Dariusz Makowski

Katedra Mikroelektroniki i Technik Informatycznych

tel. 631 2720

dmakow@dmcs.pl

http://neo.dmcs.p.lodz.pl/sw

# Zakres przedmiotu

- Systemy mikroprocesorowe, systemy wbudowane
- Laboratorium
- Rodzina procesorów ARM
- Urządzenia peryferyjne
- Interfejsy w systemach wbudowanych
- Pamięci i dekodery adresowe
- Programy wbudowane na przykładzie procesorów ARM
- Metodyki projektowania systemów wbudowanych
- Systemy czasu rzeczywistego

# Interfejsy w systemach wbudowanych

# Definicje podstawowe (3)

- **Pamięć komputerowa (ang. Computer Memory)**
  Pamięć komputerowa to urządzenie elektroniczne lub mechaniczne służące do przechowywania danych i programów (systemu operacyjnego oraz aplikacji).
- **Urządzenia zewnętrzne, peryferyjne (ang. Peripheral Device)**
  Urządzenia elektroniczne dołączone do procesora przez magistrale systemową lub interfejs. Urządzenia zewnętrzne wykorzystywane są do realizowania specjalizowanej funkcjonalności systemu.
- **Magistrala (ang. bus)**
  Połączenie elektryczne umożliwiające przesyłanie danym pomiędzy procesorem, pamięcią i urządzeniami peryferyjnymi. Magistra systemowa zbudowane jest zwykłe z kilkudziesięciu połączeń elektrycznych (ang. Parallel Bus) lub szeregowego połączenia (ang. Serial Bus).
- **Interface (ang. Interface)**
  Urządzenie elektroniczne lub optyczne pozwalające na komunikację między dwoma innymi urządzeniami, których bezpośrednio nie da się ze sobą połączyć.

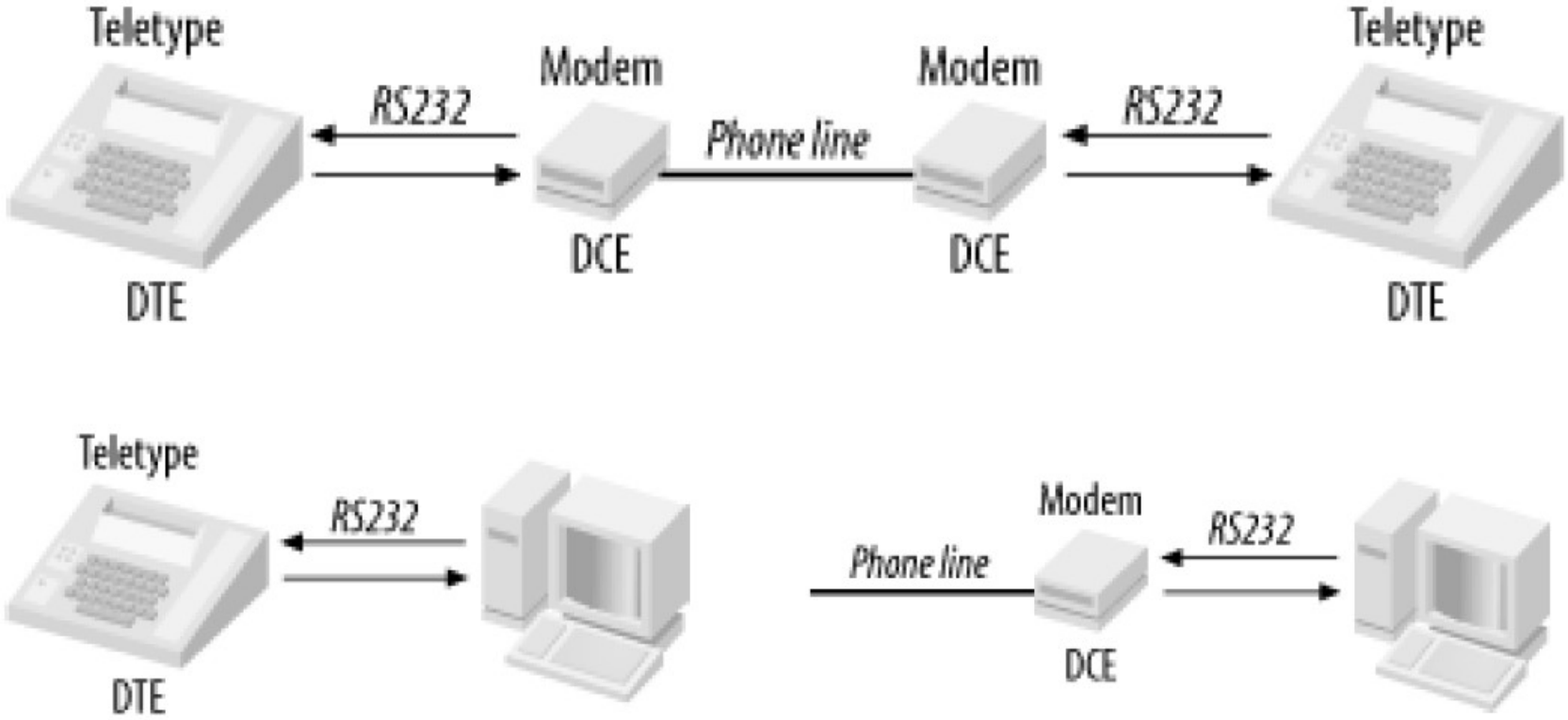
# Współpraca procesora z urządzeniami peryferyjnymi

**Interfejsy dostępne w procesorach rodziny ARM:**

- Interfejs równoległy PIO (zwykle 32 bity),
- Interfejsy szeregowe:
  - Interfejs DBGU - zgodny ze standardem EIA RS232,
  - Interfejs uniwersalny USART,
  - Interfejs Serial Peripheral Interface (SPI),
  - Interfejs Synchronous Serial Controller (SSC)
  - Interfejs I2C, Interfajs Two-wire Interface (TWI),
  - Interfejs Controlled Area Network (CAN),
  - Interfejs Universal Serial Bus (USB),
  - Interfejs Ethernet 10/100.

# Moduł transceivera szeregowego UART (Universal Asynchronous Receiver/Transmitter module)

# Interfejs szeregowy EIA RS232



# Transceiver UART

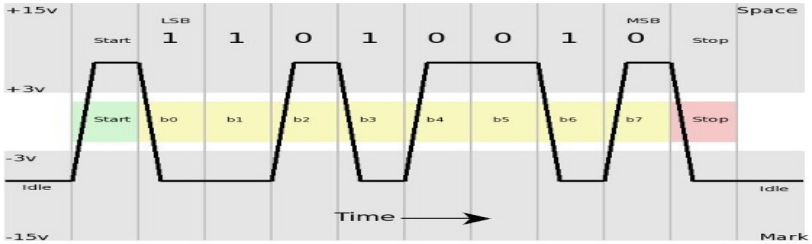
## Rejestr przesuwny

D0-D7

**Nadajnik**

Clk

**TxD**




D0-D7

**Odbiornik**

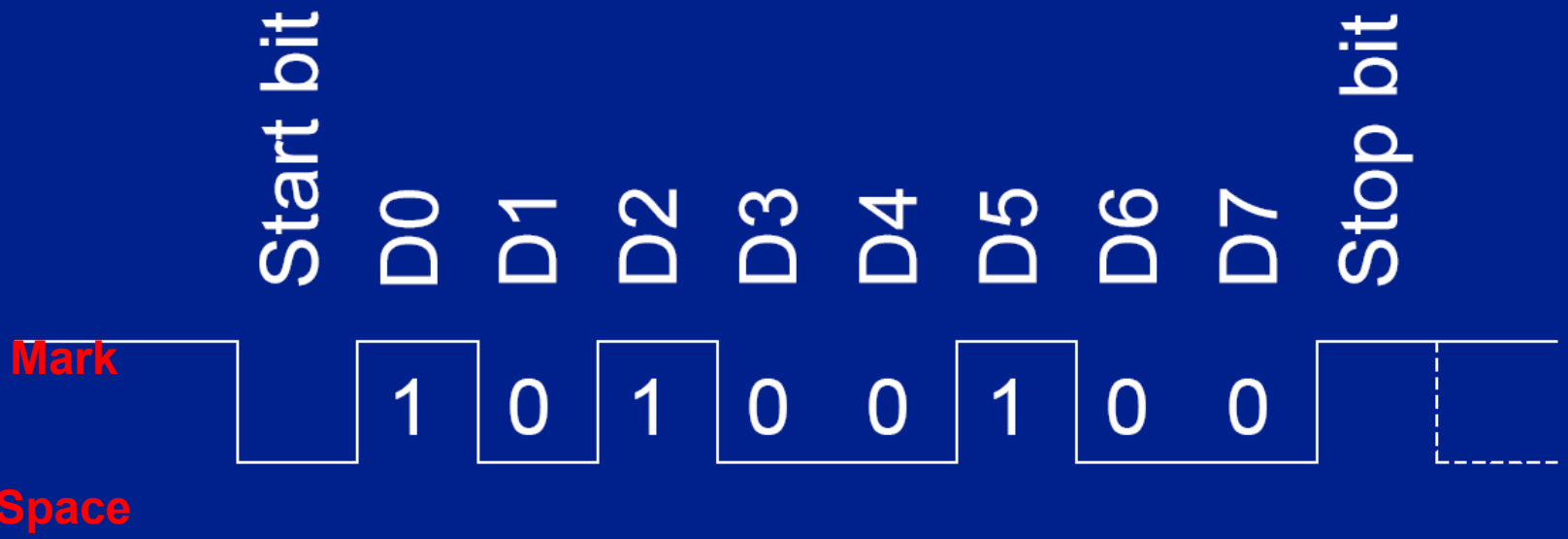
Clk

**RxD**

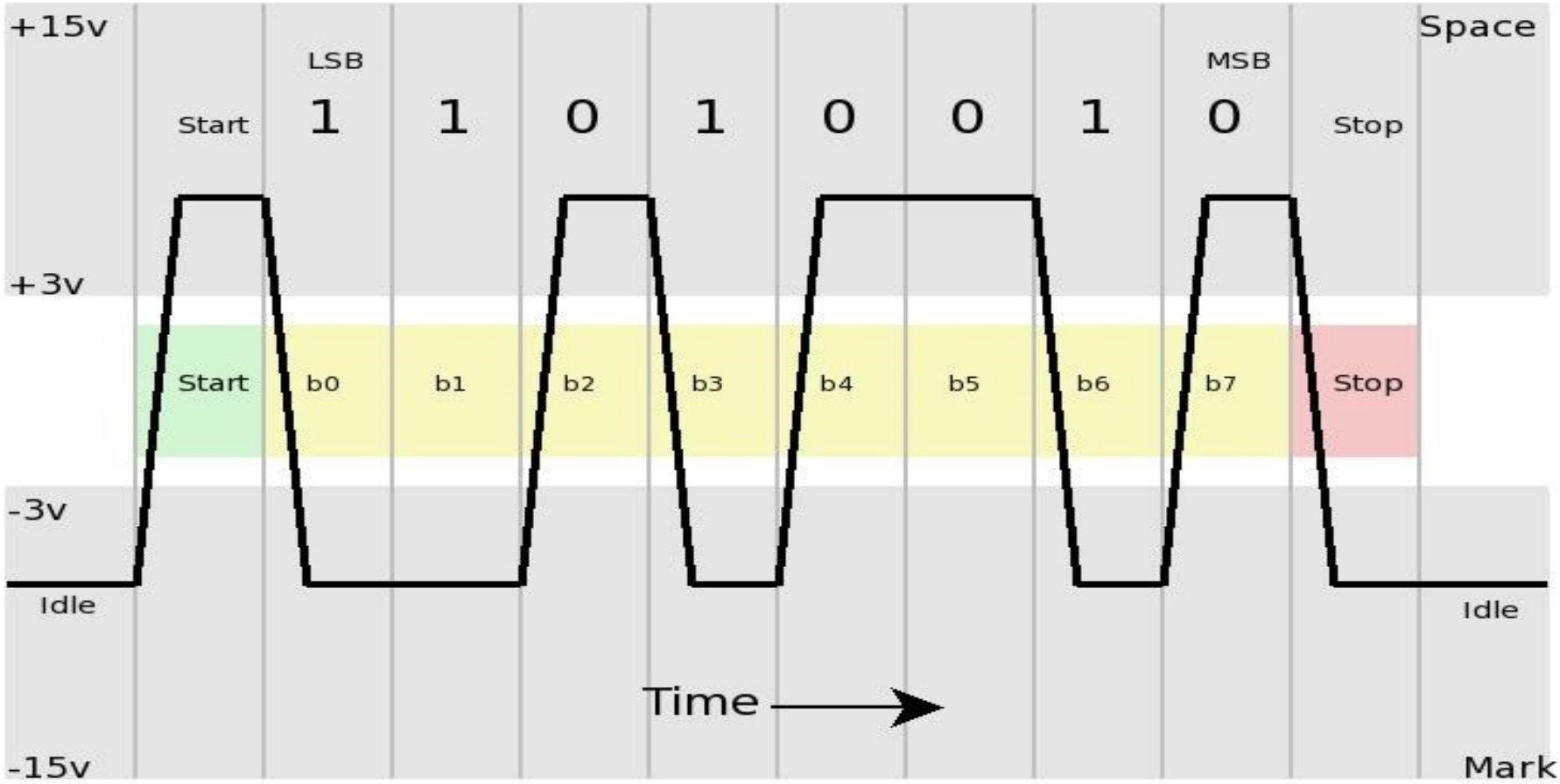
# Ramka danych transmitera UART (1)

## Asynchronous 8 bit waveform example

- Data is H'25' = B'00100101'

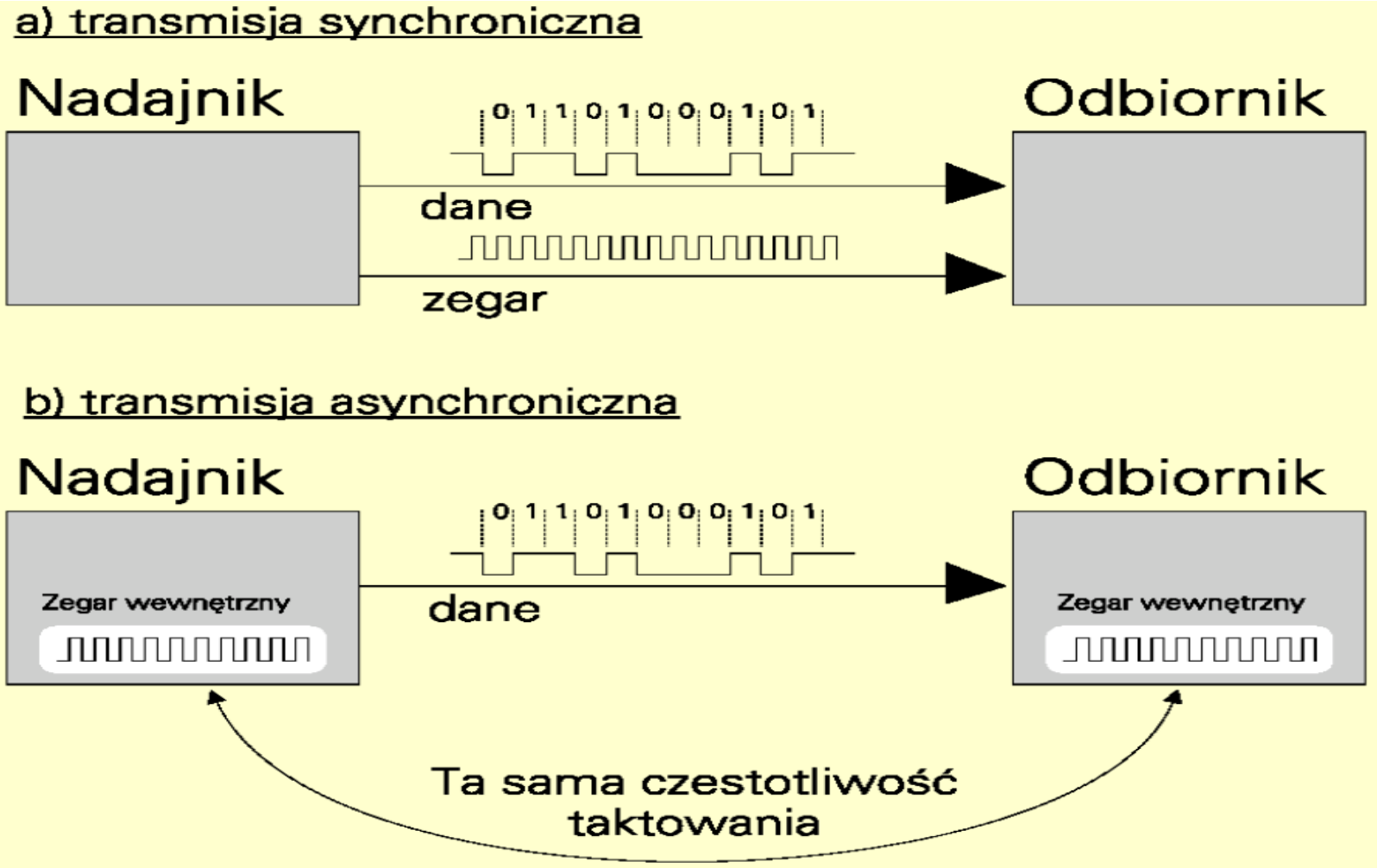


# Ramka danych transmitera UART (2)

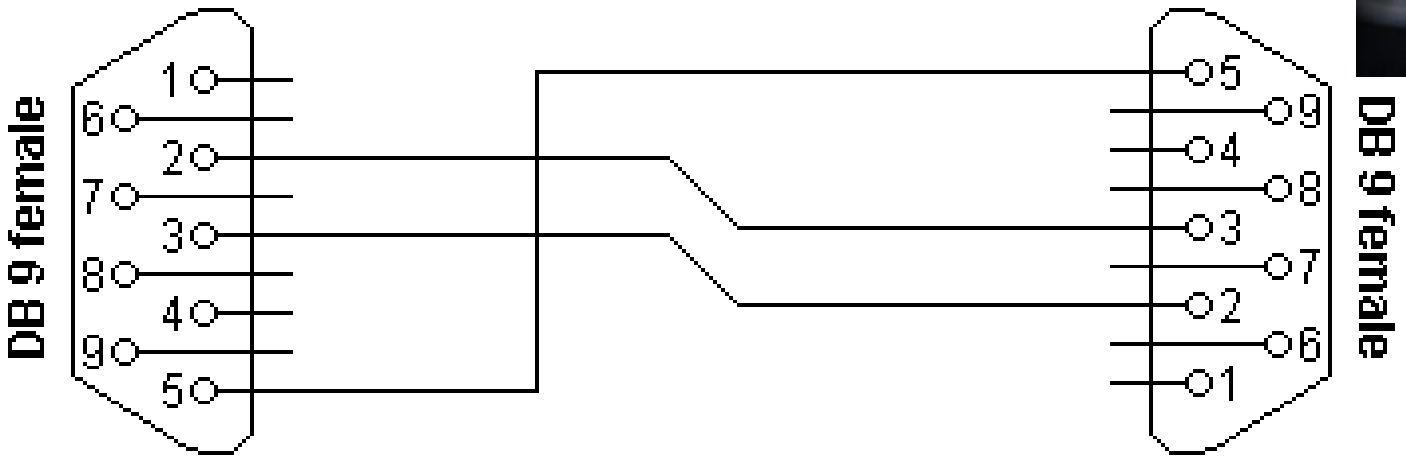



Przesylana dana: 0100.1011b = 0x4B

# Transmisja synchroniczna vs asynchroniczna?



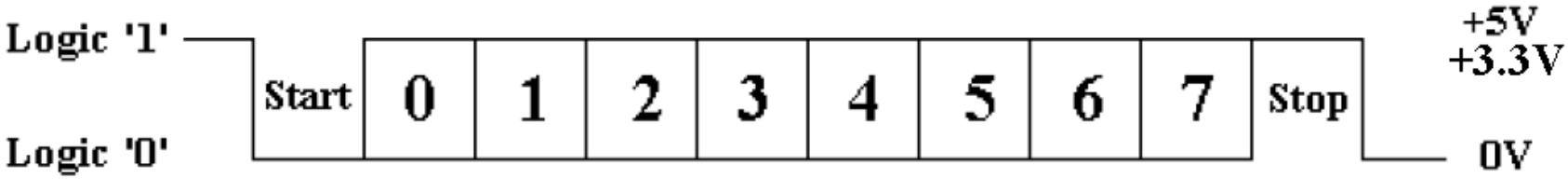
# Kabel null-modem EIA 232



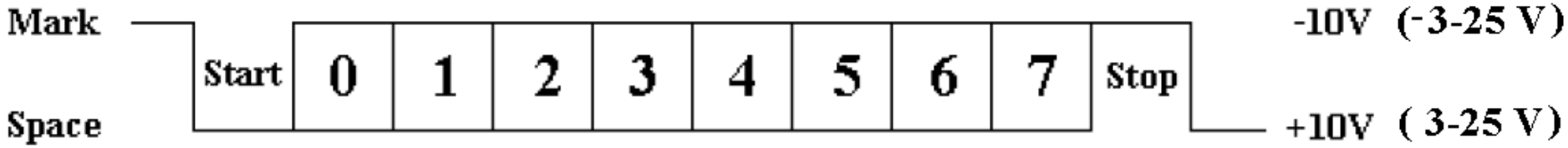

| Connector 1 | Connector 2 | Function |
|---|---|---|
| 2 | 3 | Rx ← Tx |
| 3 | 2 | Tx → Rx |
| 5 | 5 | Signal ground |

