,"Biuro Obslugi"
+CPBR: 4,"+48602950000",145,"Poczta glosowa"
+CPBR: 5,"*42913",129,"Biuro numerow"
+CPBR: 6,"+48602992000",145,"Era Premia"
+CPBR: 7,"*7100",129,"Swiat rozrywki"
+CPBR: 8,"+48602990000",145,"Informacje"

IMEI

at+cgsn
354952010642217
OK

Jakość sygnału radiowego

at+csq
+CSQ: 12,0
+CSQ: 31,0
+CSQ: 29,0

Podstawowe komendy związane z GPRS

Transmisja danych przez UDP (wersja skrócona, bez odpowiedzi)

at+zpnum="erainternet","erainternet","erainternet" //AP, user, pass

at+zpppopen //logowanie do usługi GPRS

at+zipsetupu=1,156.17.46.4,6080 //utwórz gniazdo UDP nr 1 dla adresu 156.17.46.4:6080

at+zipsendu=1,10 //do tego gniazda wyślij 10 B danych

>test udp 1 //wysyłane dane

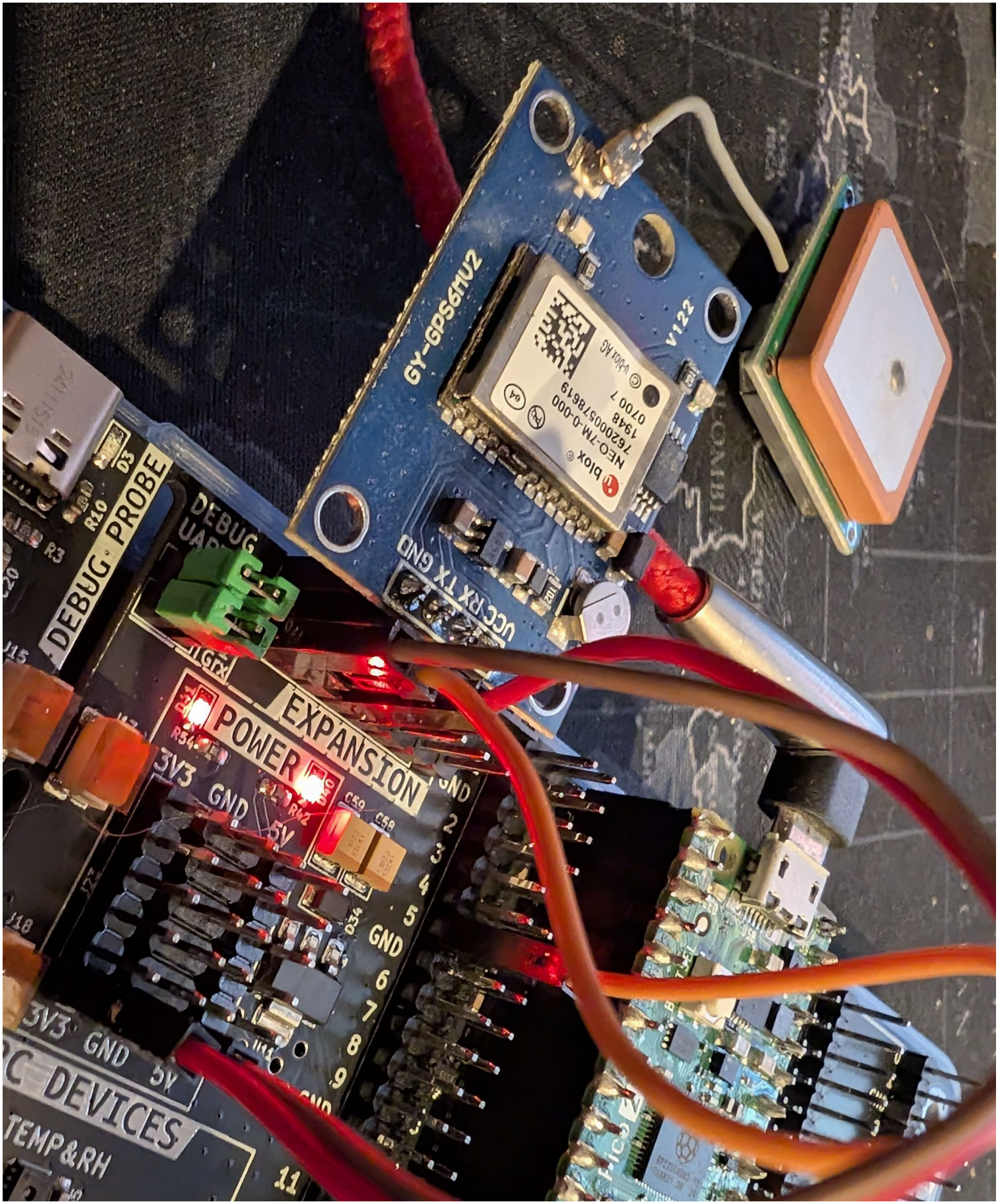
+ZIPSENDU:OK

at+zipcloseu=1 //zamykanie gniazda

+ZIPCLOSEU:OK

Z poziomu aplikacji konsolowej (serwer UDP, ten protokół będzie na dalszym wykładzie) otrzymamy:

```
D:\PRACA\RCP\udp_rx.exe
Oczekiwanie na pakiety UDP...
74 65 73 74 20 75 64 70 20 31
test udp 1
```


GY-GPS6MU2
V1.2.2
u-blox
NEO-7M-0-000
DEBUG PROBE
DEBUG UART
VCC RX TX GND
EXPANSION
POWER
3V3
GND
5V
GND
TEMP&RH

GPS (NMEA)

Kod w MicroPython:

```
from machine import UART, Pin
gps = UART(1, rx=Pin(5))
gps.init(baudrate=9600, bits=8, parity=None, stop=1)
while True:
    while gps.any(): print(gps.read())
```

Przykładowe zdanie: https://aprs.gids.nl/nmea/

```
$GPRMC,hhmmss.ss,A,llll.ll,a,yyyyy.yy,a,x.x,x.x,ddmmyy,x.x,a*hh
1    = UTC of position fix
2    = Data status (V=navigation receiver warning)
3    = Latitude of fix
4    = N or S
5    = Longitude of fix
6    = E or W
7    = Speed over ground in knots
8    = Track made good in degrees True
9    = UT date
10   = Magnetic variation degrees (Easterly var. subtracts from true course)