# Poziomy napięć określone przez standard EIA 232

## Wyjście procesora

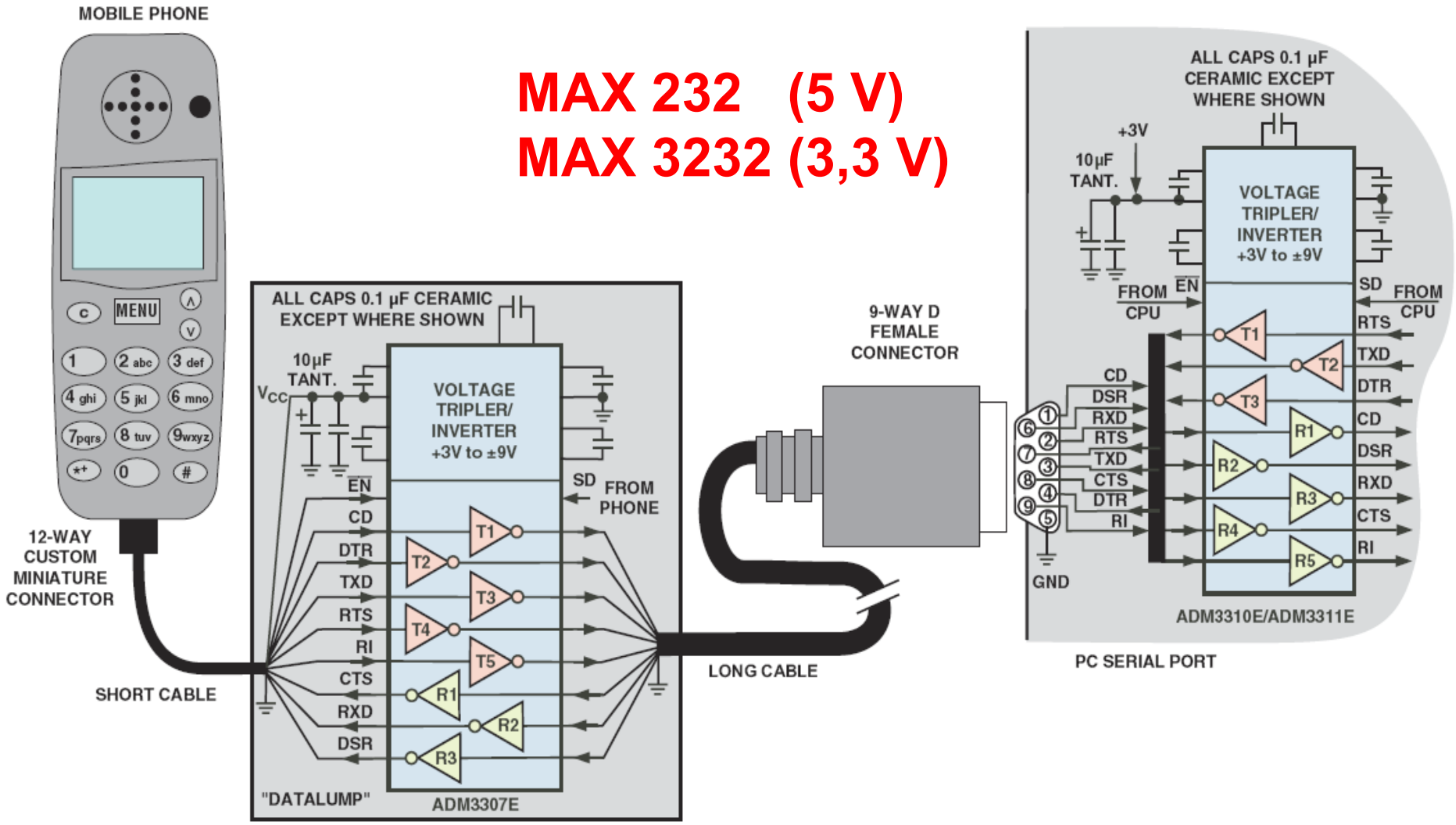



## Standard EIA RS 232




RS-232 Logic Waveform

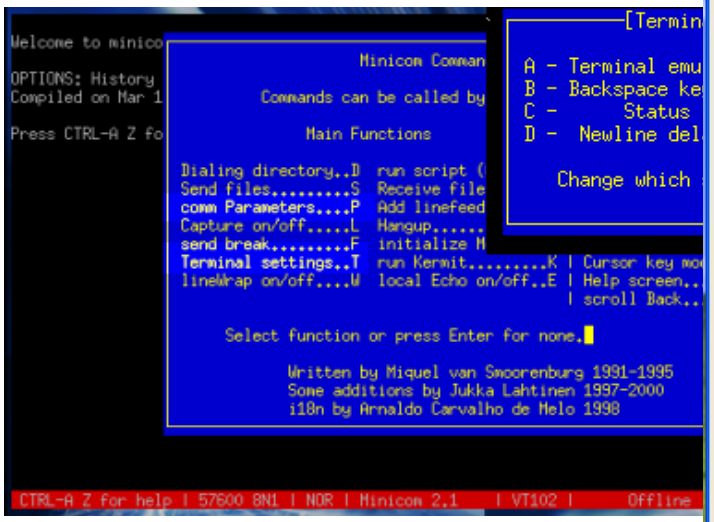
# Konwerter poziomów napięć



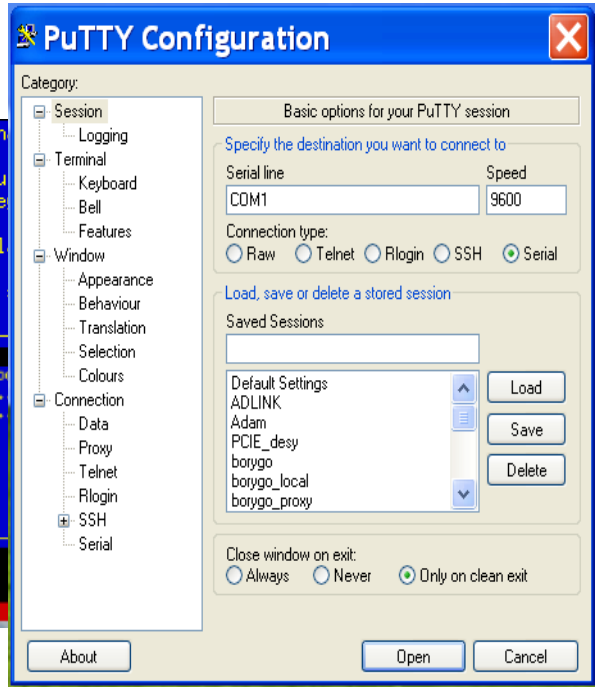
# Programy do komunikacji z wykorzystaniem standardu EIA RS232

## Program Hyper terminal

## Program minicom




## Program ssh

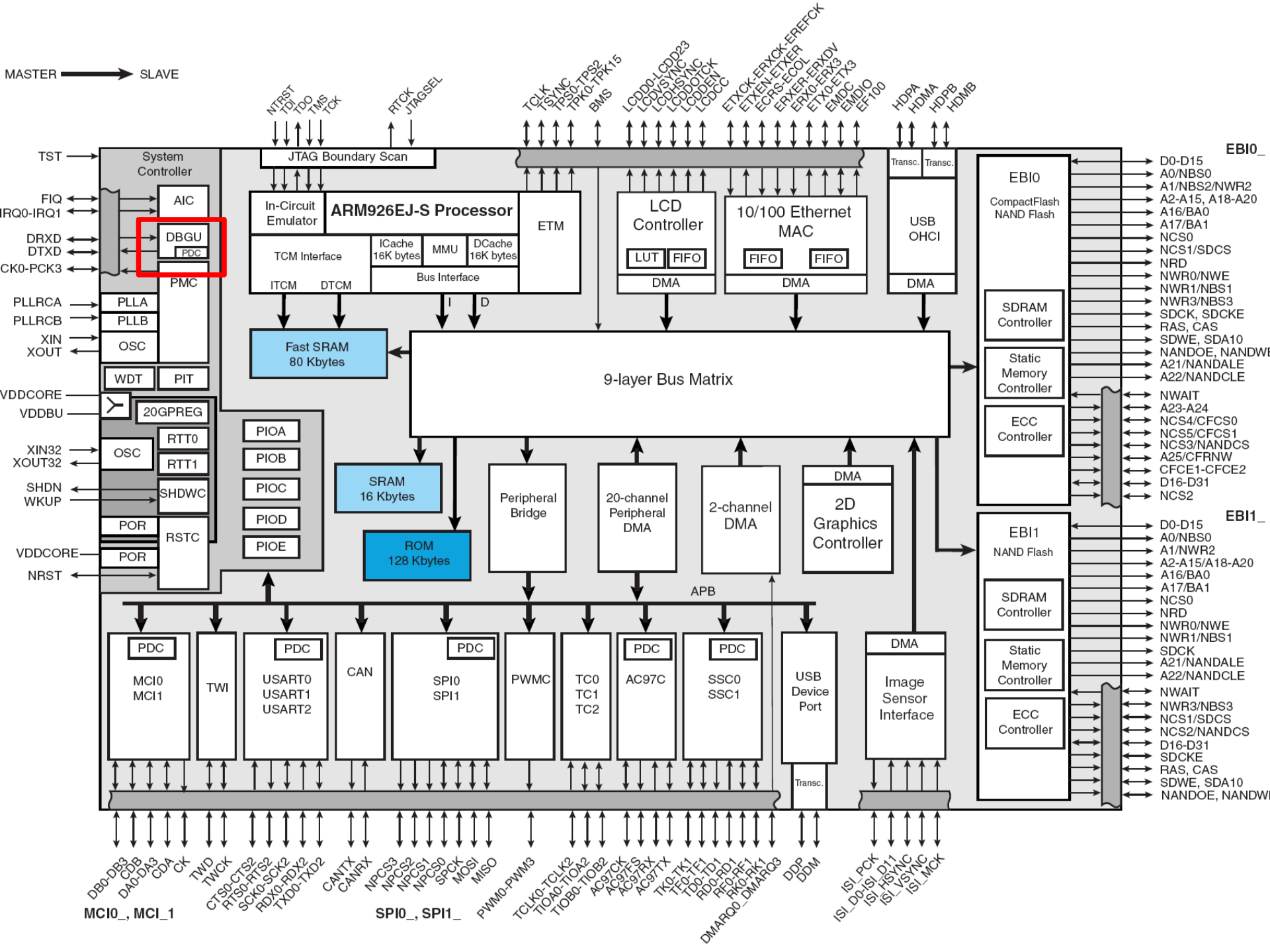



## Program Terminal

(http://www.elester-pkp.com.pl/index.php?id=92&lang=pl&zoom=0)

# Low-Power Universal Asynchronous Receiver Transmitter (LPUART)

**(chapter 41)**


MASTER
SLAVE
NTRST
TDI
TDO
TMS
TCK
RTCK
JTAGSEL
TCLK
TSYNC
TPS0-TPS2
TPK0-TPK15
BMS
LCDD0-LCDD23
LCDVSYNC
LCDHSYNC
LCDDOTCK
LCDDEN
LCDCC
ETXCK-ERXCK-EREFCK
ETXEN-ETXER
ECRS-ECOL
ERXER-ERXDV
ERX0-ERX3
ETX0-ETX3
EMDC
EMDIO
EF100
HDPA
HDMA
HDPB
HDMB
TST
System Controller
JTAG Boundary Scan
FIQ
IRQ0-IRQ1
DRXD
DTXD
CK0-PCK3
AIC
DBGU
PDC
PMC
In-Circuit Emulator
ARM926EJ-S Processor
ETM
TCM Interface
ICache 16K bytes
MMU
DCache 16K bytes
Bus Interface
ITCM
DTCM
I
D
LCD Controller
LUT
FIFO
DMA
10/100 Ethernet MAC
FIFO
FIFO
DMA
Transc.
Transc.
USB OHCI
DMA
EBI0
CompactFlash NAND Flash
SDRAM Controller
Static Memory Controller
ECC Controller
EBI0_
D0-D15
A0/NBS0
A1/NBS2/NWR2
A2-A15, A18-A20
A16/BA0
A17/BA1
NCS0
NCS1/SDCS
NRD
NWR0/NWE
NWR1/NBS1
NWR3/NBS3
SDCK, SDCKE
RAS, CAS
SDWE, SDA10
NANDOE, NANDWE
A21/NANDALE
A22/NANDCLE
NWAIT
A23-A24
NCS4/CFCS0
NCS5/CFCS1
NCS3/NANDCS
A25/CFRNW
CFCE1-CFCE2
D16-D31
NCS2
PLLRCA
PLLRCB
XIN
XOUT
PLLA
PLLB
OSC
WDT
PIT
Fast SRAM 80 Kbytes
9-layer Bus Matrix
VDDCORE
VDDBU
20GPREG
RTT0
RTT1
SHDWC
POR
RSTC
XIN32
XOUT32
OSC
SHDN
WKUP
VDDCORE
POR
NRST
PIOA
PIOB
PIOC
PIOD
PIOE
SRAM 16 Kbytes
ROM 128 Kbytes
Peripheral Bridge
20-channel Peripheral DMA
2-channel DMA
DMA
2D Graphics Controller
APB
EBI1
NAND Flash
SDRAM Controller
Static Memory Controller
ECC Controller
EBI1_
D0-D15
A0/NBS0
A1/NWR2
A2-A15/A18-A20
A16/BA0
A17/BA1
NCS0
NRD
NWR0/NWE
NWR1/NBS1
SDCK
A21/NANDALE
A22/NANDCLE
NWAIT
NWR3/NBS3
NCS1/SDCS
NCS2/NANDCS
D16-D31
SDCKE
RAS, CAS
SDWE, SDA10
NANDOE, NANDWE
PDC
MCI0
MCI1
TWI
PDC
USART0
USART1
USART2
CAN
PDC
SPI0
SPI1
PWMC
TC0
TC1
TC2
PDC
AC97C
PDC
SSC0
SSC1
USB Device Port
Transc.
DMA
Image Sensor Interface
DB0-DB3
CDB
DA0-DA3
CDA
CK
TWD
TWCK
CTS0-CTS2
RTS0-RTS2
SCK0-SCK2
RDX0-RDX2
TXD0-TXD2
CANTX
CANRX
NPCS3
NPCS2
NPCS1
NPCS0
SPCK
MOSI
MISO
PWM0-PWM3
TCLK0-TCLK2
TIOA0-TIOA2
TIOB0-TIOB2
AC97CK
AC97FS
AC97RX
AC97TX
TK0-TK1
TF0-TF1
TD0-TD1
RD0-RD1
RF0-RF1
RK0-RK1
DMARQ0_DMARQ3
DDP
DDM
ISI_PCK
ISI_D0-ISI_D11
ISI_HSYNC
ISI_VSYNC
ISI_MCK
MCI0_, MCI_1
SPI0_, SPI1_

# Low-Power Universal Asynchronous Receiver Transmitter

## Features of LPUART:

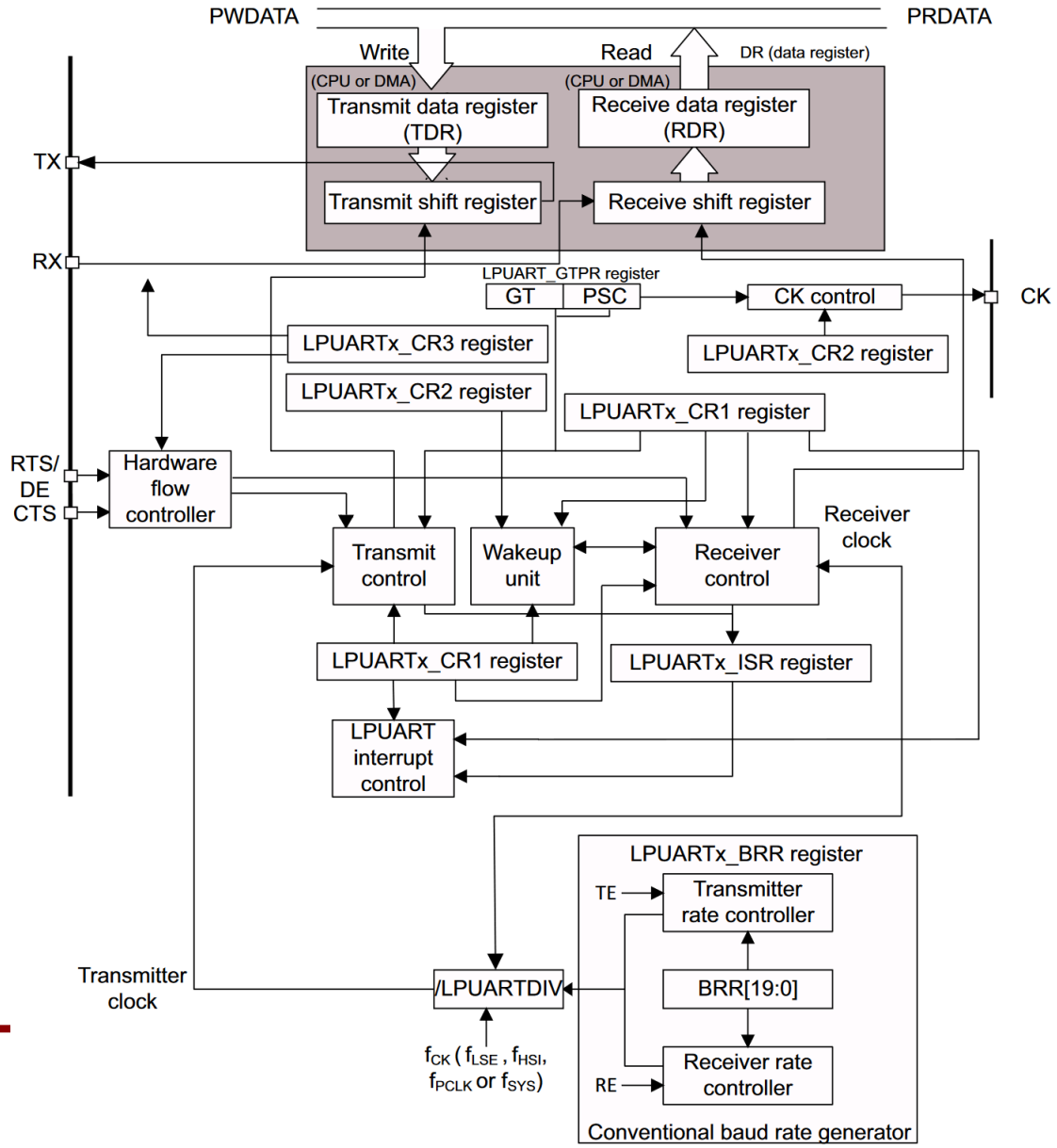
- Full-duplex asynchronous data transmission compatible with RS232 standard
- Programmable data packet size: 7, **8** or 9 bits
- Programmable data order (**LSB** or MSB first)
- Configurable stop bits (**1** or 2), configurable parity bit
- Single-wire half-dulex communication support
- Limited power consumption (available even in STOP mode)
- Hardware support for Direct Memory Access
- Swappable Tx/Rx pin configuration
- Hardware flow control for modem and RS-485 transceiver (Driver Enable)
- Support for hardware flow control: CTS, RTS

# Low-Power Universal Asynchronous Receiver Transmitter

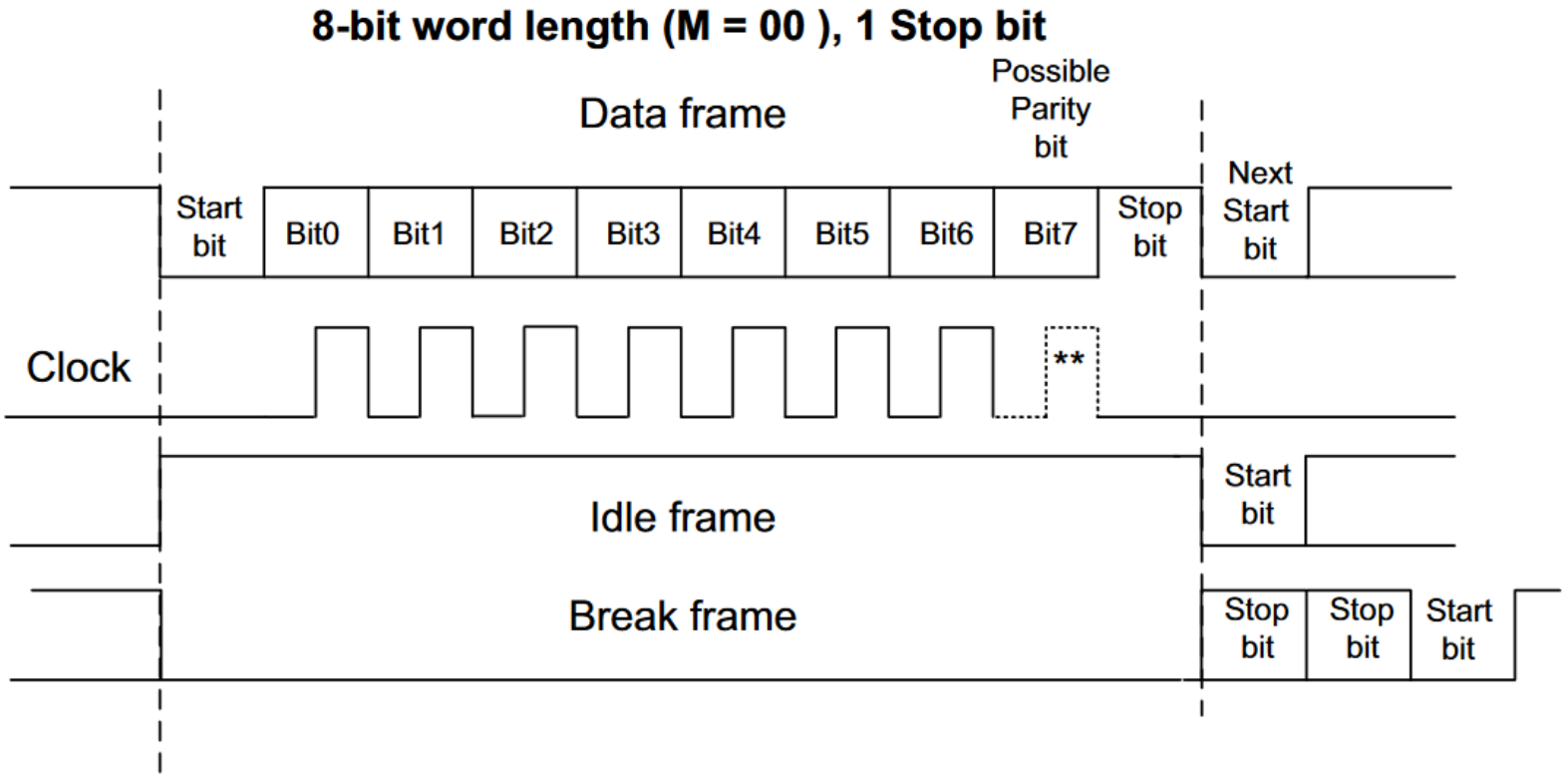
## Features of LPUART:

- **Transfer detection flags:**
  - Receive buffer full
  - Transmit buffer empty
  - Busy and end of transmission flags
- **Parity control:**
  - Transmits parity bit
  - Checks parity of received data byte
- **Four error detection flags:**
  - Overrun error
  - Noise detection
  - Frame error
  - Parity error
- Fourteen interrupt sources with flags

# Low-Power Universal Asynchronous Receiver Transmitter



# LPAURT – 8N1 Typical Data Transmission



# Character Transmission Procedure

**Configure GPIO Port for Tx function**

1. Configure GPIO_C Port to alternate/peripheral function (AF8).

**Configuration (for 8N1)**

1. Program the M bits in LPUART_CR1 to define the word length (8 bit).
2. Select the desired baud rate using the LPUART_BRR register (115.200 bps).
3. Program the number of stop bits in LPUART_CR2 (1 stop bit).
4. Enable the LPUART by writing the UE bit in LPUART_CR1 register to 1.
5. Disable DMA (DMAT) in LPUART_CR3

**Transmission**

6. Set the TE bit in LPUART_CR1 to send an idle frame as first transmission.
7. Write the data to send in the LPUART_TDR register (this clears the TXE bit). Repeat this for each data to be transmitted in case of single buffer.
8. After writing the last data into the LPUART_TDR register, wait until TC=1. This indicates that the transmission of the last frame is complete. This is required for instance when the LPUART is disabled or enters the Halt mode to avoid corrupting the last transmission.

# Character Reception Procedure

**Configure GPIO Port for Rx function**

1. Configure GPIO_C Port to alternate/peripheral function (AF8).

**Configuration (for 8N1)**

1. Program the M bits in LPUART_CR1 to define the word length (8 bits).
2. Select the desired baud rate using the baud rate register LPUART_BRR (115.200 kbps)
3. Program the number of stop bits in LPUART_CR2 (1 stop bit).
4. Enable the LPUART by writing the UE bit in LPUART_CR1 register to 1.
5. Disable DMA (DMAR) in LPUART_CR3.
6. Set the RE bit LPUART_CR1. This enables the receiver which begins searching for a start bit.

**Reception**

1. Wait for The RXNE bit to be set.
2. An interrupt is generated if the RXNEIE bit is set.
3. Check for possible error (frame, noise, overrun, parity error)
4. Read data from LPUART_RDR register.

# Configuration of DBGU transceiver

**static void Open_LPUART (void)** {

1. Disable interrupts
2. Configure Receiver/Transmitter
3. Enable Receiver/Transmitter

}

**Baudrate configuration:**

LPUARTDIV value in LPUART_BRR register

f_CK from SysTick_Config (here 4000000 / 1000)

$$\text{Tx/Rx baud} = \frac{256 \times f_{CK}}{\text{LPUARTDIV}}$$

void **LPUART_Init** (void) {

...

```
RCC->APB1ENR2 |= RCC_APB1ENR2_LPUART1EN;
LPUART1->BRR = (256 * 4000000) / 115200;
LPUART1->CR1 |= USART_CR1_RE | USART_CR1_TE | USART_CR1_UE;
}
```

# Read and write via DBGU port

Interrupts are disabled.

```
void SendData (const char Buffer)
{
   while ( data_are_in_buffer ) {
        while ( …TXE... ){};          /* wait until Tx buffer busy – not set TXE flag LPUART_ISR */
        LPUART->TDR = ...             /* write a single char to Transmitter Data Register  */
         }
}

void ReadData (char *Buffer, unsigned int Size){
        do {
        While ( ...RXNE... ){};       /* wait until data available, RXNE is set in ISR register */
        Buffer[...] = LPUART->RDR;    /* read data from Receiver Data Register */
        } while ( …read_enough_data... )
}
```

# Daty egzaminów

Egzamin #1 (sala A1-A4) – 23.06.2023  9.15-10.00

Egzamin #2 (sala A1-A4) – 30.06.2023  9.15-10.00

Egzamin #3 (sala A1) – xx.09.2023  9.15-10.00

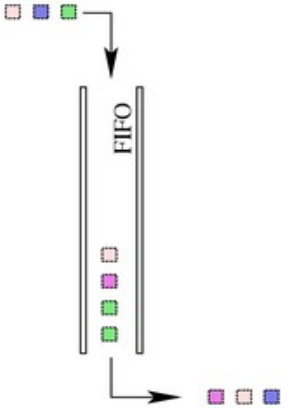
# Buforowanie danych

# Struktura stosu (1)

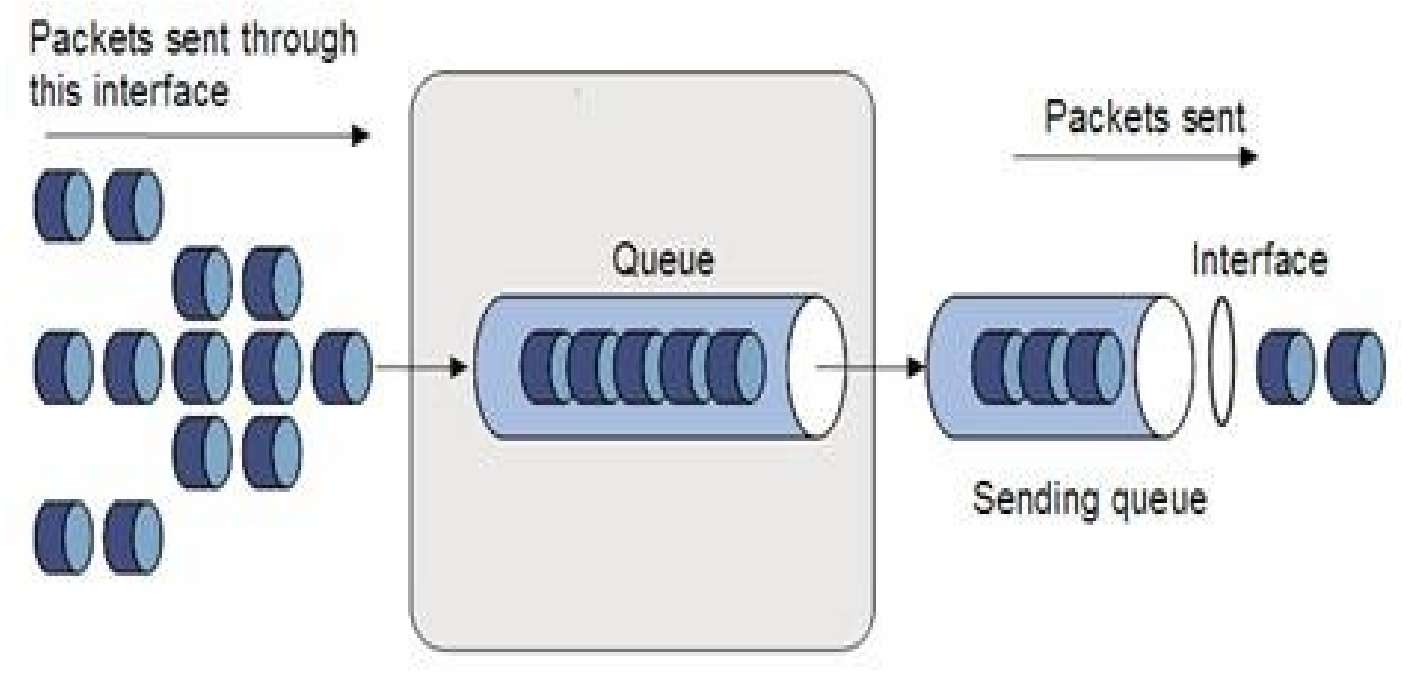
**Stos (ang. stack lub LIFO Last-In, First-Out) -** liniowa struktura danych, w której dane odkładane są na wierzch stosu i z wierzchołka stosu są zdejmowane. Ideę stosu danych można zilustrować jako stos położonych jedna na drugiej książek – nowy egzemplarz kładzie się na wierzch stosu i z wierzchu stosu zdejmuje się kolejne egzemplarze. Elementy stosu poniżej wierzchołka stosu można wyłącznie obejrzeć, aby je ściągnąć, trzeba najpierw po kolei ściągnąć to, co jest nad nimi

**FIFO (ang. First In, First Out) -** przeciwieństwem stosu LIFO jest kolejka, bufor typu FIFO (pierwszy na wejściu, pierwszy na wyjściu), w którym dane obsługiwane są w takiej kolejności, w jakiej zostały dostarczone (jak w kolejce do kasy)

First–in First–out (FIFO)

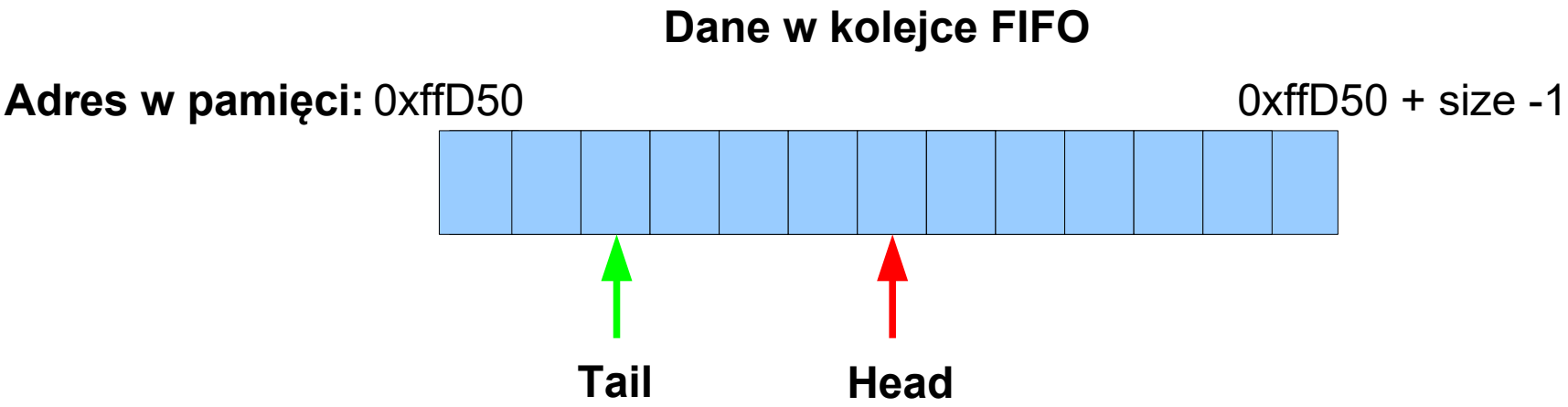


# Kolejka FIFO (1)




- Dane do kolejki FIFO mogą być wpisywane przez kilka niezależnych aplikacji, wątków lub urządzeń. W takiej sytuacji dostęp do kolejki kontrolowany jest przez Semafor (zmienna globalna).
- Dane zgomadzone w kolejce wysyłane są w kolejności w jakiej zostały wpisane.

# Kolejka FIFO (2)




**Zapis danej do kolejki FIFO:**

- Zwiększ wskaźnik Head o jeden, zapisz daną.

**Odczyt danej z kolejki FIFO:**

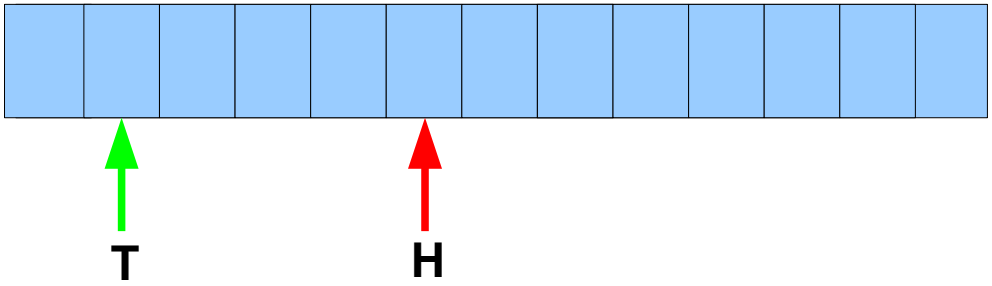
- Odczytaj daną, zwiększ wskaźnik Tail o 1.

W przypadku, gdy Tail lub Head wskazuje na ostatni dostępny elemet kolejki zamiast inkrementacji wskaźnik jest zerowany. Pozwala to na płynne przesuwanie wskaźników – bufor kołowy (ang. circular buffer).

# Kolejka FIFO (3)

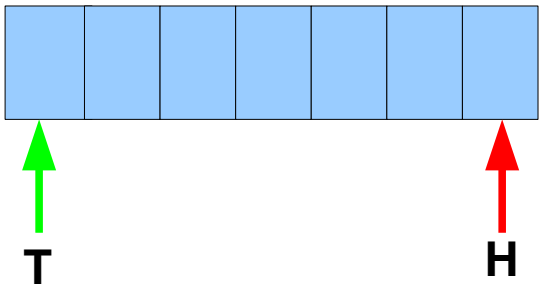
**Kolejka pusta T = H**

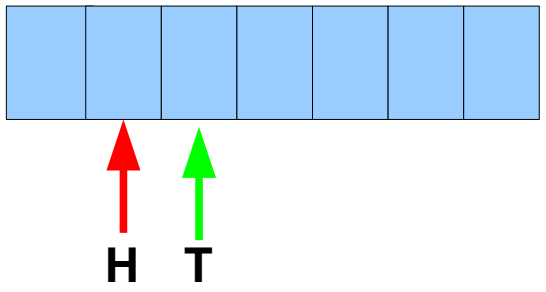

**Dane w kolejce, ilość danych = H – T**




**Brak miejsca w kolejce**

(T = 0) & (H = Size) lub T – H = 1



# Kolejka FIFO – implementacja w C (1)

```
#define BUFFERSIZE   0xFF                    /* FIFO buffer size and mask */
typedef struct FIFO {
    char buffer [BUFFERSIZE+1];
     int head;
     int tail;
};
void FIFO_Init (struct FIFO *Fifo);
void FIFO_Empty (struct FIFO *Fifo);
int FIFO_Put (struct FIFO *Fifo, char Data);
int FIFO_Put (struct FIFO *Fifo, char *Data);


void FIFO_Init (struct FIFO *Fifo){
          Fifo->head=0;
          Fifo->tail=0;

          /* optional: initialize data in buffer with 0 */
}
```

# Kolejka FIFO – implementacja w C (2)

```
void FIFO_Empty (struct FIFO *Fifo){
        Fifo->head = Fifo->tail;                                    /* now FIFO is empty*/
}
int FIFO_Put (struct FIFO *Fifo, char Data){
        if ((Fifo->tail-Fifo->head)==1 || (Fifo->tail-Fifo->head)==BUFFERSIZE)){
        return -1; };                                               /* FIFO overflow */
        Fifo->buffer[Fifo->head] = Data;
        Fifo->head = Fifo->head++ & BUFFERSIZE;
        return 1;                                                   /* Put 1 byte successfully */
}
int FIFO_Get (struct FIFO *Fifo, char *Data){
        If ((Fifo->head!=Fifo->tail)){
        *Data = Fifo->buffer[Fifo->tail];
        Fifo->tail = Fifo->tail++ & BUFFERSIZE;
        return 1;                                                   /* Get 1 byte successfully */
        } else return -1;                                           /* No data in FIFO */
}
```

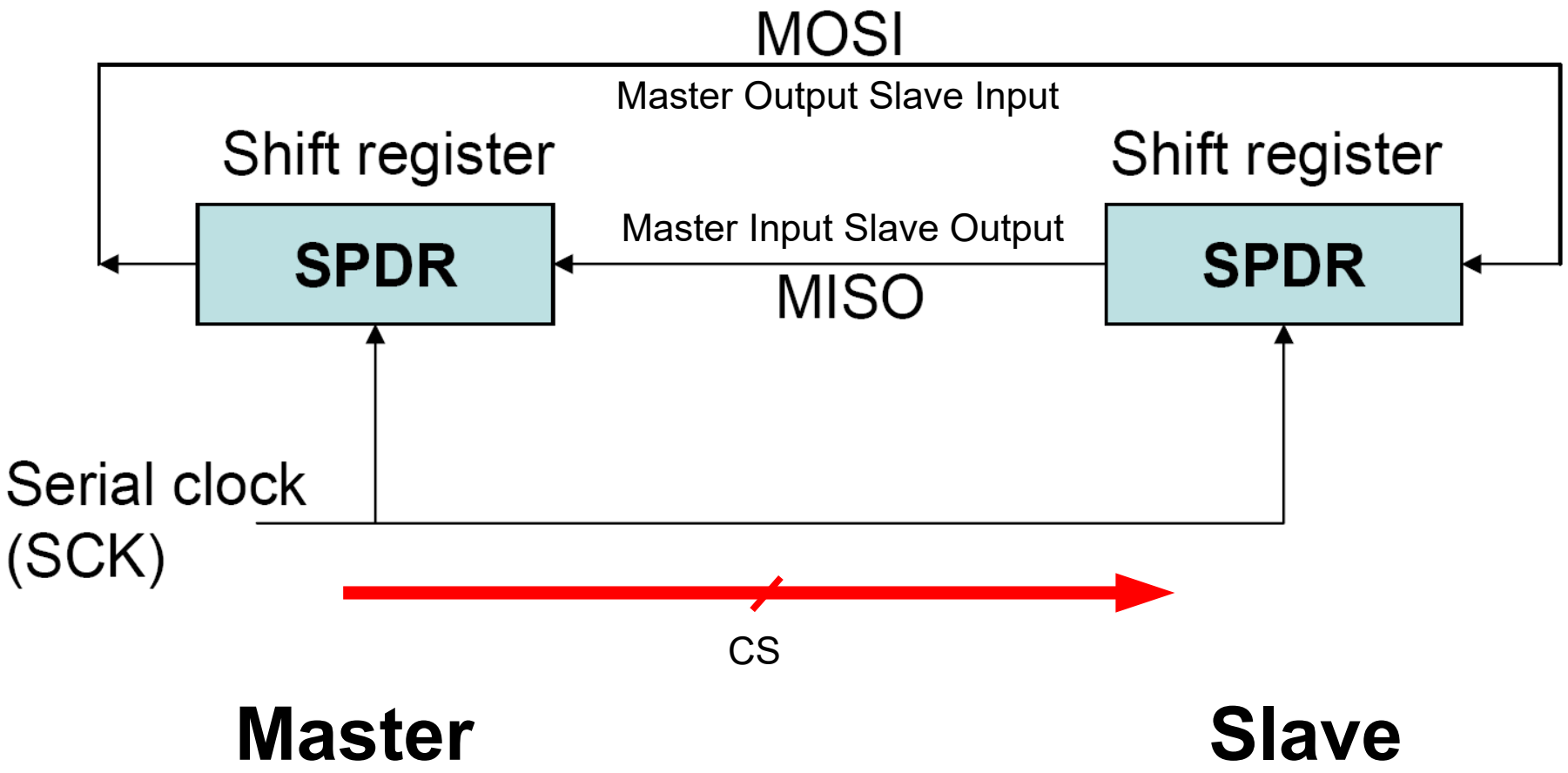
# Serial Peripheral Interface

# Serial Peripheral Interface

**Cechy interfejsu SPI:**

- Szeregowa transmisja synchroniczna,
- Transfer full duplex, master-slave lub master-multi-slave,
- Duża szybkość transmisji (>12 Mbit/s),
- Zastosowanie:
  - układy peryferyjne (ADC, DAC, RTC, EEPROM, termometry, itp),
  - sterowanie pomocnicze (matryca CCD z szybkim interfejsem równoległym),
  - karty pamięci z interfejsem szeregowym SD/SDHC/MMC.

# Serial Peripheral Interface



# Protokół interfejsu SPI

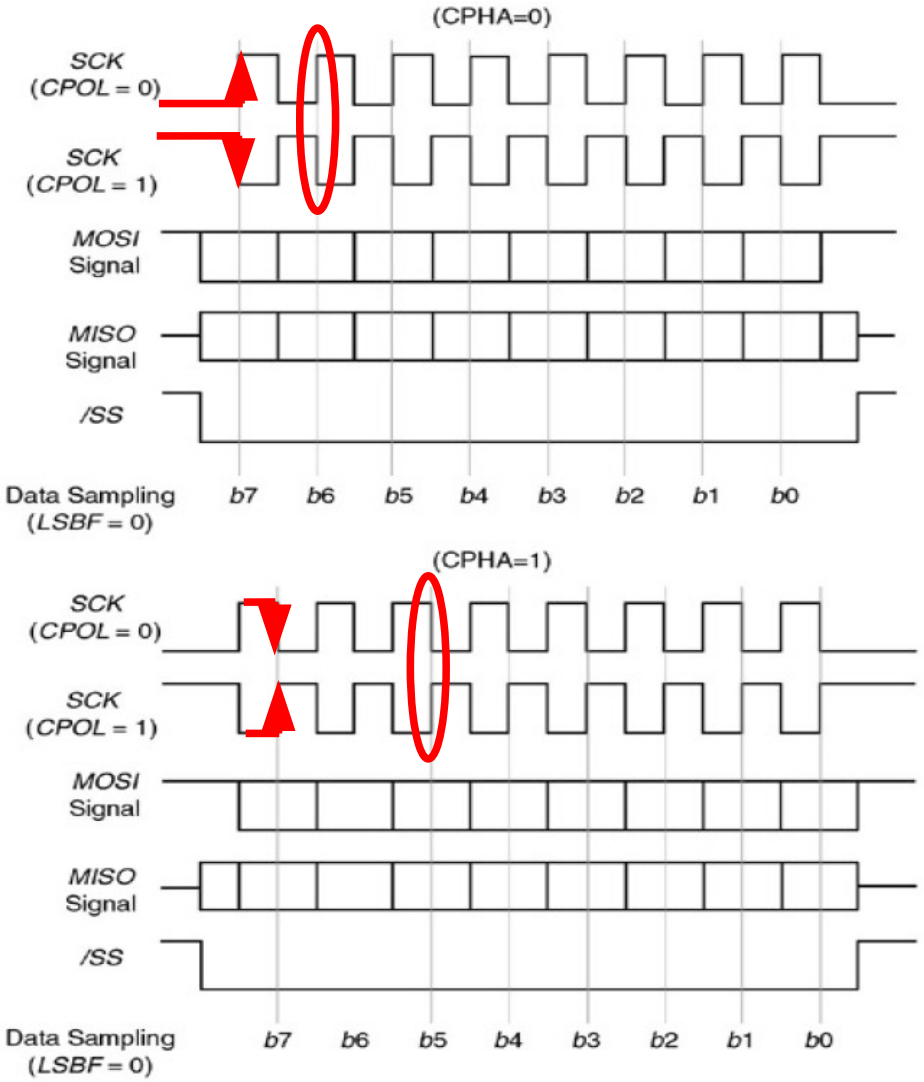
**Konfiguracja sygnału zegarowego:**

**Polaryzacja zegara:**
Polaryzacja **ujemna CPOL = 0** (**stan niski**, 8 impulsów zegara),

Polaryzacja **dodatnia CPOL = 1** (**stan wysoki**, 8 ujemnych impulsów zegara).

**Faza zegara:**
Zerowa faza zegara (próbkowanie na pierwszym zboczu zegara),

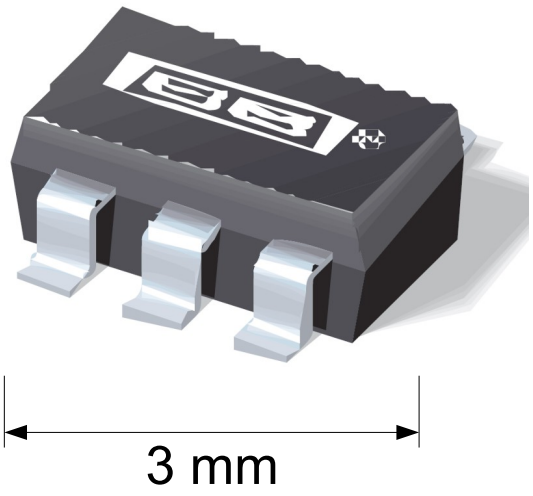
Opóźniona faza zegara (próbkowanie na drugim zboczu zegara).



# Termometr z interfasem SPI

**TMP 121:**

- Obudowa SOT 23-6,
- fclk mak. = 15 MHz
- Interfejs: SPI-Compatible Interface
- Rozdzielczość: 12-Bit + Sign, 0,0625°C
- Dokładność: ±1.5°C od −25°C do +85°C
- Pobór prądu w stanie uśpienia: 50µA (mak.)
- Zasilanie: 2,7V to 5,5V




| D15 | D14 | D13 | D12 | D11 | D10 | D9 | D8 |
|---|---|---|---|---|---|---|---|
| T12 | T11 | T10 | T9 | T8 | T7 | T6 | T5 |

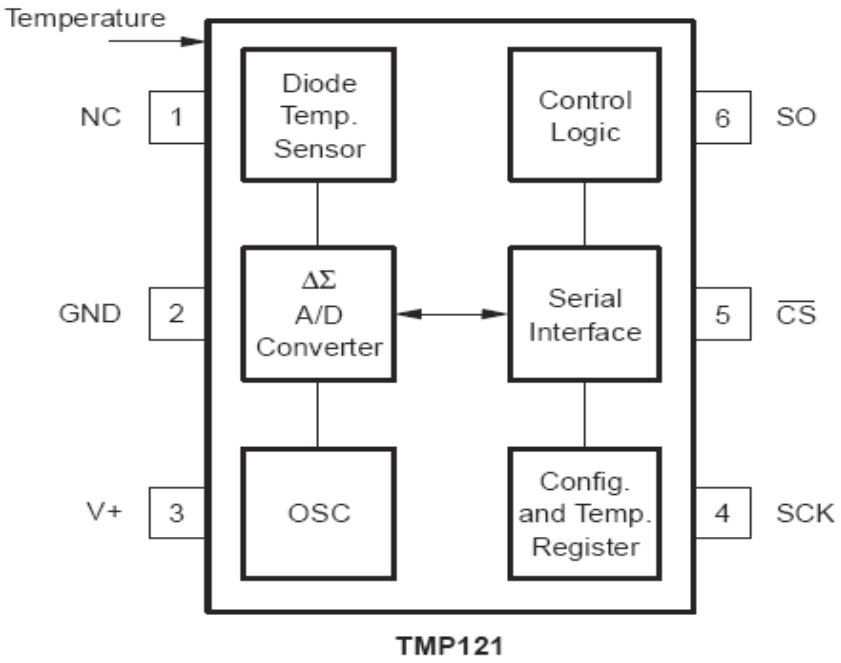
| D7 | D6 | D5 | D4 | D3 | D2 | D1 | D0 |
|---|---|---|---|---|---|---|---|
| T4 | T3 | T2 | T1 | T0 | 0 | Z | Z |

Table 1. Temperature Register

| TEMPERATURE (°C) | DIGITAL OUTPUT(1) (BINARY) | HEX |
|---|---|---|
| 150 | 0100 1011 0000 0000 | 4B00 |
| 125 | 0011 1110 1000 0000 | 3E80 |
| 25 | 0000 1100 1000 0000 | 0C80 |
| 0.0625 | 0000 0000 0000 1000 | 0008 |
| 0 | 0000 0000 0000 0000 | 0000 |
| −0.0625 | 1111 1111 1111 1000 | FFF8 |
| −25 | 1111 0011 1000 0000 | F380 |
| −55 | 1110 0100 1000 0000 | E480 |
| (1) The last two bits are high impedance and are shown as 00 in the table. | | |