11   = E or W
12   = Checksum
```

Bluetooth+SPP i przykład protokołu o strukturze bitowej (nie tekstowej)

Transmisja danych przy użyciu profilu SPP to nie tylko protokół tekstowy. Przyjrzyjmy się komercyjnemu protokołowi bitowemu, który został użyty w zestawie Lego Mindstorms NXT.

Przykładowa komenda – wygenerowanie dźwięku (rozkaz 0x03) – wygląda następująco:

[PLEN(LSB)] [PLEN(MSB)] [0x80] [CMD] [frequency(LSB)] [frequency(MSB)] [duration(LSB)] [duration(MSB)]

- *PLEN* to długość pakietu w bajtach (liczba 16-bitowa),
- *CMD* to kod rozkazu (0x03),
- *frequency* to częstotliwość generowanego dźwięku w Hz (16 bitów)
- *duration* to czas jego trwania w milisekundach (16 bitów)

Aby wygenerować dźwięk o częstotliwości 1000 Hz i czasie trwania 300 ms należy wysłać następującą porcję danych:

0x05 0x00 0x80 0x03 0xE8 0x03 0x2C 0x01

Zalety tego rozwiązania to przede wszystkim:

- oszczędność przepustowości kanału (brak narzutu wynikającego z serializacji)
- z góry określona struktura pakietu (łatwy w dekodowaniu)
- duża szybkość działania kodera i dekodera
- możliwość dodania pola z kodem kontrolnym (detekcja błędów)
- wygodna implementacja dzięki SPP (brak sterowników, zgodność z większością platform)

Wady:

- długotrwałe negocjowanie połączenia
- często problemy po utracie łączności
- duże narzuty niższych warstw modelu odniesienia OSI
- niewielka sprawność (uzyskany zasięg/zużywana moc)
- bardzo duże opóźnienia, praktycznie wykluczające pracę w czasie rzeczywistym

Inne rozwiązanie to sprzętowe przerwania UART-RX. Gdy wykryty zostanie odebrany znak, moduł UART zgłosi przerwanie, co spowoduje tymczasowe wstrzymanie działania bieżącego kodu wykonywanego przez MCU i przejście do funkcji ISR która zajmie się odebranym znakiem.