Table 2. Temperature Data Format

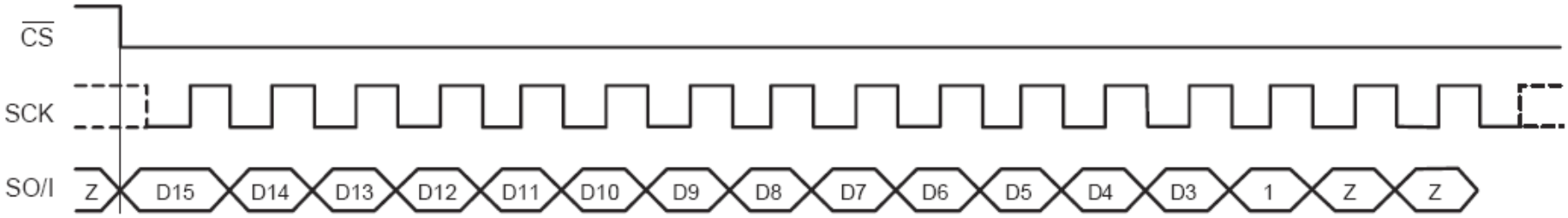
# Ramka SPI termometru TMP121




| D15 | D14 | D13 | D12 | D11 | D10 | D9 | D8 |
|---|---|---|---|---|---|---|---|
| T12 | T11 | T10 | T9 | T8 | T7 | T6 | T5 |

| D7 | D6 | D5 | D4 | D3 | D2 | D1 | D0 |
|---|---|---|---|---|---|---|---|
| T4 | T3 | T2 | T1 | T0 | 0 | Z | Z |

Table 1. Temperature Register

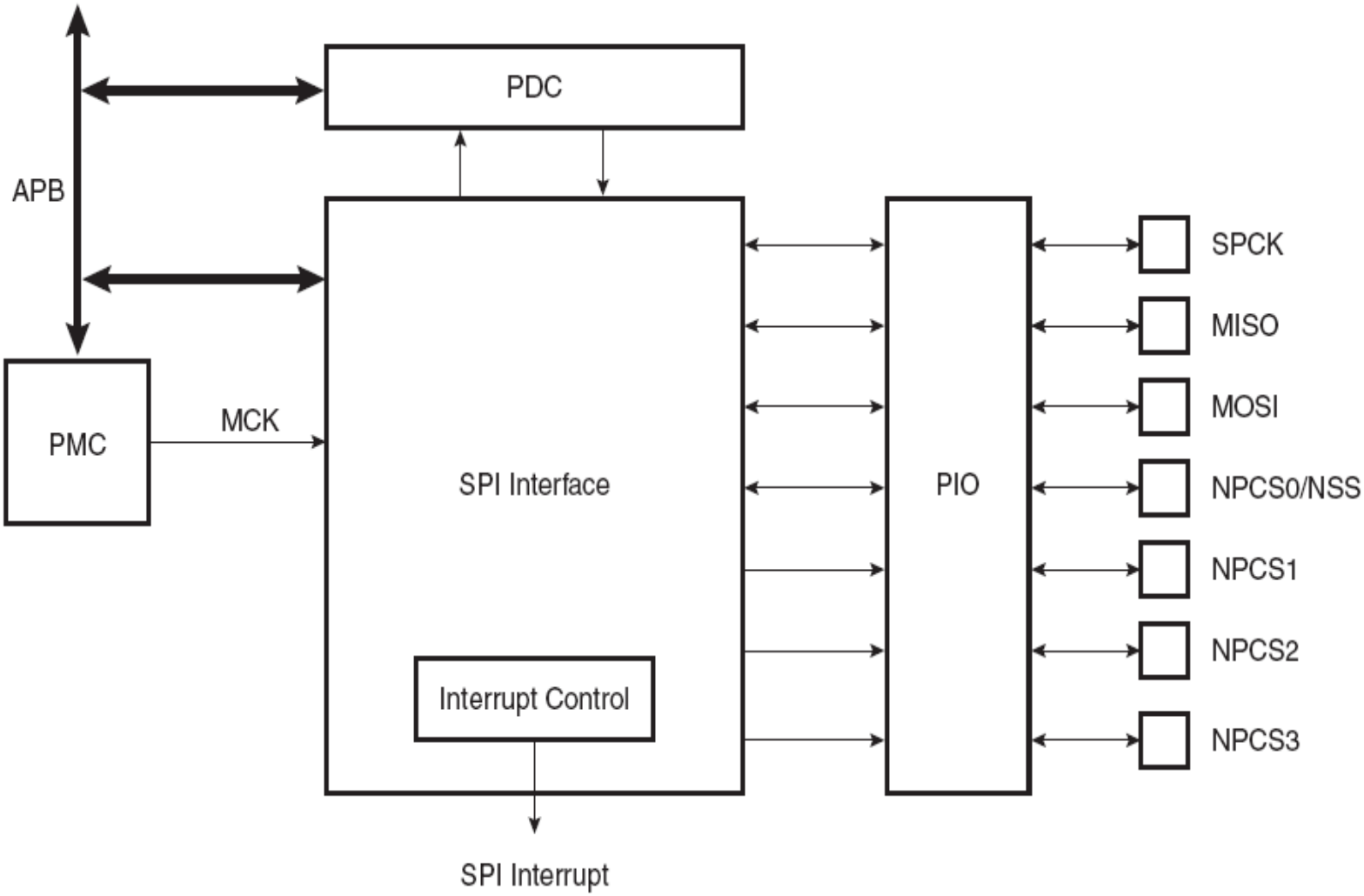


# Moduł SPI procesora ARM AT91SAM9263 (1)

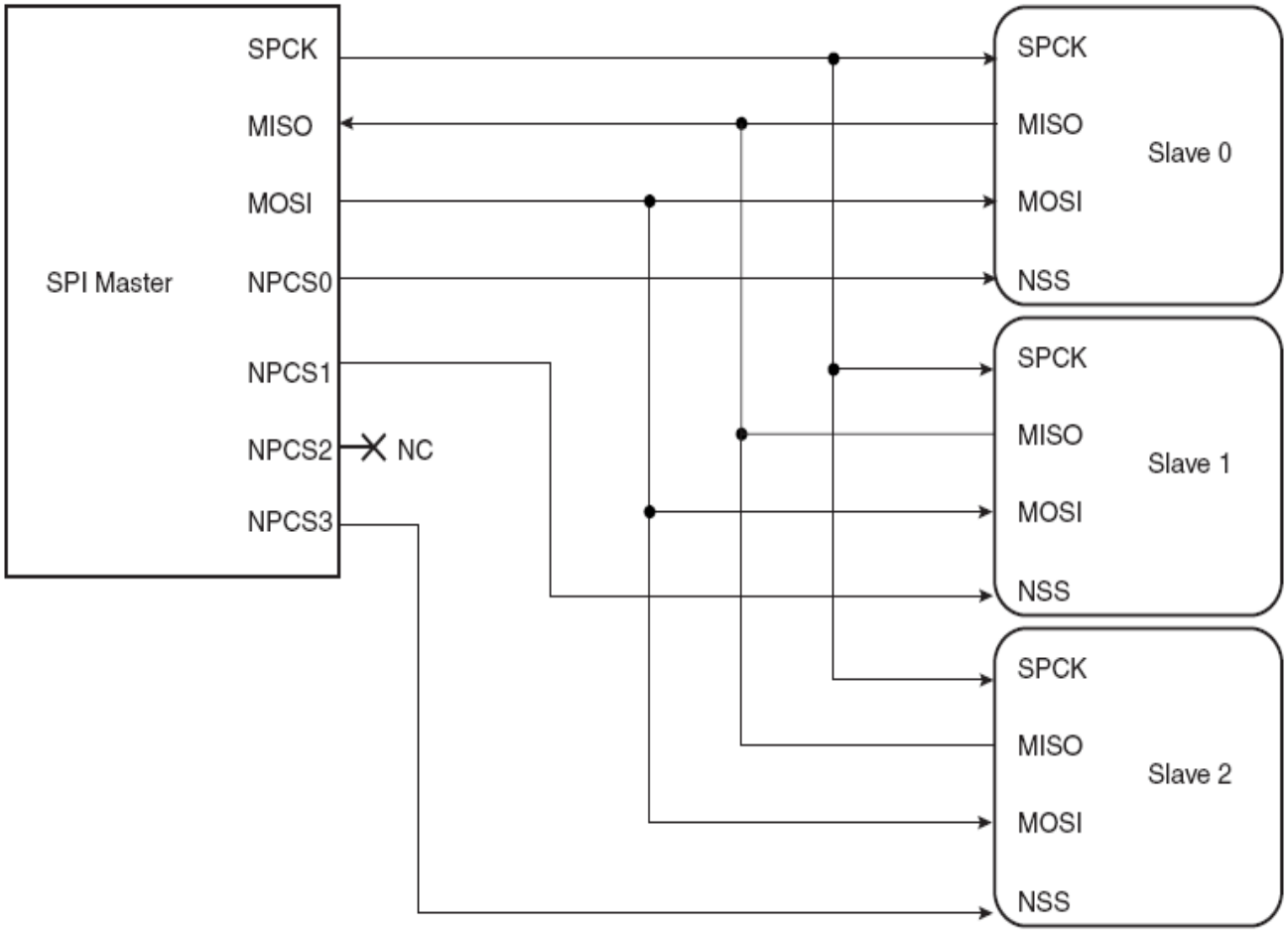
**Cechy modułu SPI:**

- Obsługa transferów w trybie Master lub Slave,
- Bufor nadawczy, odbiorczy oraz bufor transceivera,
- Transfery danych od 8 do 16 bitów,
- Cztery programowalne wyjścia aktywujące urządzenia dołączone do SPI (obsługa do 15 urządzeń),
- Programowalne opóźnienia pomiędzy transferami,
- Programowalna polaryzacja i faza zegara.

# Moduł SPI procesora ARM AT91SAM9263 (2)



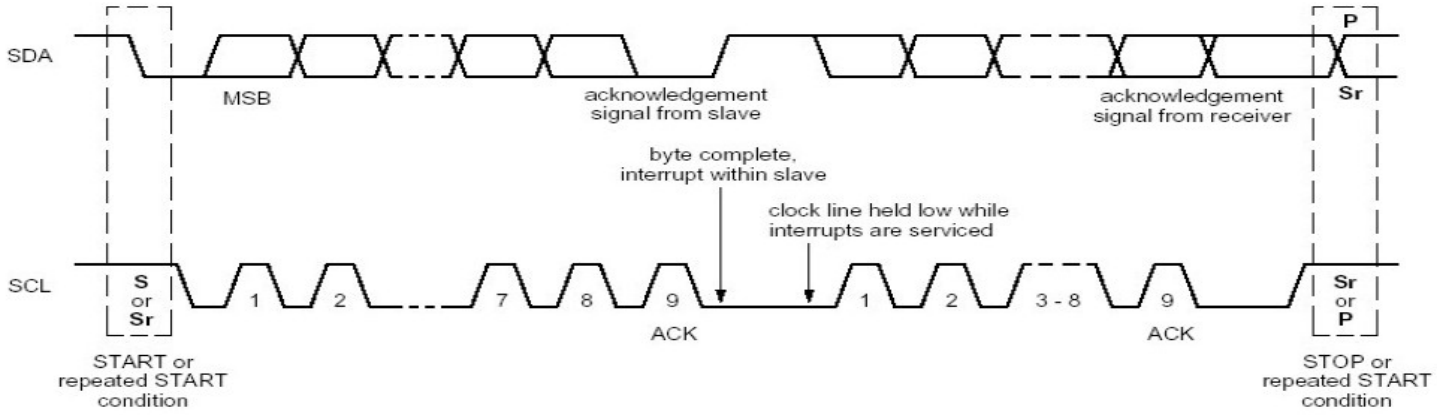
# Moduł SPI procesora ARM (3)



# Magistrala I2C

# Magistrala I2C

- Standard opracowany przez firmę Philips na początku lat 80,
- Dwuprzewodowy interfejs synchroniczny (SDA – linia danych, SCL – linia zegara),
- Transmisja dwukierunkowa, typu master-slave (multi-master), ramki 8-bitowe,
- Szybkość transmisji:
  - 100 kbps (standard mode),
  - 400 kbps (fast mode),
  - 3,4 Mbps (high-speed mode),
- Urządzenia posiadają niepowtarzalne adresy (7-bitów lub 10-bitów),
- Synchronizacja przy pomocy sygnału zegarowego umożliwia pracę urządzeń komunikujących się z różnymi szybkościami,
- Liczba urządzeń dołączonych do magistrali ograniczona jest pojemnością mag. (400 pF),
- Mechanizmy arbitrażu umożliwiające uniknięcie kolizji i utraty danych.

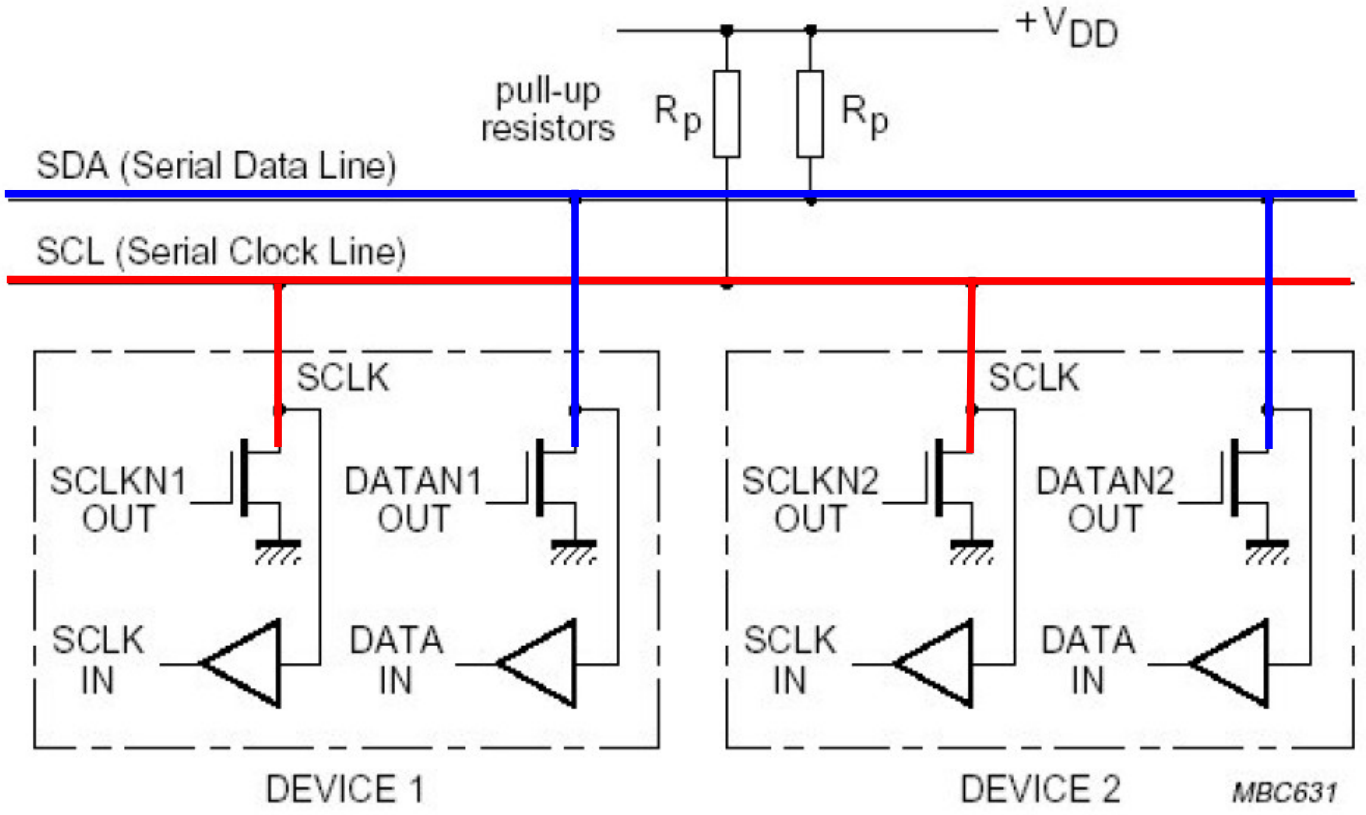


# Zastosowanie interfejsu I2C

W sprzedaży dostępnych jest wiele bardzo tanich układów scalonych sterowanych poprzez I²C:

- PCF8563/8583 - zegar, kalendarz, alarm, timer, dodatkowo może służyć jako RAM
- PCF8574 - pseudo-dwukierunkowy 8-bitowy ekspander
- PCF8576, PCF8577 - sterowniki wyświetlaczy LCD
- PCF8582 - pamięć EEPROM 256 bajtów (1, 2, 4 kB, ... MB)
- PCF8591 - 8-bitowy, 4-kanałowy przetwornik analogowo-cyfrowy i cyfrowo-analogowy

# Magistrala I2C

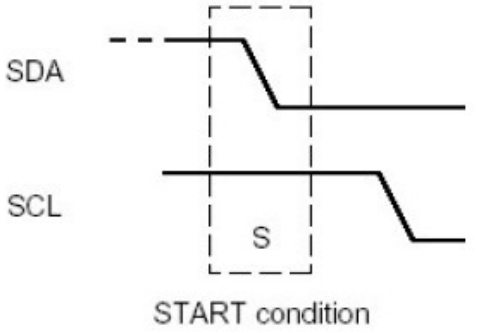



Urządzenie nadrzędne (Master) – inicjuje transmisję, generuje sygnał zegarowy

Urządzenie podrzędne (Slave) – analizuje wysłany przez urządzenie adres i transmituje lub odbiera dane.

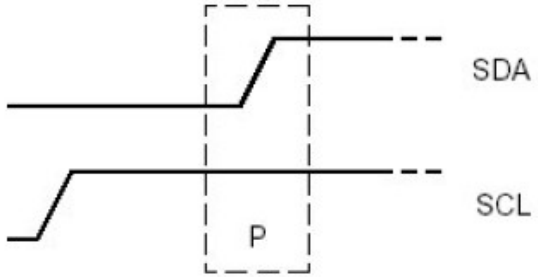
# Rozpoczęcie oraz zakończenie transmisji

Rozpoczęcie transmisji – generacja sygnału **START** (opadające zbocze na szynie SDA, zmiana stanu z “1” na “0” logiczne, podczas ważnego sygnału SCL = ”1”). Sygnał generuje Master.



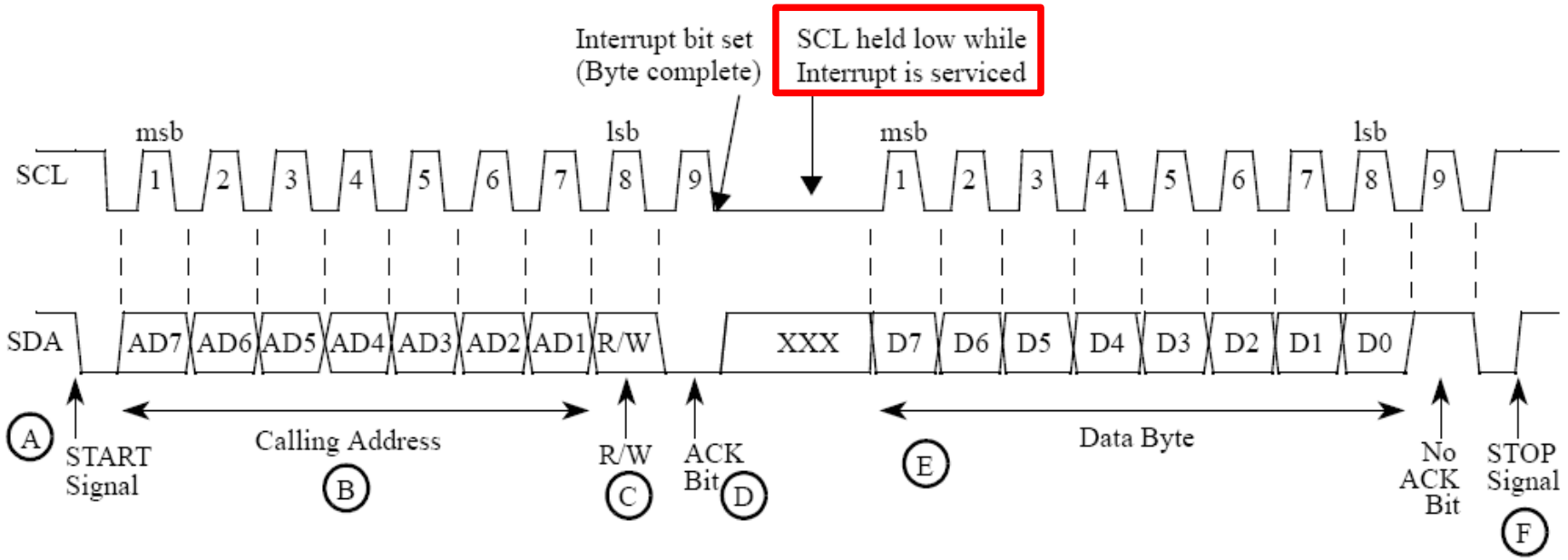
START condition

Zakończenie transmisji – generacja sygnału **STOP** (narastające zbocze na szynie SDA, zmiana stanu z “0” na “1” logiczną,podczas ważnego sygnału SCL = ”1”). Sygnał generuje Master.



STOP condition

# Protokół I2C

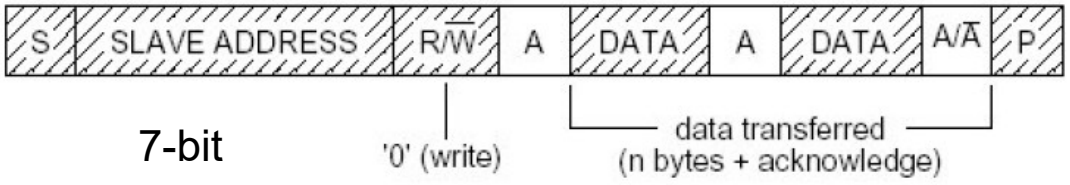



A) Transmisje rozpoczyna Master generując sygnał START.
B) Następnie transmituje 8 bitów danych (7 bitów adresowych, bit R/W).
C) Po transmisji 8 bitów Slave przejmuje magistralę i wymusza odpowiedni poziom na linii SDA (9 takt zegara). Odpowiada w ten sposób bitem potwierdzenia ACK (brak potwierdzenia, ACK = “1”).
E) Po przesłaniu adresu następuje faza odczytu lub zapisu danej do obsługiwanego urządzenia (8 bitów danych).
F) Po przesłaniu danych urządzenie nadrzędne kończy transmisję generując brak potwierdzenia (ACK = “1”) oraz bit stopu.

# Zapis lub odczyt

## Zapis n-bajtów danych

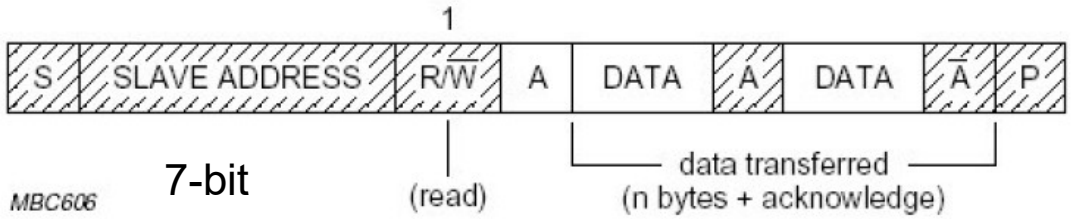
master-transmitter





## Odczyt n-bajtów danych

master-receiver (since second byte)



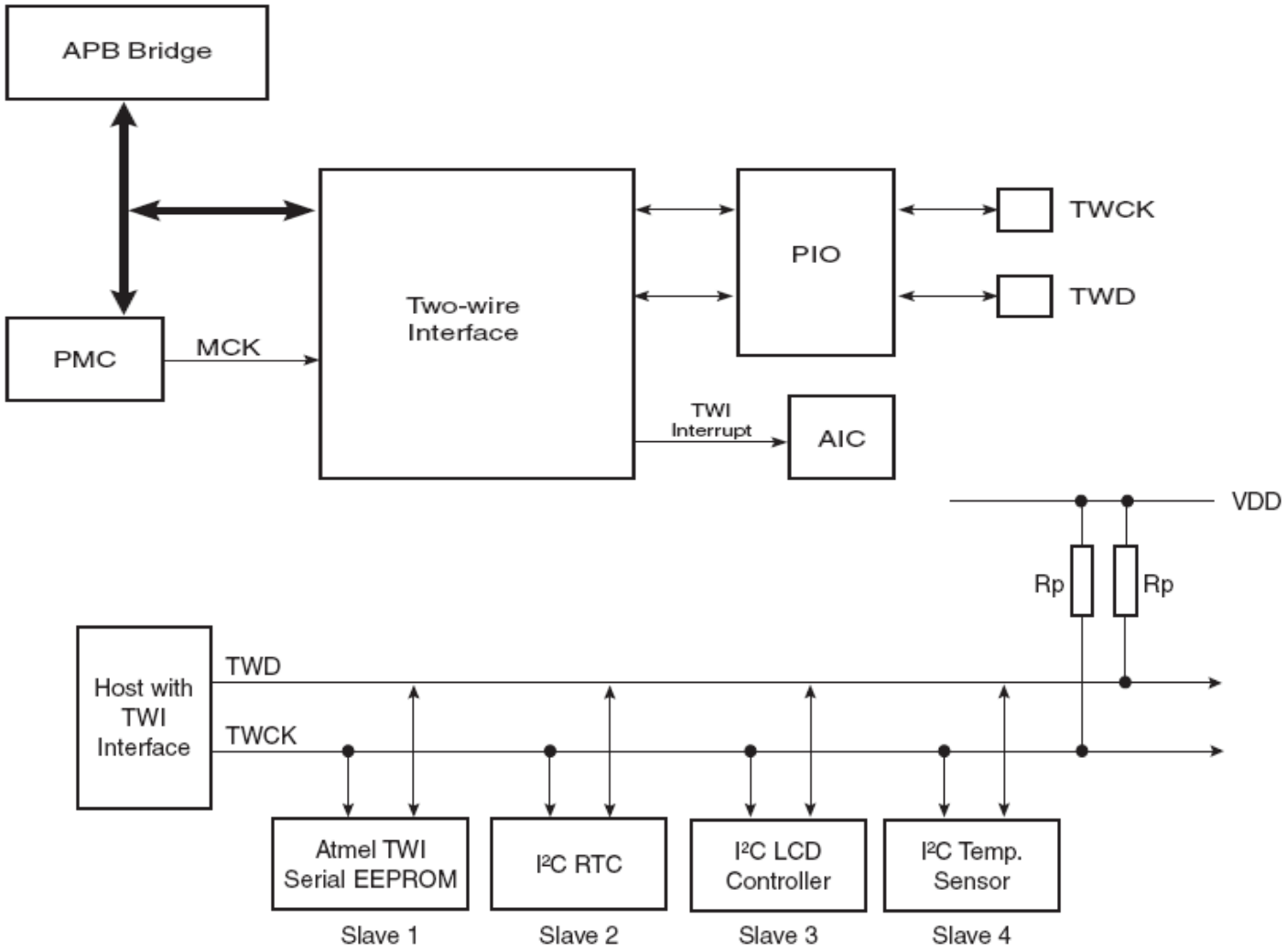
# Two-Wire Interface – standard zgodny z I2C ?

Moduł TWI procesorów ARM jest odpowiednikiem standardu opracowanego przez firmę Philips (firma Philips posiada patent na interfejs I2C).

## Cechy interfejsu SWI procesora AMR firmy ATMEL:

- Zgodny ze standardem I2C,
- Praca w trybie Master, Multimaster lub Slave,
- Umożliwia dołączenie urządzeń zasilanych napięciem 3,3 V,
- Transmisja danych z częstotliwością zegara do 400 kHz,
- Transfery poszczególnych bajtów wyzwalane przerwaniami,
- Automatycznie przejście do trybu Slave w przypadku kolizji na magistrali (Arbitration-lost interrupt),
- Przerwanie zgłaszane, gdy zostanie wykryty adres urządzenia w trybie Slave,
- Automatyczne wykrywanie stanu zajętością magistrali,
- Obsługa adresów 7 i 10-cio bitowych.

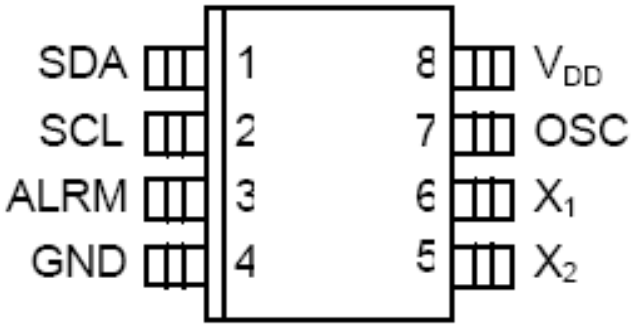
# Schemat blokowy modułu TWI



# Zegar czasu rzeczywistego

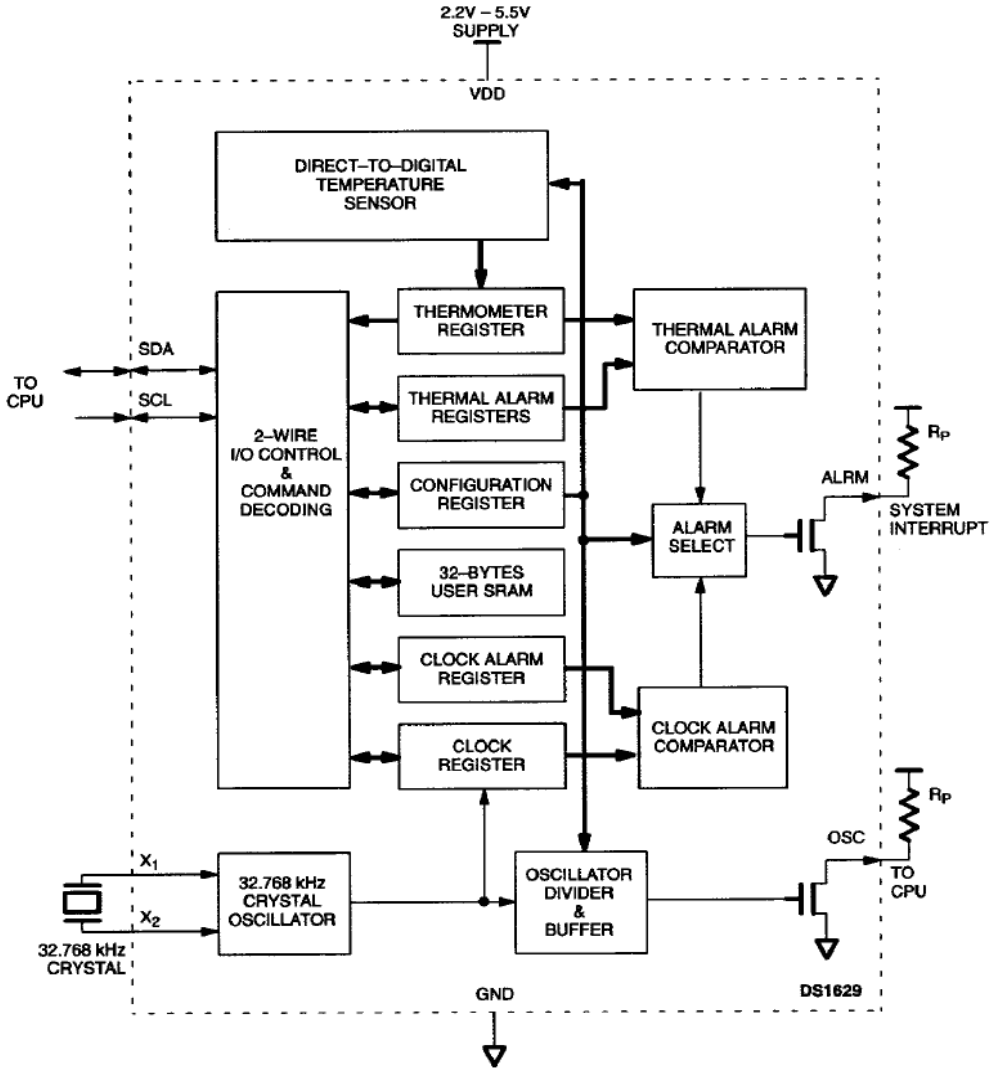
## Cechy układu DS1629:

- Zegar czasu rzeczywistego,
- Pomiar temperatury -55 – 125 C,
- Rozdzielczość termometru: 9 bitów,
- Dokładność termometru +/- 2 C,
- Układ termostatu,
- 32 bajty pamięci SRAM,
- Zasilanie 2,2 – 5,5 V,
- Interfejs zgodny ze standardem I2C (400 kHz).

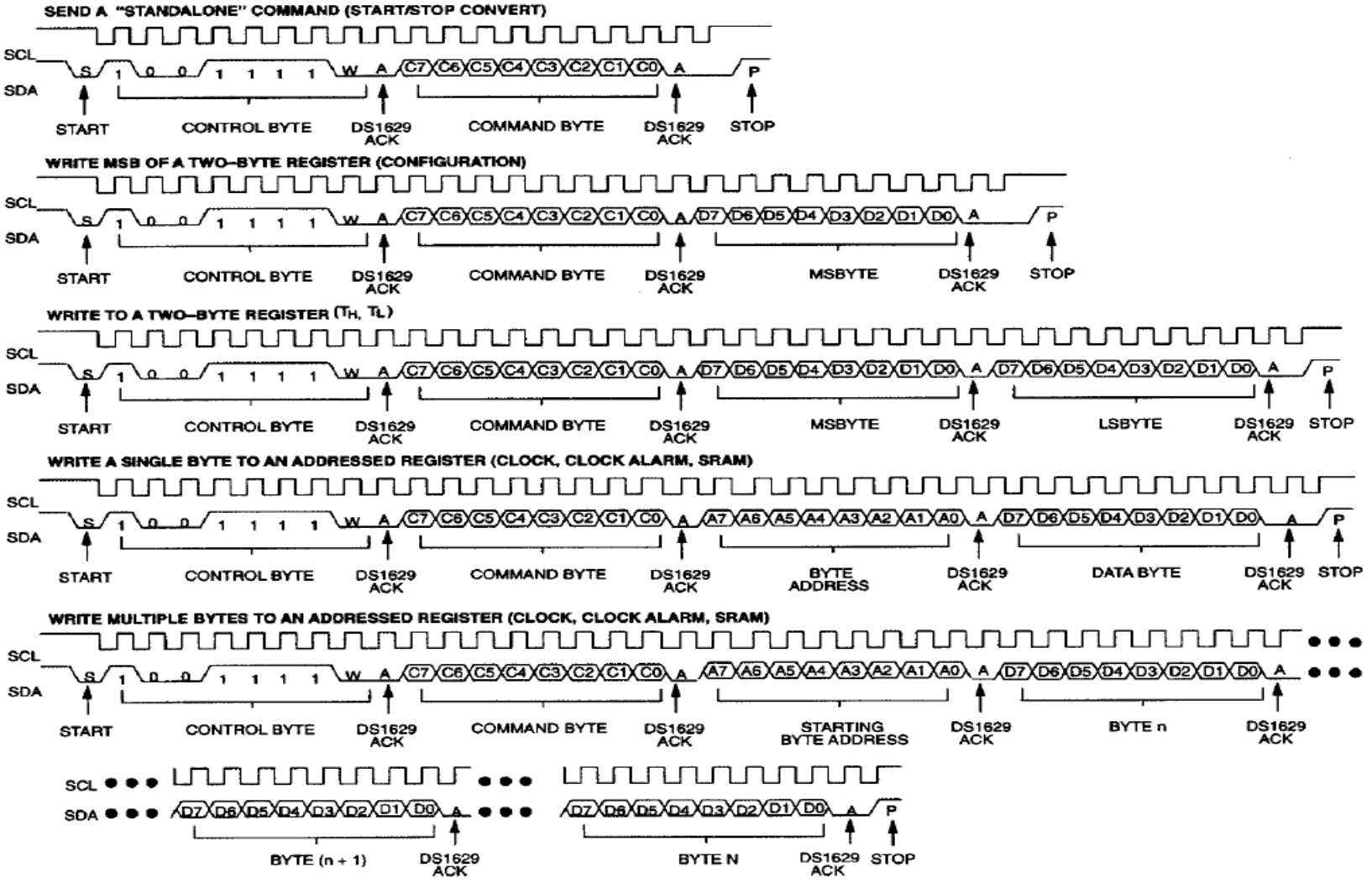



DS1629S 8-Pin SOIC
(150-mil)

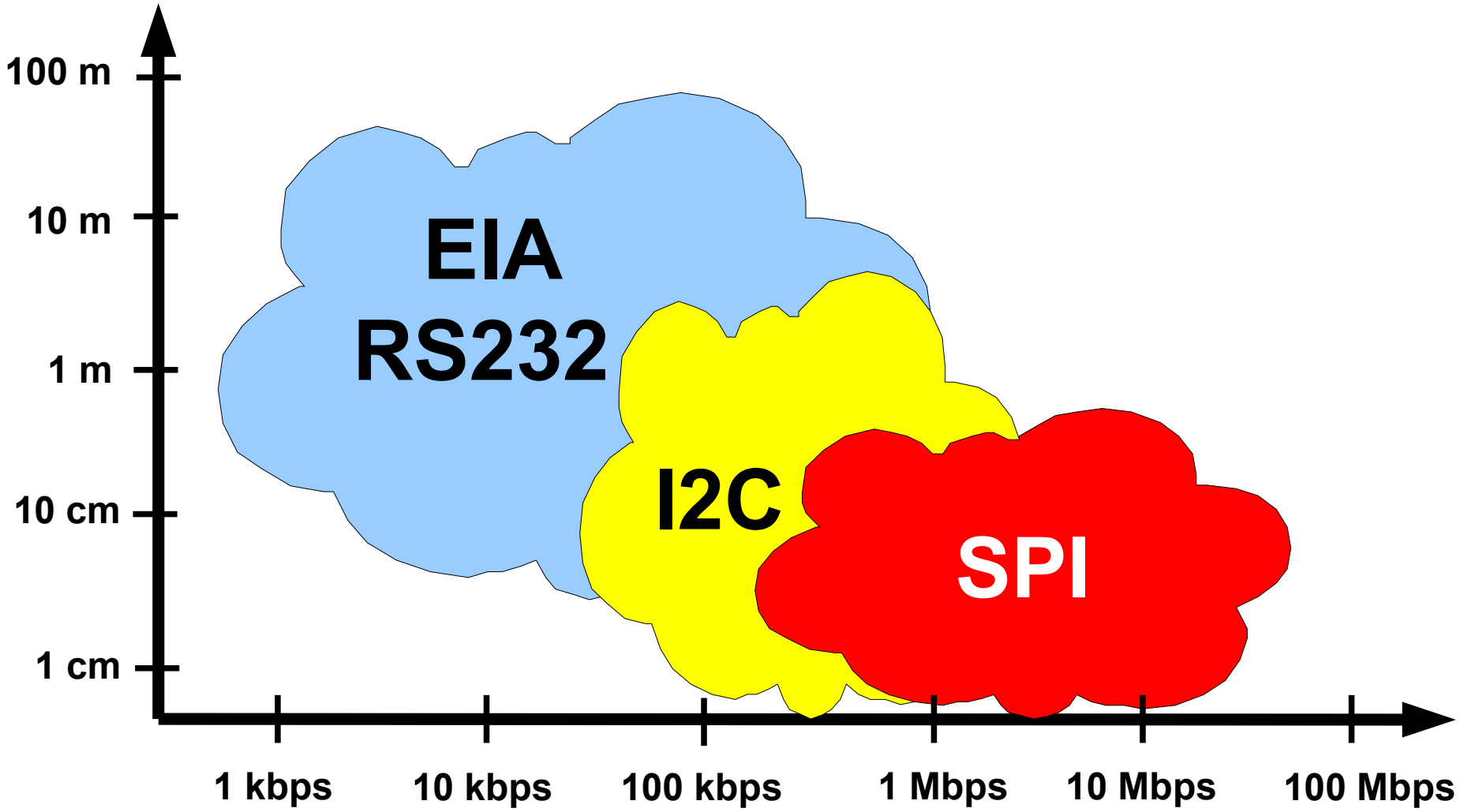
# Zegar czasu rzeczywistego



# Transmisja z wykorzystaniem interfejsu I2C



# Interfejsy szeregowe - podsumowanie

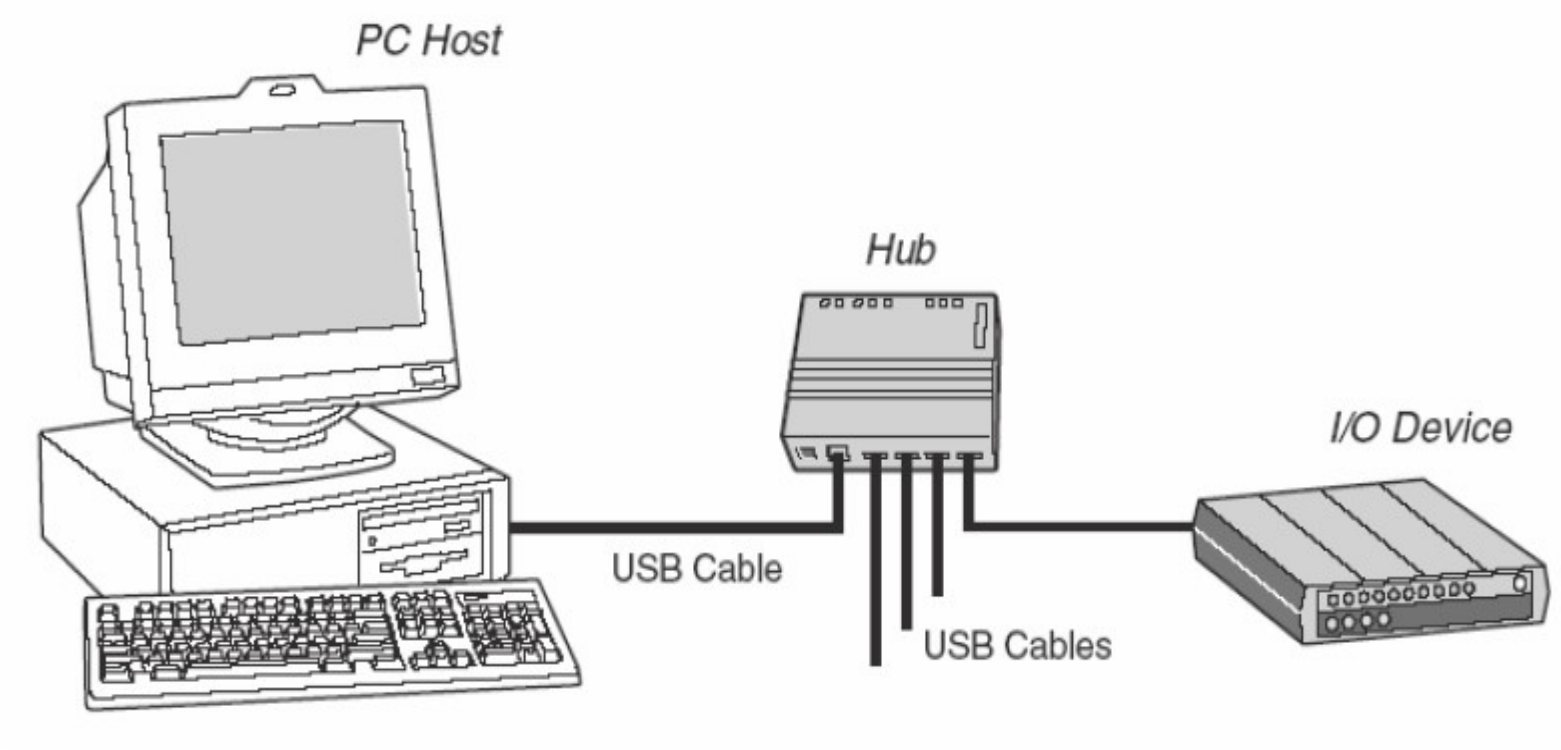


# Daty egzaminów

- Egzamin #1 (sala A1-A4) – 23.06.2023 9.15-10.00
  Egzamin #2 (sala A1-A4) – 30.06.2023 9.15-10.00
  Egzamin #3 (sala A1) – ~11.09.2023 9.15-10.00

# Magistrala USB
# (Universal Serial Bus)

# Magistrala USB

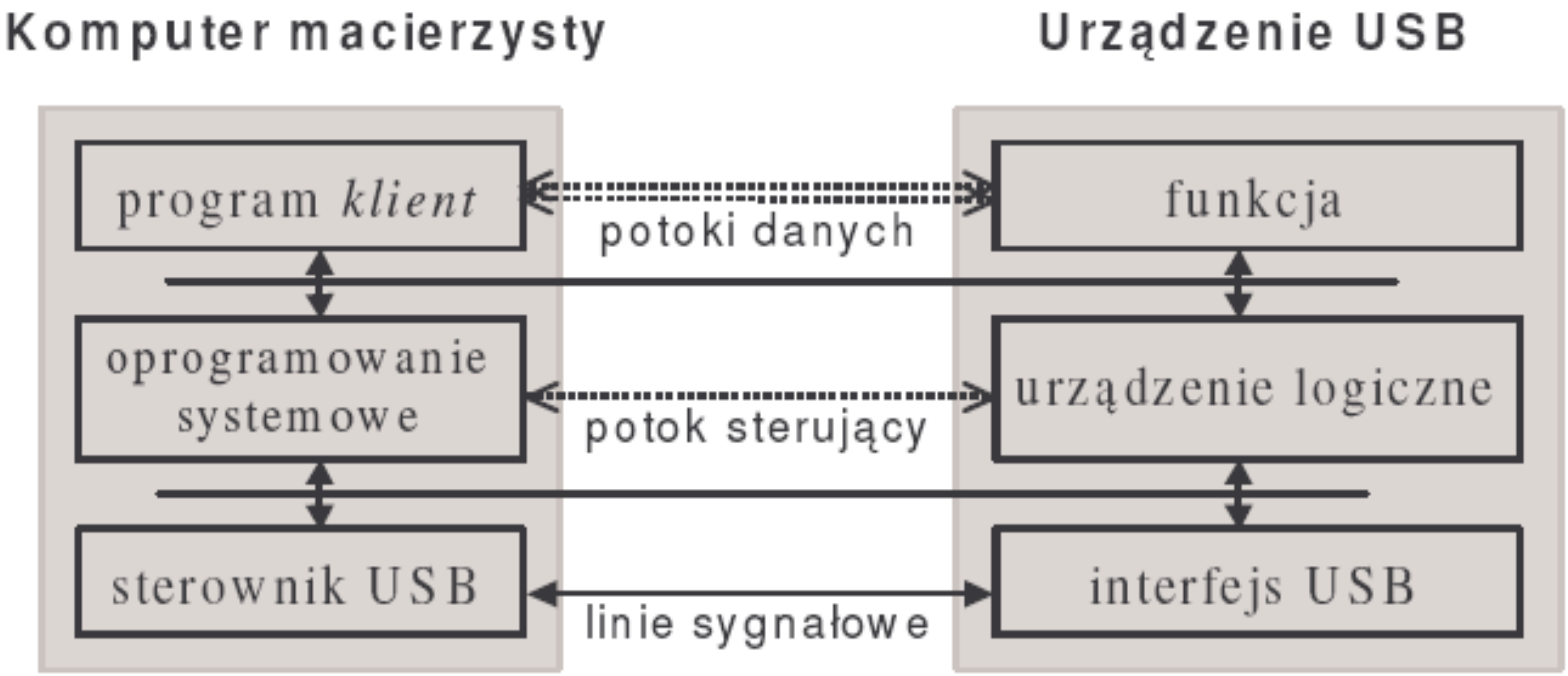


# Cechy magistrali USB

- Asynchroniczna, szeregowa, różnicowa transmisja danych,
- Automatyczna detekcja dołączenia/odłączenia urządzenia oraz automatyczna konfiguracja,
- Pojedyncze, ustandaryzowane złącze,
- Możliwość dołączenia do 127 urządzeń do magistrali,
- Automatyczna detekcja i korekcja błędów,
- Szybkość transmisji danych:
  - LOW 1.5 Mb/s, specyfikacja USB >1.1,
  - FULL 12 Mb/s, specyfikacja USB >1.1,
  - HIGH 480 Mb/s, specyfikacja USB 2.0,
  - **Specyfikacja USB 3.0 => 5 Gb/s.**

| TRANSMISJA | PRZYKŁADOWE ZASTOSOWANIA | CZĘSTOTLIWOŚĆ PRACY INTERFEJSU USB |
|---|---|---|
| **WOLNA** 10 - 100 kb/s | Klawiatura, mysz, manipulatory. | **mała - 1,5 Mb/s** |
| **ŚREDNIA** 500 kb/s - 10 Mb/s | Urządzenia do transmisji danych po liniach telefonicznych, urządzenia audio. | **pełna - 12 Mb/s** |
| **SZYBKA** 25 - 400 Mb/s | Urządzenia wideo, pamięci dyskowe. | **duża - 480 Mb/s** |

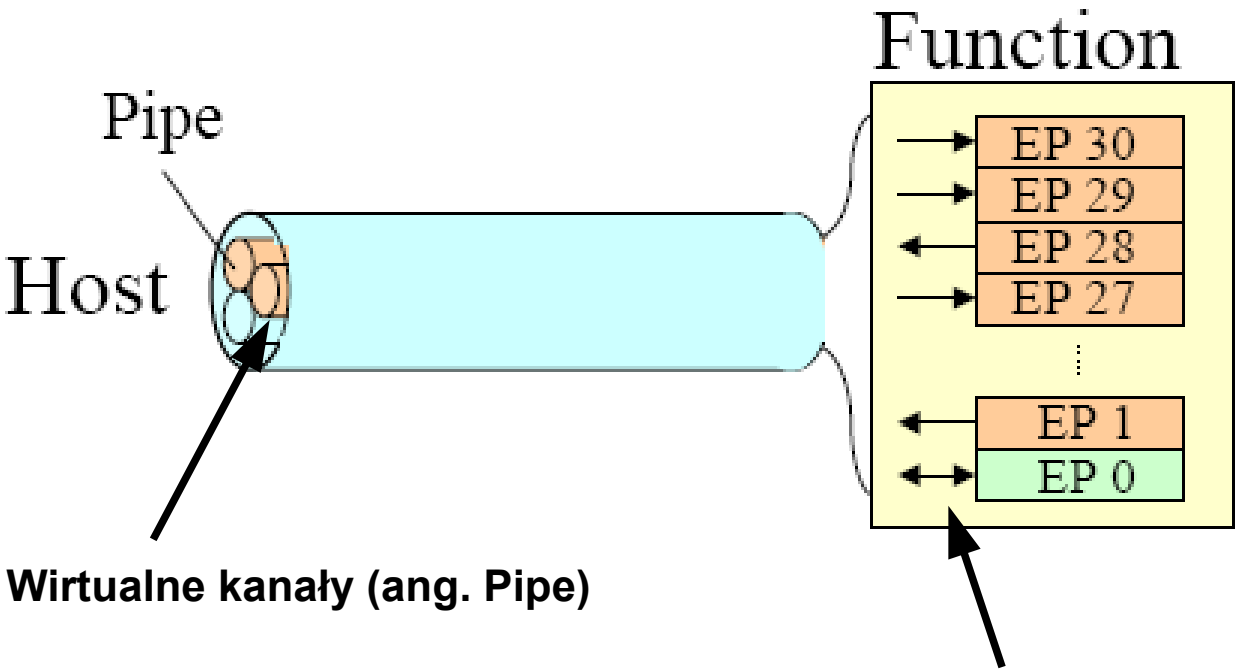
# Struktura warstwowa magistrali USB




**Magistrala USB zbudowana jest na bazie architektury typu gwiazda.**
Model systemu USB składa się z trzech warstw:

- warstwa fizyczna,
- warstwa logiczna,
- warstwa funkcjonalna.