Ten przykład był przygotowany pod RS485, ale usunąłem fragmenty specyficzne dla RS485 (kod działa jako UART lub RS232 null modem).

```
void led_toggle() {
   static int toggle = 0;
   cyw43_arch_gpio_put(CYW43_WL_GPIO_LED_PIN, toggle);
   toggle = !toggle;
}
```

Wskazówka: static przed definicją zmiennej lokalnej!

```
#define N_TERM_BUF 32 // Intentionally that short
char term_buf[N_TERM_BUF]; // Here we collect the received data
uint32_t term_idx = 0; // term_buf cursor - current position

void process_command() {
/// tutaj wasze zadanie - dekodowanie zawartości term_buf
}

void rs485_send_str(const char *src) {
   size_t len = strlen(src);
   uart_write_blocking(RS485_UART_ID, (const uint8_t *)src, len);
   uart_tx_wait_blocking(RS485_UART_ID); //wait for fifo empty
}
```

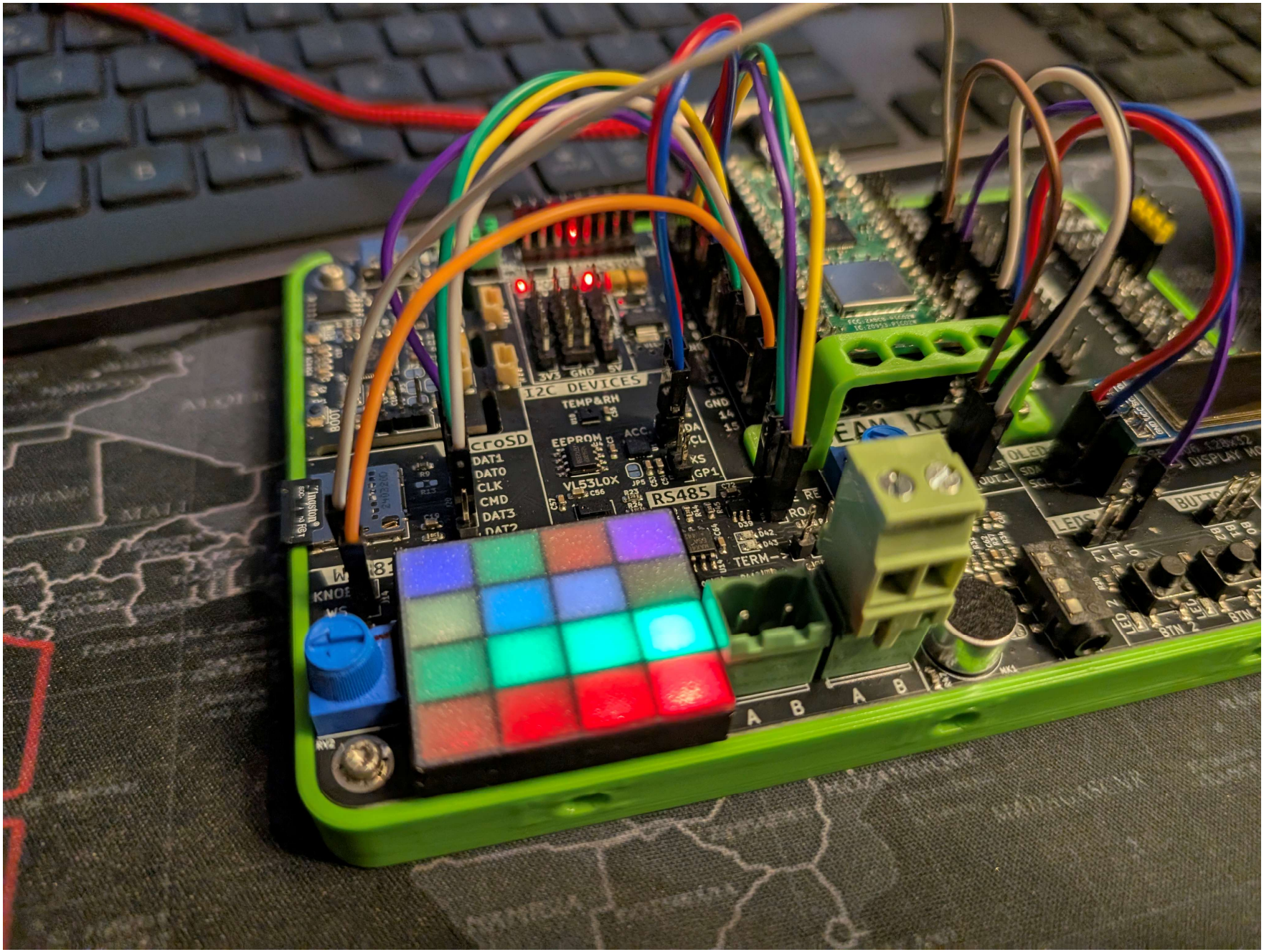
```
void on_uart_rx() { // RX interrupt handler
   while (uart_is_readable(RS485_UART_ID)) {
      uint8_t ch = uart_getc(RS485_UART_ID);
      if(ch == '\r') {
         process_command();
         term_idx = 0; // Prepare for another command reception
         term_buf[term_idx] = '\0';  // even empty C string must end with zero (\0)
      } else {
         if(term_idx<N_TERM_BUF-1) {
            term_buf[term_idx++] = ch;
            term_buf[term_idx] = '\0'; // ASCIIZ - C string must end with zero (\0)
         } else {
            ch = '~'; // Notify user: term buffer is full
         }
         if (term_echo && uart_is_writable(RS485_UART_ID)) {
            uart_write_blocking(RS485_UART_ID, (const uint8_t *) &ch, 1);
            uart_tx_wait_blocking(RS485_UART_ID); //wait fifo empty
         }
      }   }   }
```

```
// Set up UART
void rs485_init(uart_inst_t *uart, uint baudrate,
uint tx_pin, uint rx_pin,
uint data_bits, uint stop_bits, uart_parity_t parity) {
    uart_init(uart, baudrate);
    gpio_set_function(tx_pin, GPIO_FUNC_UART);
    gpio_set_function(rx_pin, GPIO_FUNC_UART);
    uart_set_hw_flow(uart, false, false); // disable flow control CTS/RTS
    uart_set_format(uart, data_bits, stop_bits, parity); // Set our data format
    uart_set_fifo_enabled(uart, false); // Turn off FIFO's - we want char by char
    int UARTx_IRQ = uart == uart0 ? UART0_IRQ : UART1_IRQ; // Select interrupt
    irq_set_exclusive_handler(UARTx_IRQ, on_uart_rx); // enable interrupt handlers
    irq_set_enabled(UARTx_IRQ, true);
    uart_set_irq_enables(uart, true, false); // enable the UART RX interrupts
}
```

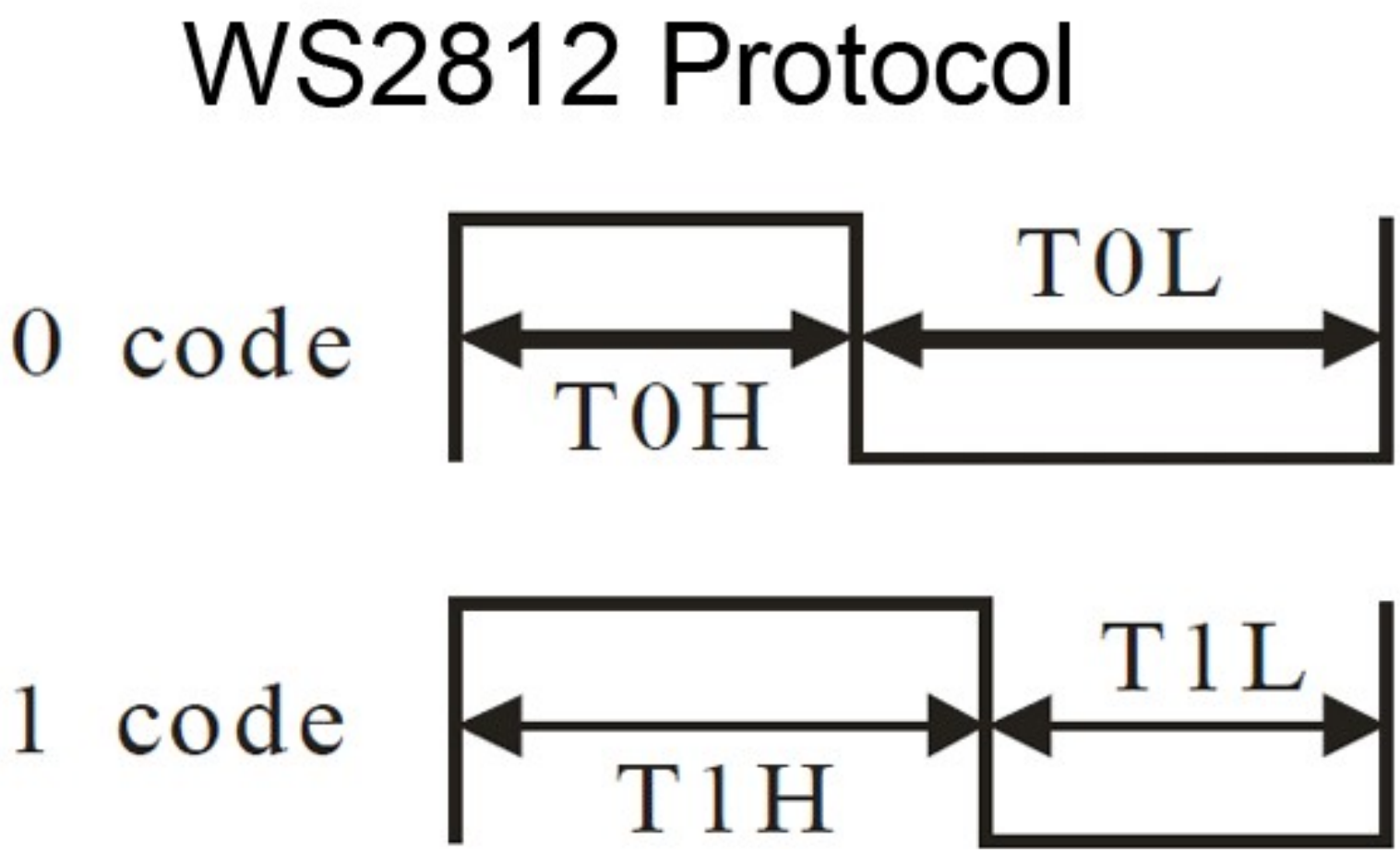
```
int main() {
    stdio_init_all();
    if (cyw43_arch_init()) { // Initialise the Wi-Fi chip
        printf("Wi-Fi init failed\n");
        return -1;
    }
    rs485_init(RS485_UART_ID, RS485_BAUD_RATE,
               RS485_TX_PIN, RS485_RX_PIN,
               RS485_DATA_BITS, RS485_STOP_BITS, RS485_PARITY);
    int32_t cnt = 0;
    while (true) {
        static char buf[100];
        sprintf(buf, "%d Hi, embedded!\r\n", cnt++);
        rs485_send_str(buf);
        sleep_ms(1000);
    }
}
```