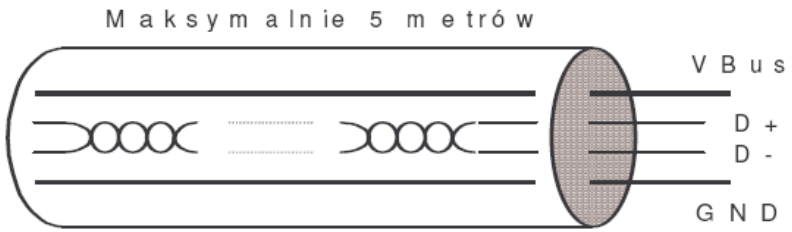
# Przepływ danych w systemie USB



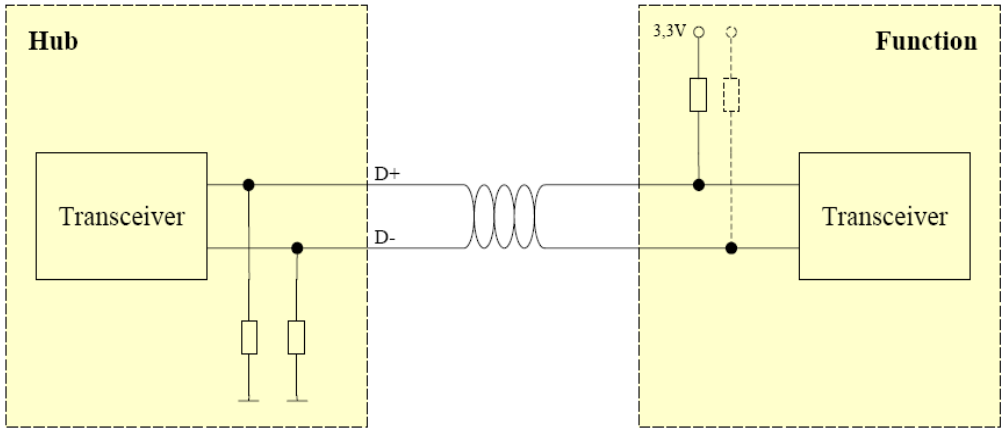

**Wirtualne kanały (ang. Pipe)**

Kanał sterujący (EP0) oraz
Kanały danych EP1 – EP30 (ang. End Points)

# Warstwa fizyczna



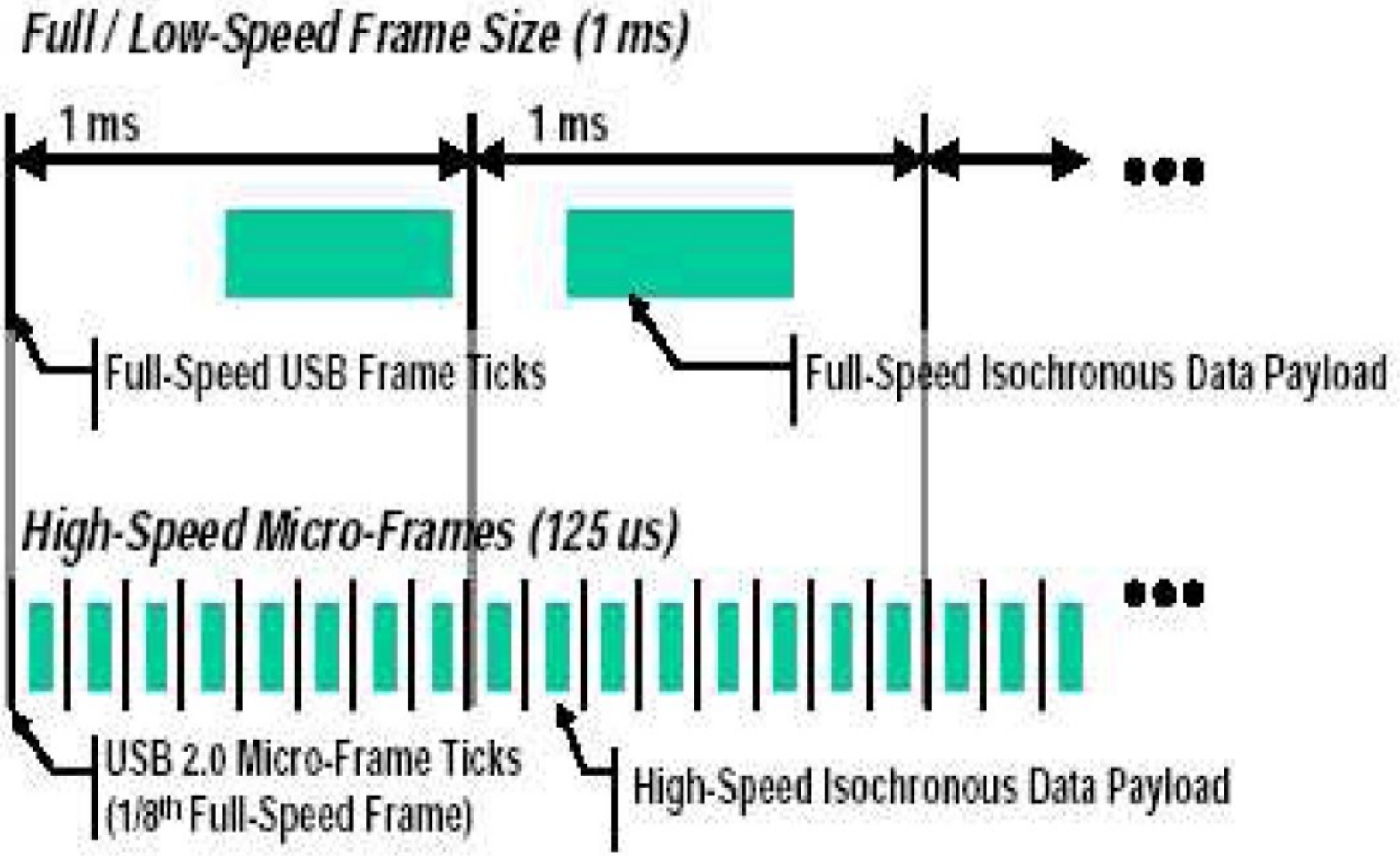

Transmisja różnicowa, typu half-duplex. Dwa dodatkowe przewody zasilające 5 V/500 mA




Złącza typu mini USB

Złącza USB typu “A” i “B”

# Zależności czasowe ramek USB



# Rodzaje transferów

| Type | Important attributes | Max size LS | Max size FS | Max size HS | Examples |
|---|---|---|---|---|---|
| Interrupt | Quality + time | 8 | 64 | 3072 | Mouse, keyboard |
| Bulk | Quality | - | 64 | 512 | Printer, scanner |
| Isochronous | time | - | 1023 | 3072 | Audio, video |
| Control | Quality + time | 8 | 64 | 64 | System control |

# Proces konfiguracji

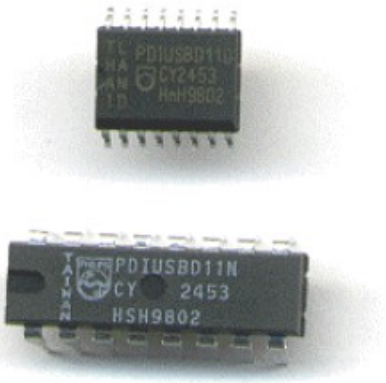
**Enumeracja (ang. Enumeration)** – konfiguracja urządzeń przeprowadzana po dołączeniu lub odłączeniu nowego urządzenia od magistrali. Proces konfiguracji przeprowadzany jest przez urządzenie nadrzędne (Master). Master przypisuje indywidualne adresy do urządzeń oraz ustanawia podstawowe parametry transmisji:

- Adres urządzenia w przestrzeni USB,
- Rodzaj transferu,
- Kierunek transmisji danych (read, write, read-write),
- Rozmiar przesyłanych pakietów,
- Szybkość transmisji,
- Adresy buforów używanych przez sterowniki urządzenia,
- Prąd pobierany przez urządzenie.

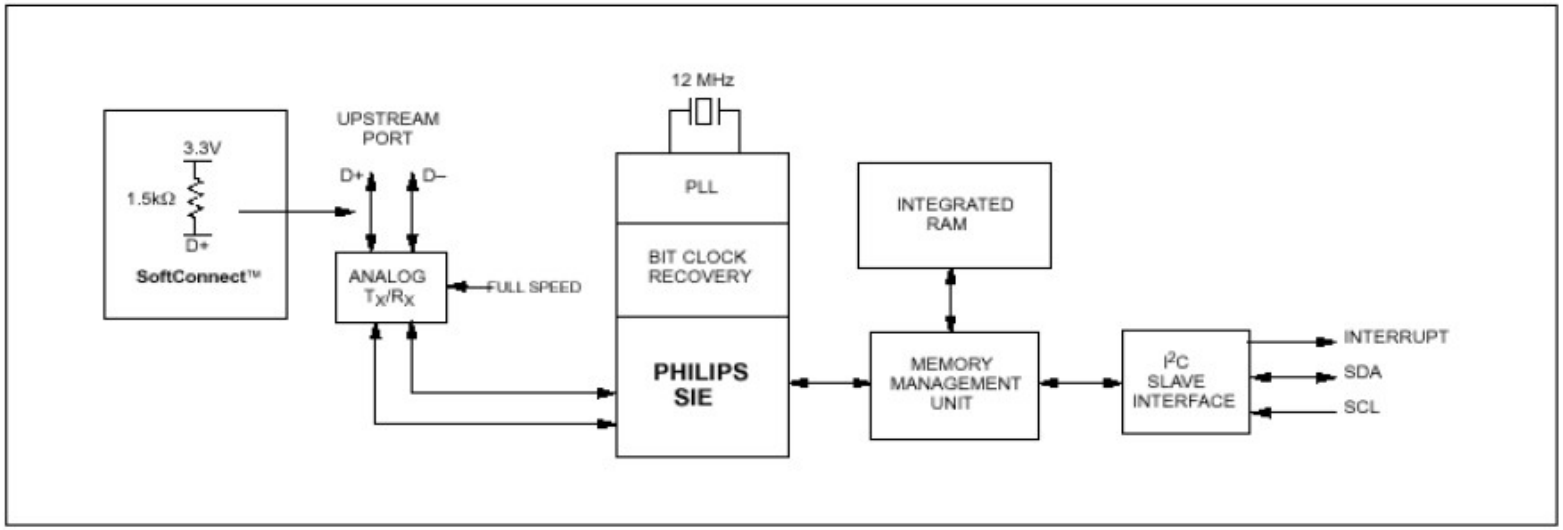
# Koncentratory USB

# Konwerter USB - I2C

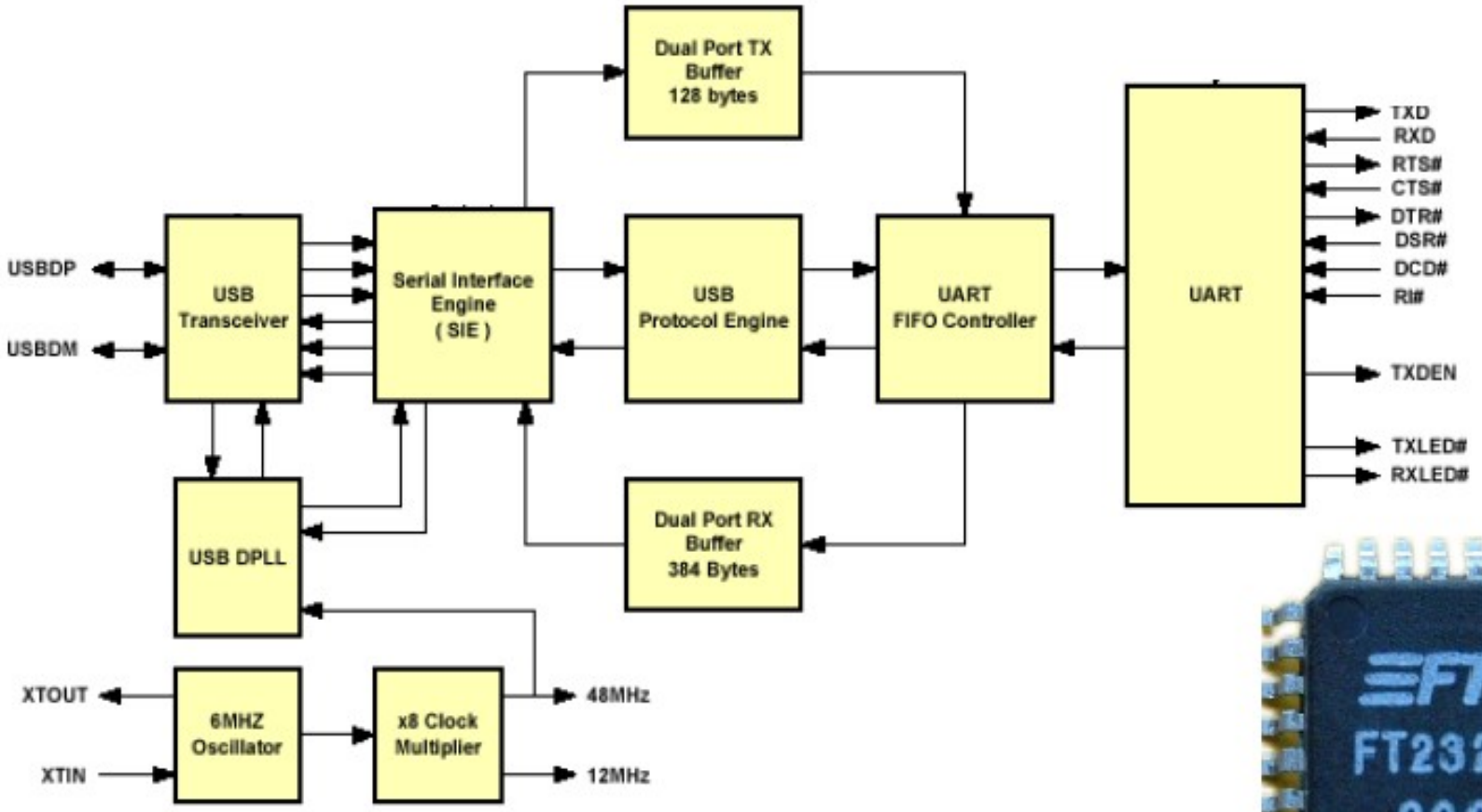
## Philips PDIUSBD11 (USB to I2C)




**BLOCK DIAGRAM**



# Konwerter EIA 232-USB




# USB i procesory ColdFire a USB

## Low\Full speed:

| | |
|---|---|
| MCF 527X (72-75) | 66 – 166 MHz |
| MCF 5221X (72-75) | 80 MHz |
| MCF 5222X (72-75) | 80 MHz |
| MCF 527X (72-73) | 240 MHz |
| 68HCS08JW32 | 8 MHz |

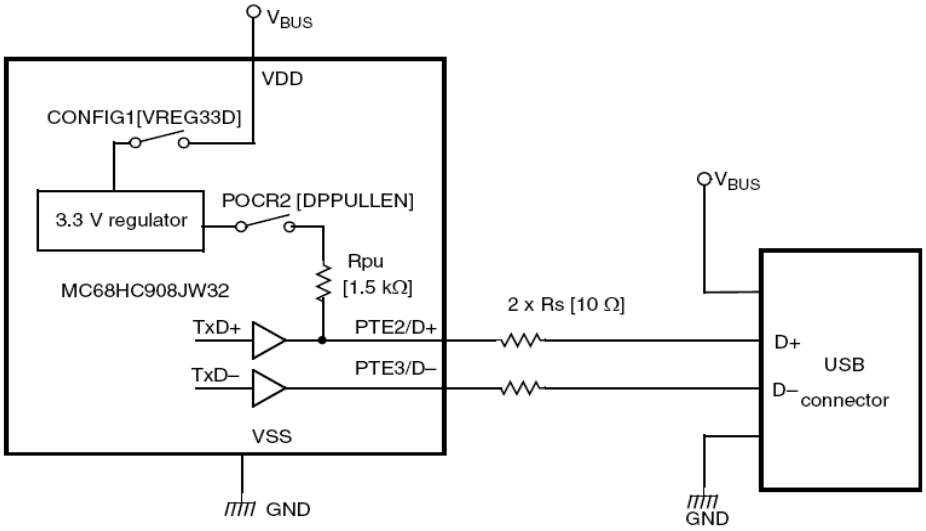
## High Speed:

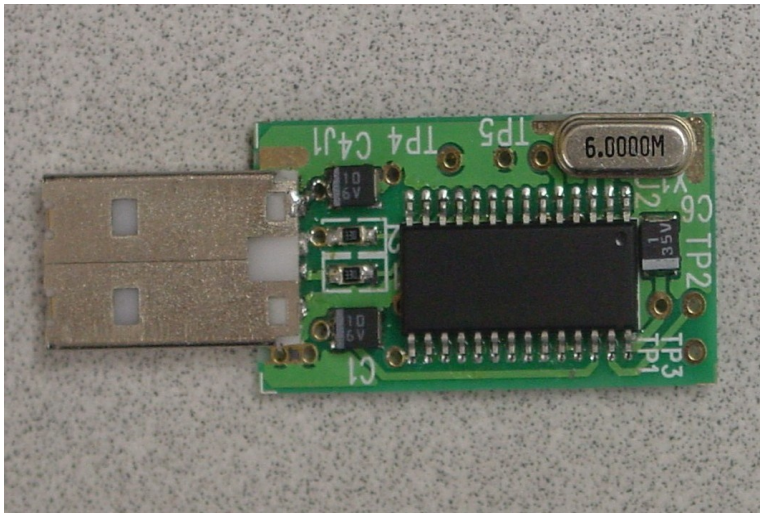
| | |
|---|---|
| MCF 547X (72-75) | 200 –266 MHz |
| MCF 548X (82-85) | 166 – 200 MHz |
| MCF 537X (77-79) | 240 Mhz |
| MCF 5253 | 140 MHz |

# Motorola 68HC908JW32

Cechy modułu USB procesora HC908:

- Interfejs zgodny ze standardem USB 2.0 full speed,
- 12 Mbps data rate,
- Wbudowany stabilizator napięcia 3.3 V,
- Endpoint 0 wyposażony w 8-bytowy bufor nadawczy i odbiorczy
- 64 bajtowy bufor endpoint współdzielony przez bufory końcowe 1-4.