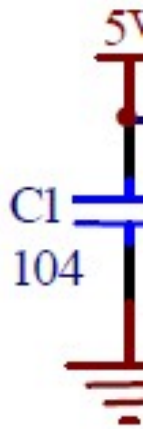
W main nie ma śladu odbierania znaków, jest tylko pętla while wysyłająca co sekundę napis “Hello, embedded!” poprzedzony wartością licznika.

Dancefloor WS2812B



Też asynchroniczny i szeregowy – ale inaczej





→patrz: datasheet

Nie będzie łatwo, ale...

```
from machine import Pin
import rp2, array

NUM_LEDS = const(16)
PIN_NUM = const(15)

# dane pikseli po 32 bity (używane 24:GRB) na każdy LED
ar = array.array("I", [0 for _ in range(NUM_LEDS)])

# PIO w wersji zanegowanej
@rp2.asm_pio(sideset_init=rp2.PIO.OUT_HIGH,
out_shiftdir=rp2.PIO.SHIFT_LEFT, autopull=True, pull_thresh=24)
def ws2812():
    T1 = 2
    T2 = 5
    T3 = 3
    wrap_target()
    label("bitloop")
    out(x, 1)                .side(1)    [T3 - 1]
    jmp(not_x, "do_zero")    .side(0)    [T1 - 1]
    jmp("bitloop")           .side(0)    [T2 - 1]
    label("do_zero")
    nop()                    .side(1)    [T2 - 1]
    wrap()
```

```
# włączenie maszyny PIO
sm = rp2.StateMachine(0, ws2812, freq=8_000_000,
sideset_base=Pin(PIN_NUM))
sm.active(1)

# pompowanie danych z podmianą RGB na GRB
def ws_show():
    dimmer_ar = array.array("I", [0 for _ in range(NUM_LEDS)])
    for i,c in enumerate(ar):
        r = int(((c >> 8) & 0xFF))
        g = int(((c >> 16) & 0xFF))
        b = int((c & 0xFF))
        dimmer_ar[i] = (g<<16) + (r<<8) + b
    sm.put(dimmer_ar, 8) ← tutaj rozpoczęcie transmisji

# ustaw kolor jednego piksela
def ws_set(i, color):
    ar[i] = (color[1]<<16) + (color[0]<<8) + color[2]

# ustaw kolor wszystkich pikseli
def ws_fill(color):
    for i in range(len(ar)):
        ws_set(i, color)
```