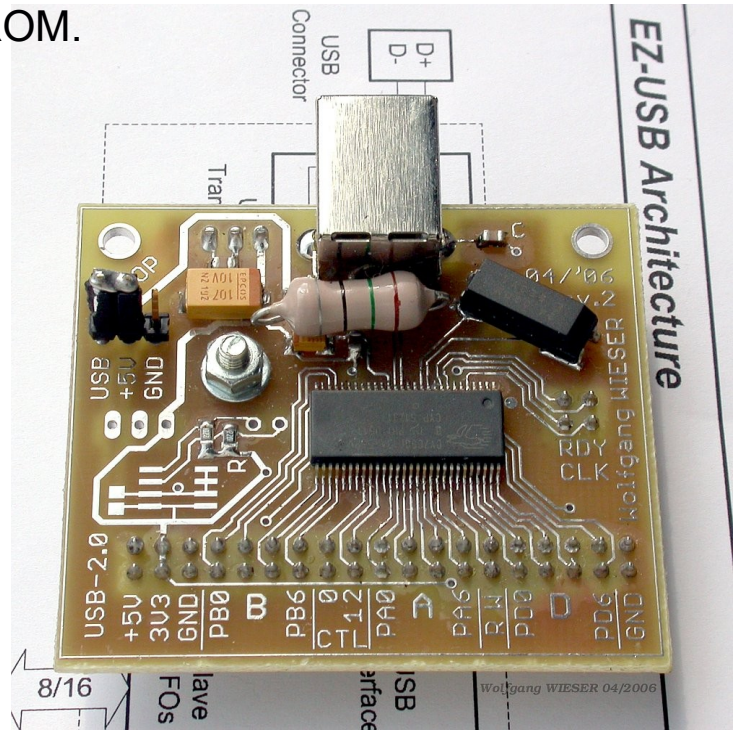


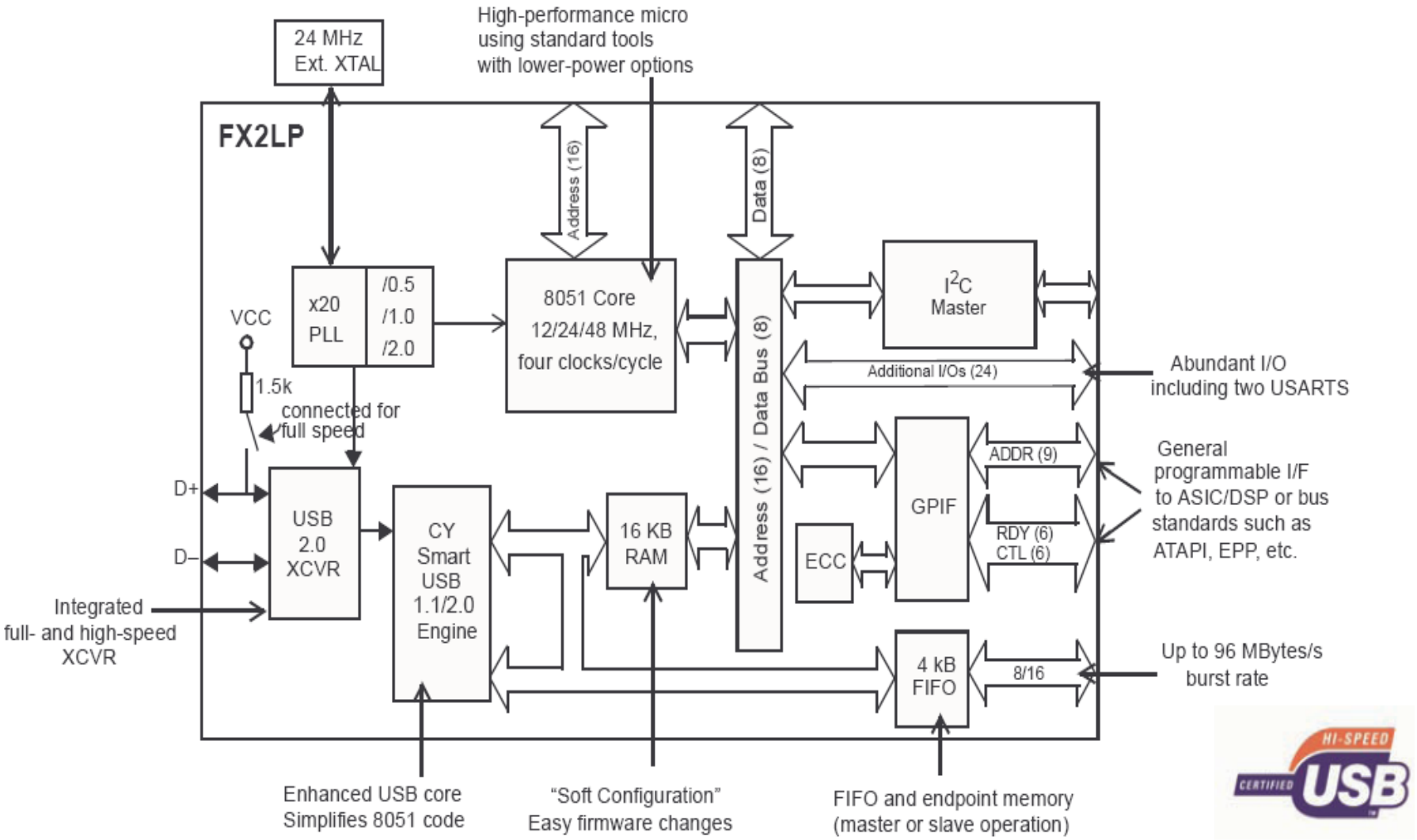
# Procesor Cypress CY7C68013A

## Cechy procesora CY7C68013:

- Interfejs zgodny ze standardem USB 2.0–USB-IF high speed,
- Rozbudowane jądro procesora rodziny **8051**,
- Zintegrowana pamięć programu 16 kB (RAM)
  - Pamięć ładowana z USB,
  - Pamięć ładowana z zewnętrznej pamięci EEPROM.
- Cztery programowalne bufory końcowe (BULK/INTERRUPT/ISOCHRONOUS)
- Dodatkowy 64 bajtowy endpoint (BULK/INTERRUPT),
- 8- lub 16-bitowy interfejs zewnętrzny,
- Kanał DMA, GPIF (General Programmable Interface)

# Procesor Cypress CY7C68013A

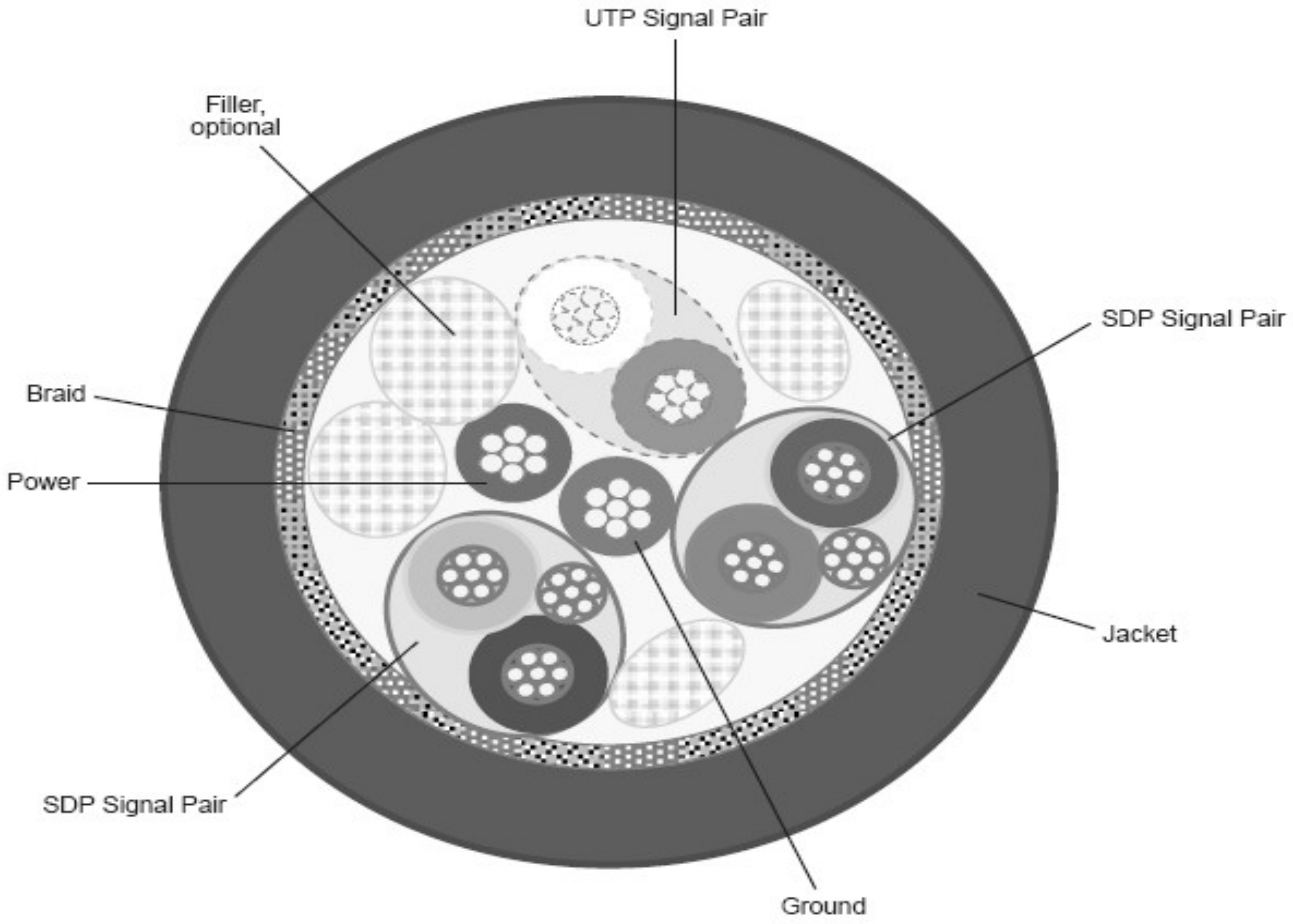



SuperSpeed USB 3.0 Specification Revolutionizes An Established Standard

# USB 3.0

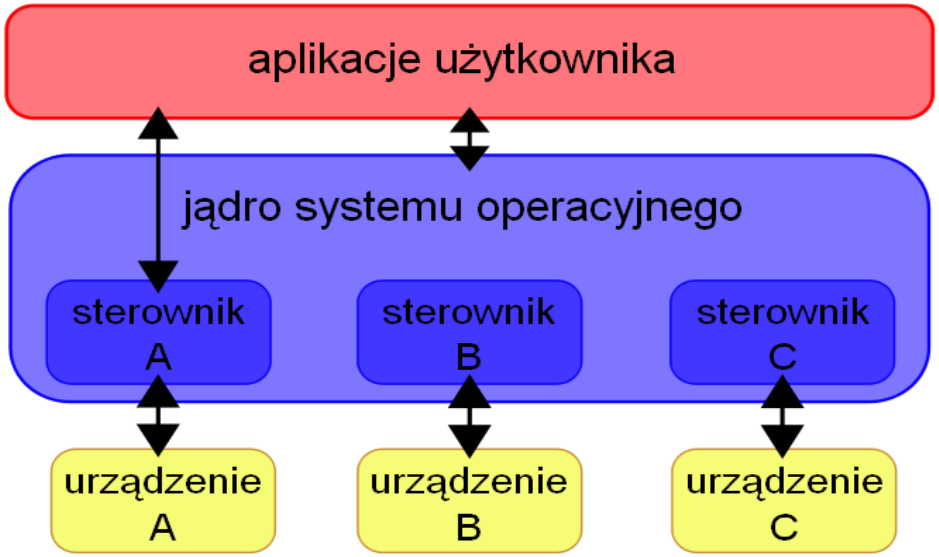
- Interfejs szeregowy, full-duplex
- Szybkość transmisji danych: 5 Gb/s (10 razy szybciej niż USB 2.0)
- Standard kompatybilny z USB 2.0 (sterowniki i złącza), jednak znacznie różniący się od USB 2.0
- Transmisja danych full-duplex, zasilanie
- Inteligentne zarządzanie poborem energii, mniejsze zużycie energii
- Warstwa łącza danych i fizyczna podobna do interfejsu PCI express 2.0

# Warstwa fizyczna USB 3.0



# Sterowniki urządzeń peryferyjnych

# Sterowniki urządzeń (1)




**Sterownik urządzenia (ang. driver) -** program lub fragment programu odpowiadający za dane urządzenie i pośredniczący pomiędzy nim, a resztą systemu komputerowego. Sterownik zwykle traktowany jest jako zestaw funkcji przeznaczonych do obsługi urządzenia peryferyjnego. Sterownik odwzorowuje pewne cechy urządzenia. Nazewnictwo funkcji oraz parametry przyjmowane i zwracane przez funkcje są zwykle narzucone przez system operacyjny. Sterowniki urządzeń dostępne w systemach operacyjnych udostępniają programiście interfejs API (Application Programing Interface), **bezpośredni dostęp do urządzenia jest zabroniony.**

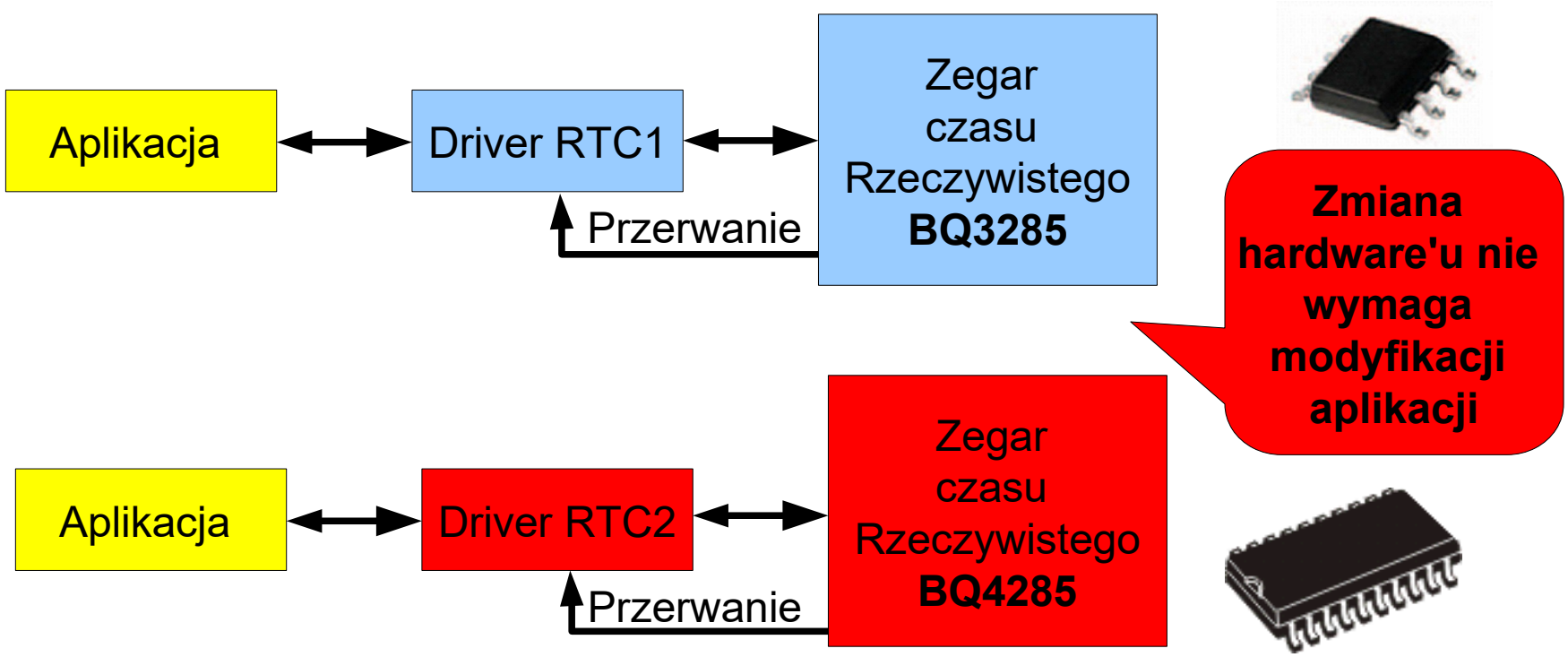
W przypadku systemów wbudowanych **aplikacje mogą się bezpośrednio odwoływać do urządzeń,** czasami trudno jest odróżnić aplikację od sterownika.

# Sterowniki urządzeń (2)

**Sterownik urządzenia peryferyjnego** (urządzenia peryferyjne wewnętrzne i zewnętrzne) udostępnia podstawowe funkcje pozwalające na łatwe korzystanie z danego urządzenia.

Sterownik urządzenia pozwala programiście „ukryć” dane urządzenie – dostarczając tylko zestaw funkcji umożliwiających sterowanie oraz wymianę danych z danym urządzeniem.

Pisząc sterownik urządzenia musimy pamiętać o procedurach obsługujących przerwania.



# Sterowniki – inicjalizacja urządzeń

**Sterownik urządzenia peryferyjnego - z**wykle implementowane są następujące funkcje:

**Device_Open ()** - funkcja wykorzystywana do inicjalizacji urządzenia.

Funkcja może przyjmować parametry jeżeli sterownik obsługuje więcej niż jedno urządzenie, np. dwa porty USART, których rejestry są dostępne pod innymi adresami bazowymi.

Funkcja może zwrócić wynik operacji związanej z inicjalizacją urządzenia lub deskryptor (wskaźnik) dający dostęp do danego urządzenia (do rejestrów urządzenia lub struktury umożliwiającej komunikację z nim).

Sterownik urządzenia może zostać „otworzony” przez kilka różnych aplikacji.

W takim przypadku należy zaimplementować, tzw. licznik odwołań do urządzenia. Licznik odwołań zwiększany jest przy każdym wywołaniu funkcji Open. Inicjalizacja urządzenia przeprowadzana jest tylko jeden raz.

**Device_Close () -** funkcja wywoływana, gdy aplikacja przestaje korzystać z danego urządzenia. Zadaniem funkcji jest bezpieczne wyłączenie urządzenia, np. w przypadku portów IO – konfiguracja jako porty wejściowe, USART – wyłączenie nadajnika/odbiornika, zamaskowanie przerwań.

Jeżeli funkcja Open została wykonana kilka razy, z urządzenia korzysta kilka aplikacji, należy jedynie zdekrementować licznik odwołań. Urządzenie wyłączane jest w przypadku, gdy licznik odwołań zmniejszy się do 0. Podobnie do funkcji Open, funkcja może przyjmować parametry oraz zwracać rezultat operacji.

Funkcje Open i Close powinny również konfigurować przerwania skojarzone z danym urządzeniem peryferyjnym.

# Sterowniki – komunikacja z urządzeniami

**ReadData Device_Read ()** - funkcja wykorzystywana do odczytywania danych z urządzenia, np. portu szeregowego. Funkcja do odczytu danych może być funkcją blokująca lub nie. Funkcja blokująca czeka, aż dane będą dostępne (możliwe jest wcześniejsze opuszczenie funkcji jeżeli upłynie określony okres czasu, a danych nadal nie ma - **timeout**). Timeout jest zwykle obliczany przez jeden z timerów procesora. W takim przypadku procesor czekając na nadejście danych może wykonywać inne obliczenia.

Funkcja Read może również korzystać z przerwań lub kanału DMA (Direct Memory Access). W takim przypadku dane wpisywane są do bufora. Gdy zgromadzi się odpowiednio duża ilość danych ustawiana jest flaga informująca o ich nadejściu lub zgłaszane jest przerwanie systemowe.

**Device_Write () -** funkcja wykorzystywana do zapisywania danych do urządzenia, np. do portu szeregowego. Funkcja do odczytu danych może być funkcją blokująca lub nie. Funkcja blokująca czeka, aż dane zostaną wysłane. Transmisja danych przez port szeregowy również zajmuje dużo czasu (przesłanie 1 znaku z szybkością 9600 bit/s zajmuje około 1 ms). W takim przypadku dane zgromadzone w buforze mogą być przesyłane przy wykorzystaniu przerwania – funkcja nieblokująca. Funkcja ustawia flagę informująca o zakończeniu transmisji. Wykorzystanie kanału DMA znacznie przyśpiesza wykonanie operacji.

Funkcje mogą zwracać rezultat wykonane operacji, np. przesłanie danych przez port USART wymaga potwierdzenia poprawności ich odbioru. W takim przypadku po wysłaniu danych uruchamiany jest odbiornik, który czeka na przesłanie potwierdzenia zgodnego w użytym protokołem transmisji danych.

# Sterowniki – funkcje pomocnicze

**DeviceStatus DeviceStatus ()** - funkcja wykorzystywana do odczytywania statusu urządzenia, np. sprawdzanie flagi Timer'a, USART'a, itp...

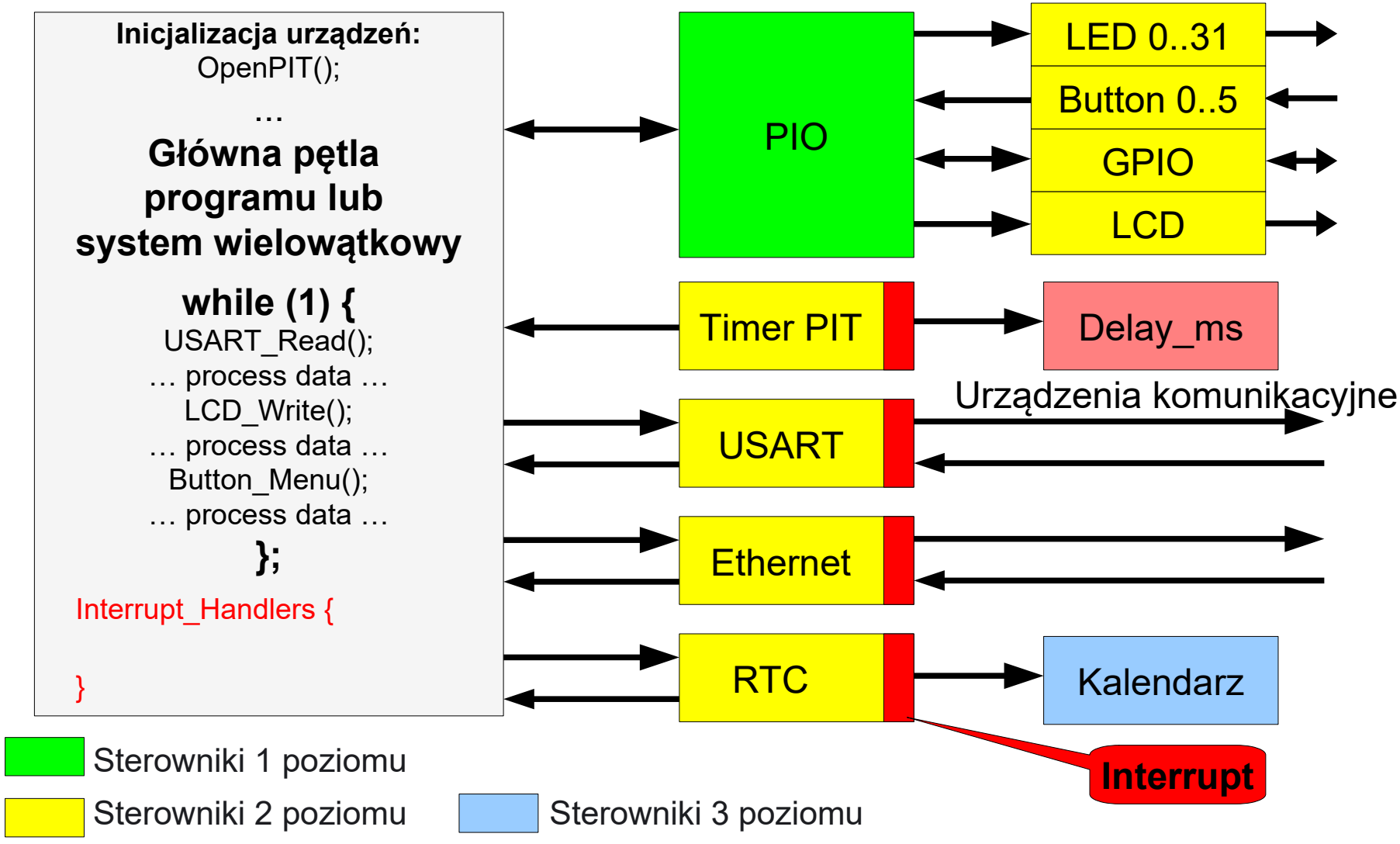
może zostać wywołana przez inną funkcję sterownika lub aplikację.

Wywołanie może nastąpić w funkcji blokującej (polling - ciągłe sprawdzanie stanu urządzenia – funkcja czeka na ustawienie lub wyzerowanie flagi) lub nieblokującej (sprawdzanie stanu wywoływane w funkcji przerwania).

**Device_INT_Handler() -** funkcja obsługująca przerwania od urządzeń peryferyjnych, np. handler do timer'a PIT.

**Device_WriteString () -** funkcja wykorzystywana do zapisywania ciągu znaków do urządzenia. Funkcja korzysta z funkcji **Device_Write()** zapisującej pojedynczy znak. Funkcja dziedziczy własności blokujące po funkcji niższego poziomu.

# Przykładow struktury sterowników systemu mikroprocesorowego



# Sterowniki systemu mikroprocesorowego

- Sterowniki urządzeń 1 poziomu:
  - Sterownik portu równoległego PIO,
- Sterowniki 2-go poziomu (korzystają ze sterowników 1-go poziomu):
  - Sterownik diod LED,
  - Sterownik klawiatury,
  - Sterownik wyświetlacza LCD,
  - Sterownik portów GPIO,
  - Sterownik Timera PIT,
  - Sterownik interfejsu USART,
  - Sterownik interfejsu Ethernet,
  - Sterownik zegara RTC.
- Sterowniki 3-go poziomu (korzystają ze sterowników 2-go poziomu):
  - Sterownik kalendarza.

# Przykładowe funkcje sterownika portu równoległego

PIO_Struct* **PIO_Open** (unsigned int *RegistersPointer, unsigned int PortMask);

void **PIO_Close** (unsigned int *RegistersPointer, unsigned int PortMask);

unsigned int **PIO_Read** (PIO_Struct* PoiterToPIO);

void **PIO_Write** (PIO_Struct* PoiterToPIO, unsigned int Data);

unsigned int **PIO_status** (PIO_Struct* PoiterToPIO);

## Funkcje zwracające status operacji:

unsigned int **PIO_Read** (PIO_Struct* PoiterToPIO, unsigned int *ReadData);

unsigned int **PIO_Write** (PIO_Struct* PoiterToPIO, unsigred int *DataToSend);

unsigned int **PIO_Status** (PIO_Struct* PoiterToPIO, unsigned int *DeviceStatus);

## Funkcje pomocnicze:

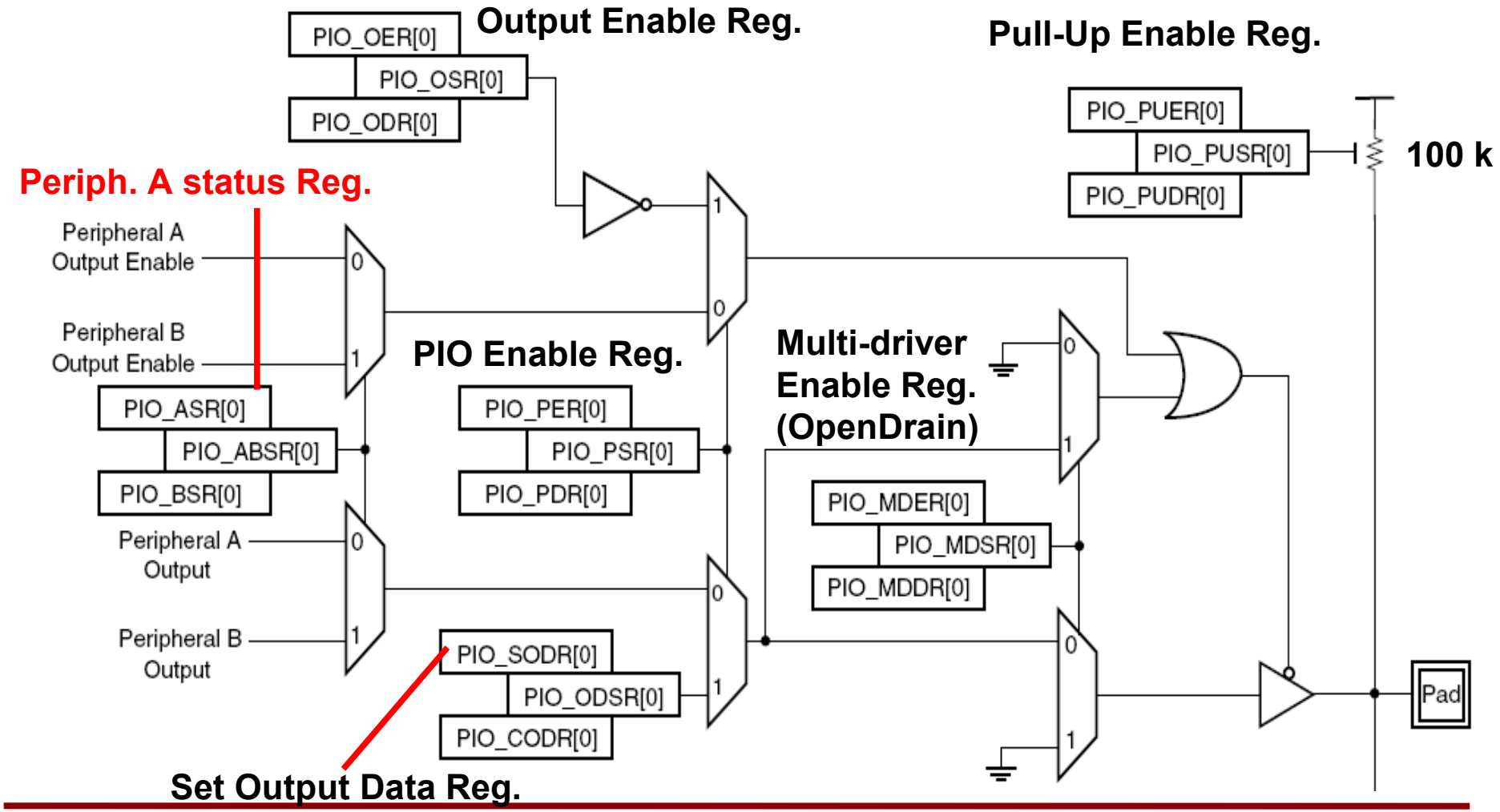
void **PIO_EnablePullUp** (unsigned int *RegistersPointer, unsigned int PortMask);

void **PIO_DisablePullUp** (unsigned int *RegistersPointer, unsigned int PortMask);

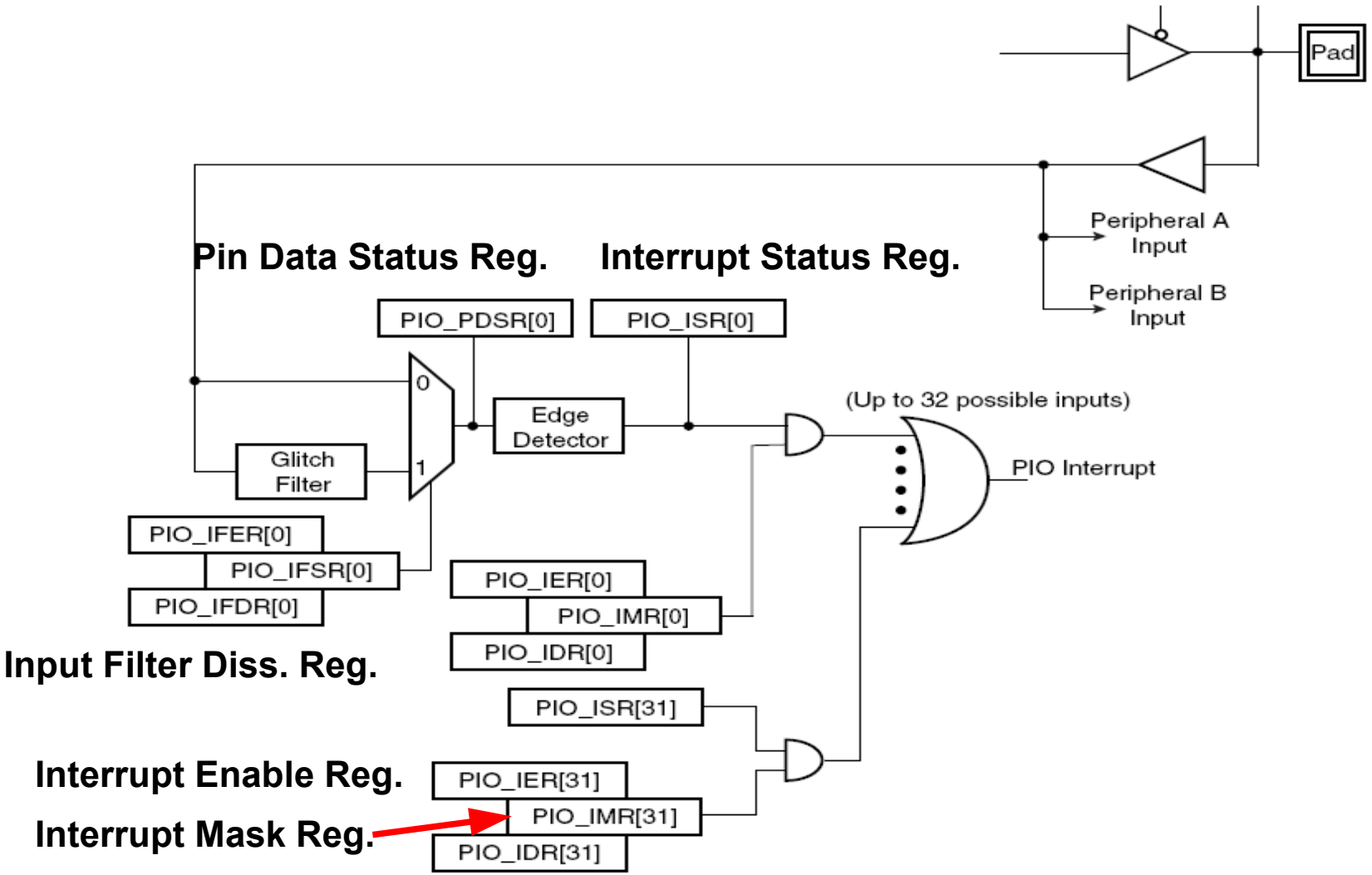
unsigned int **PIO_StatusPullUp** (unsigned int *RegistersPointer, unsigned int PortMask);

# Schemat blokowy portu I/O – sterowanie wyjściem

**Figure 31-3.** I/O Line Control Logic



# Schemat blokowy portu I/O – odczyt stanu wejścia



# Jak napisać sterownik ?

1. Przygotowanie struktury odzwierciedlającej rejestry danego urządzenia oraz masek pomocnych podczas operacji na rejestrach,

2. Opracowanie zmiennych pozwalających na sprawdzenie stanu danego urządzenia (np. czy urządzenie było już zainicjalizowane, czy ze sterownika korzysta jakieś urządzenie? Czy jedno?, jakie opóźnienie odmierza timer,...),

3. Opracowanie funkcji sterownika (Open, Close, Read, Write) oraz API służącego do komunikacji ze sterownikiem,

4. Opracowanie procedur obsługujących przerwania (wcześniejsze funkcje powinny na tym etapie działać – późniejsza lokalizacja problemów z włączonymi przerwaniami może być bardzo trudna lub nawet niemożliwa)

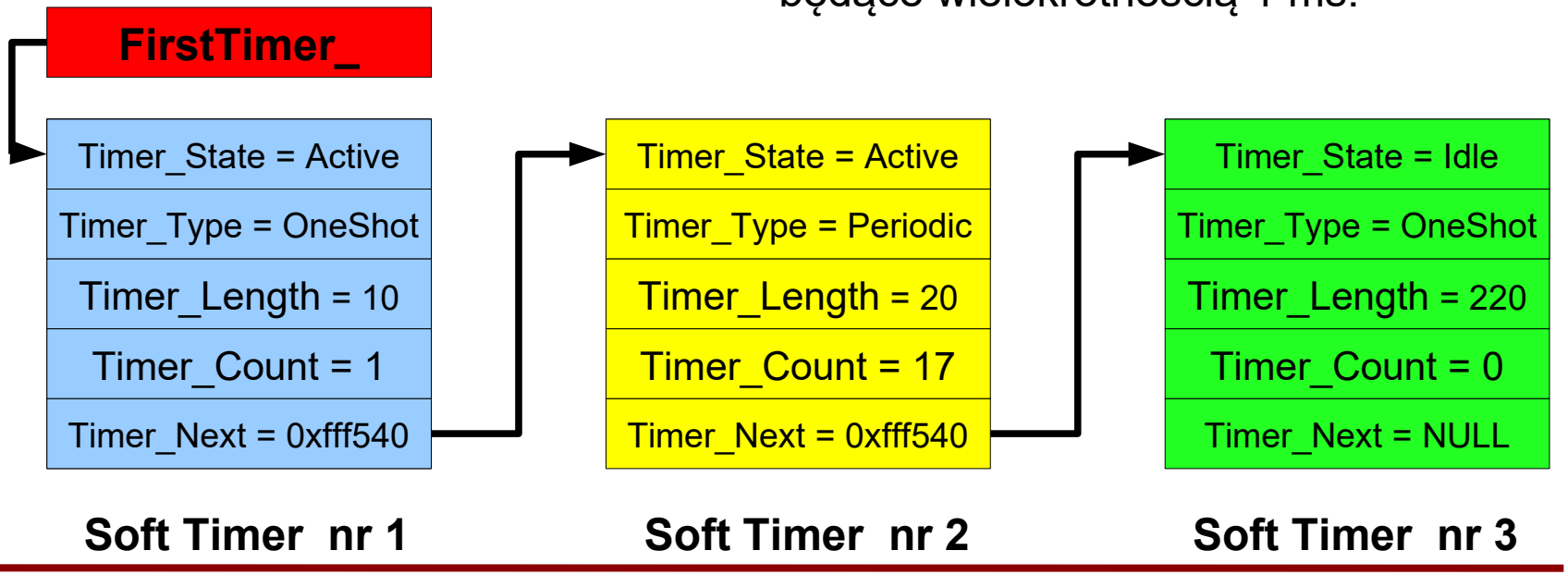
# Sterownik obsługujący kilka timer'ów

```
struct {
        TimerState      Timer_State;        /* current timer state */
        TimerType       Timer_Type;         /* current timer mode */
        unsigned int    Timer_Length;       /* length of delay - number of hardware timer ticks */
        unsigned int    Timer_Count;        /* number of ticks to expiere for each software timer */
        Timer *         Timer_next;         /* pointer to the next software timer */
}       FirstTimer, *FirstTimer_;
```

**Timer sprzętowy generuje przerwanie co 1 ms.**

**Timery programowe** mogą generować przerwania będące wielokrotnością 1 ms.

**Uporządkowana lista timer'ów**



# Przykładowy sterownik timer'a

```
enum TimerState {Idle, Active, Done};
enum TimerType {OneShot, Periodic};
typedef struct {
        TimerState      Timer_State;        /* current timer state */
        TimerType       Timer_Type;         /* current timer mode */
        unsigned int    Timer_Length;       /* length of delay - number of hardware timer ticks */
        unsigned int    Timer_Count;        /* number of ticks to expiere for each software timer */
        Timer *         Timer_next;         /* pointer to the next software timer */
}       Timer, *Timer_;

int Timer_Open(Timer_ * TPoin)              /* configure hardware and soft timer */
int Timer_Close(Timer_ * TPoin)             /* release hardware or soft timer */
int Timer_Start(unsigned int miliseconds, TimerType Type, Timer_ * TPoin) /* start timer */
int Timer_Wait_For (Timer_ * TPoin)         /* wait until timer fired */
void Timer_Cancel (Timer_ * TPoin)          /* turn off software timer */

static void Timer_INT (void);               /* hardware timer interrupt, e.g. 1 ms */
```

# Funkcje sterownika Timer'a programowego (1)

```
int Timer_Start (unsigned int miliseconds, TimerType Type, Timer_ * TPoin){
  if (Tpoin->Timer_State != Idle)
        return -1;
  Tpoin->Timer_State = Active;
  Tpoin->Timer_Type = Type;
  Tpoin->Timer_Length = miliseconds / MSPERTICK;     /* delay in ms */
  AddTimerToList (Tpoin);                            /* add pointer to the previous timer structure */
  return 0;
}

void Timer_Cancel (Timer_ * TPoin){
  if (Tpoin->Timer_State == Active)
        RemoveTimerFromList (Tpoin);
  Tpoin->Timer_State = Idle;
}
```

# Funkcje sterownika Timer'a programowego (2)

```
int Timer_Wait_For (Timer_ * TPoin){
        if (Tpoin->Timer_State != Active)
                return -1;
        while (Tpoin->Timer_State != Done);
        if (Tpoin->Timer_Type = Periodic){
                Tpoin->Timer_State = Active;
                AddTimerToList (Tpoin);
        }
        else
        {
                Tpoin->Timer_State = Idle;
        }
        return 0;
}
```

# Obsługa przerwania od timera sprzętowego

static void Timer_INT (void){ /* hardware timer interrupt, e.g. 1 ms */

0. Obsługa timera sprzętowego (reinicjalizacja t = 1 ms, potwierdzenie przerwania, itd...)
1. Sprawdź listę timerów,
2. Zdekrementuj pola Timer_Count,
3. Jeżeli Timer_Count równe 0 i TimerType = OneShot usuń timer z listy,
4. Jeżeli Timer_Count równe 0 i TimerType = Periodic uruchom timer ponownie, Timer_Count = Timer_Length.
5. Modyfikacja flagi od danego timer'a (Timer_Fired) lub wygenerownie przerwania programowego od danego timera.

}

# Sterowniki, a język C++

```
enum TimerState { Idle, Active, Done };
enum TimerType { OneShot, Periodic };
class Timer {
        public:
                Timer ();
               ~Timer ();

                int Start (unsigned int miliseconds, TimerType = OneShot);
                int Wait_For ();
                void Cancel ();

                TimerState      State;
                TimerType       Type;
                unsigned int    Length;


                unsigned int    Count;
                Timer *         pNext;
        private:
                static void INT ();
};
```

# Przykładowe pytania (1)

- Interfejs I2C jest interfejsem:

- Równoległym,
- Szeregowym,
- Umożliwiającym transmisję danych na duże odległości rzędu dziesiątek metrów,
- Umożliwiającym transmisję danych z szybkością kilku Mbps,
- Umożliwiającym transmisję Master – Slave,
- Umożliwiającym transmisję Multi Master – Slave,
- Umożliwiającym transmisję Master – Multi Slave,
- W którym ramka danych zawiera bit startu oraz bit stopu,
- W którym ramka danych zawiera bit parzystości,
- Umożliwia transmisję od 5 do 9 bitów w jednej,
- Umożliwia adresowanie urządzeń przy użyciu 8-bitowego adresu,
- Umożliwia adresowanie urządzeń przy użyciu 10-bitowego adresu
- Wymaga konwersji napięć odpowiadających przesyłanym symbolom MARK i SPACE określonych w standardzie,
- Pozwala na transmisje danych typu Full-duplex, Half-duplex, itd...

# Startup File

# Structure of Startup File

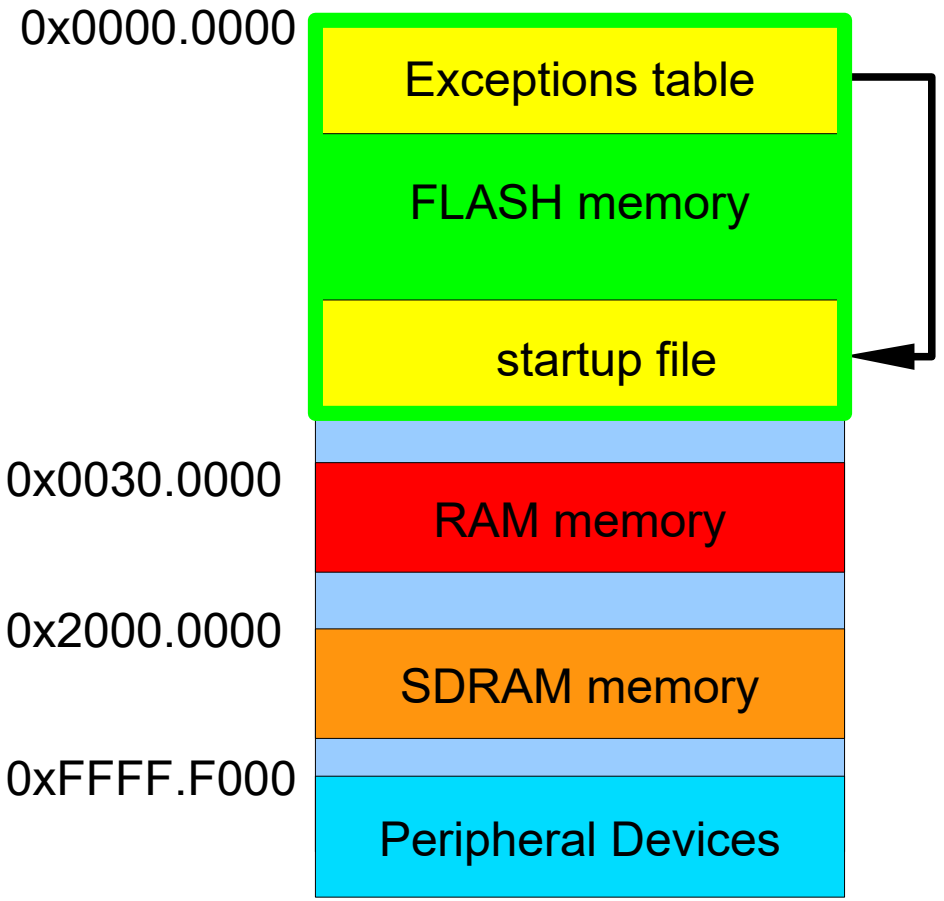
**Startup file** code is executed by processor just after reset. The code is responsible for configuration of basic peripheral devices and modules of processor.

Startup file is usually written in assembler because the code requires access to all processor resources that cannot be accessed from higher level languages. The code is executed before high level code (e.g. C/C++). Startup file is responsible for:

- Allocation of memory and configuration of stack pointers for different modes of operation (user, supervisor, IRQ, FIQ, etc...),
- Configuration of memory (remap FLASH memory, activate SRAM/DRAM, clean memory),
- Initialise exception vectors,
- Copy of Operating System or application code to memory,
- Initialise global variables in RAM (copy data from ROM, init variables with 0s),
- Configure peripheral devices,
- Initialise interrupt controller,
- Change processor mode if required,
- Call int main (void) function.

# Structure of Startup File (2)

```
RESET vector (address 0)
.section .text
reset_handler:
ldr     pc, =_low_level_init
/*       Initialization...          */
_low_level_init:


_stack_init:
_init_data:
_init_bss:

_branch_main:
```



# Structure of Startup File (3)

Program in the main loop cannot be finished. Processor cannot execute return or exit functions.

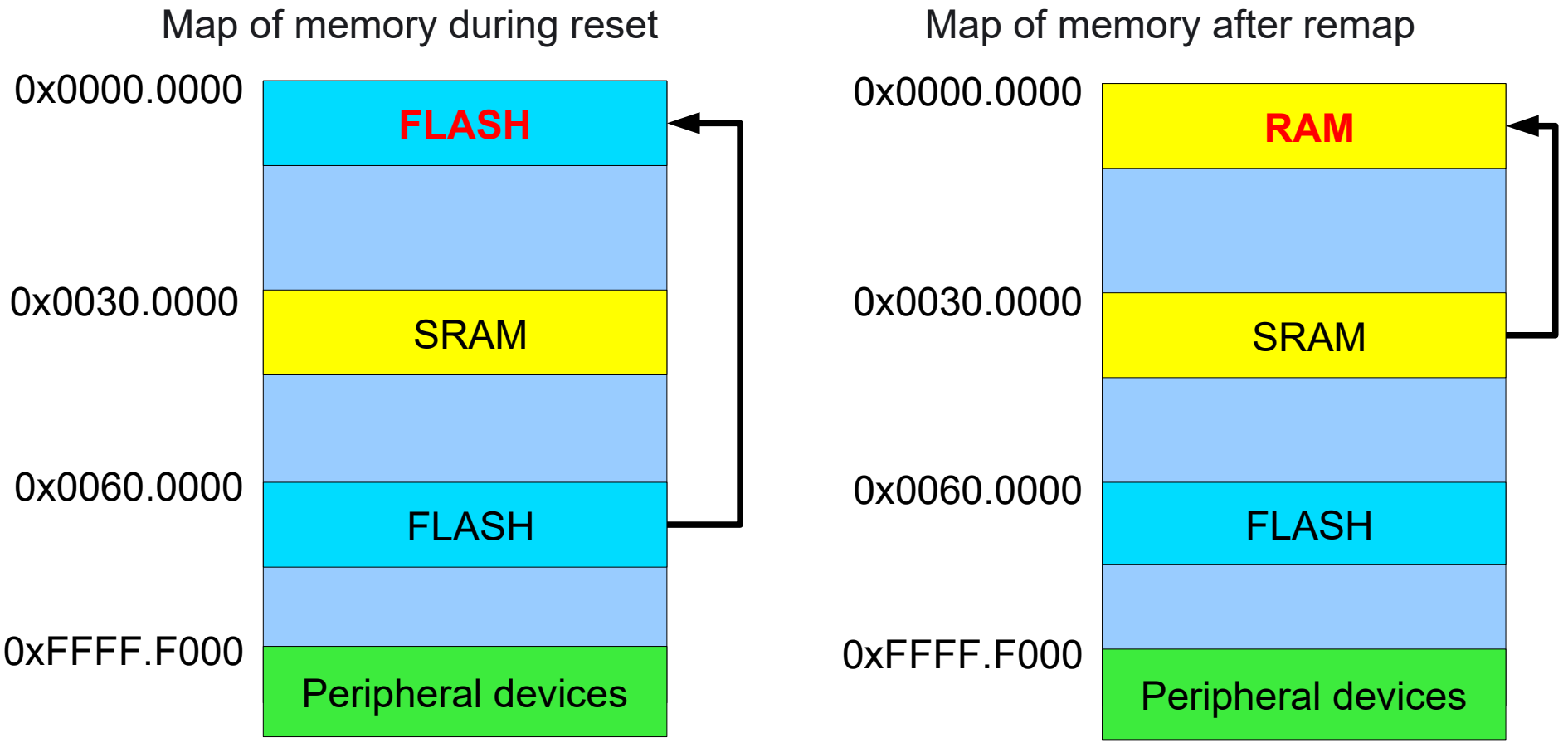
```
...
...
_branch_main:
        ldr      r0, =main
        mov       lr, pc
        bx       r0
...
...
void main (int) {
                While (1)
                {

                        main program

                }
        return 0;
}
```

# Structure of Startup File (4)

**Configuration of essential devices, required for processor operation:**

- Configuration of reference clock module (PLL). After reset processor operates with “slow clock” (internal RC generator),
- Configuration of memory controller (FLASH, RAM) – configure number of WaitStates, base address,
- Remap memory FLASH<->SRAM,
- Configure Watch-Dog timer (after reset Watch-Dog is active),
- Configure AIC module (assign default interrupt handlers),
- Initialise stack pointers for different modes of operation (user, IRQ, FIQ,...),
- Unblock NRST input (external reset).

# Remapping of Memory

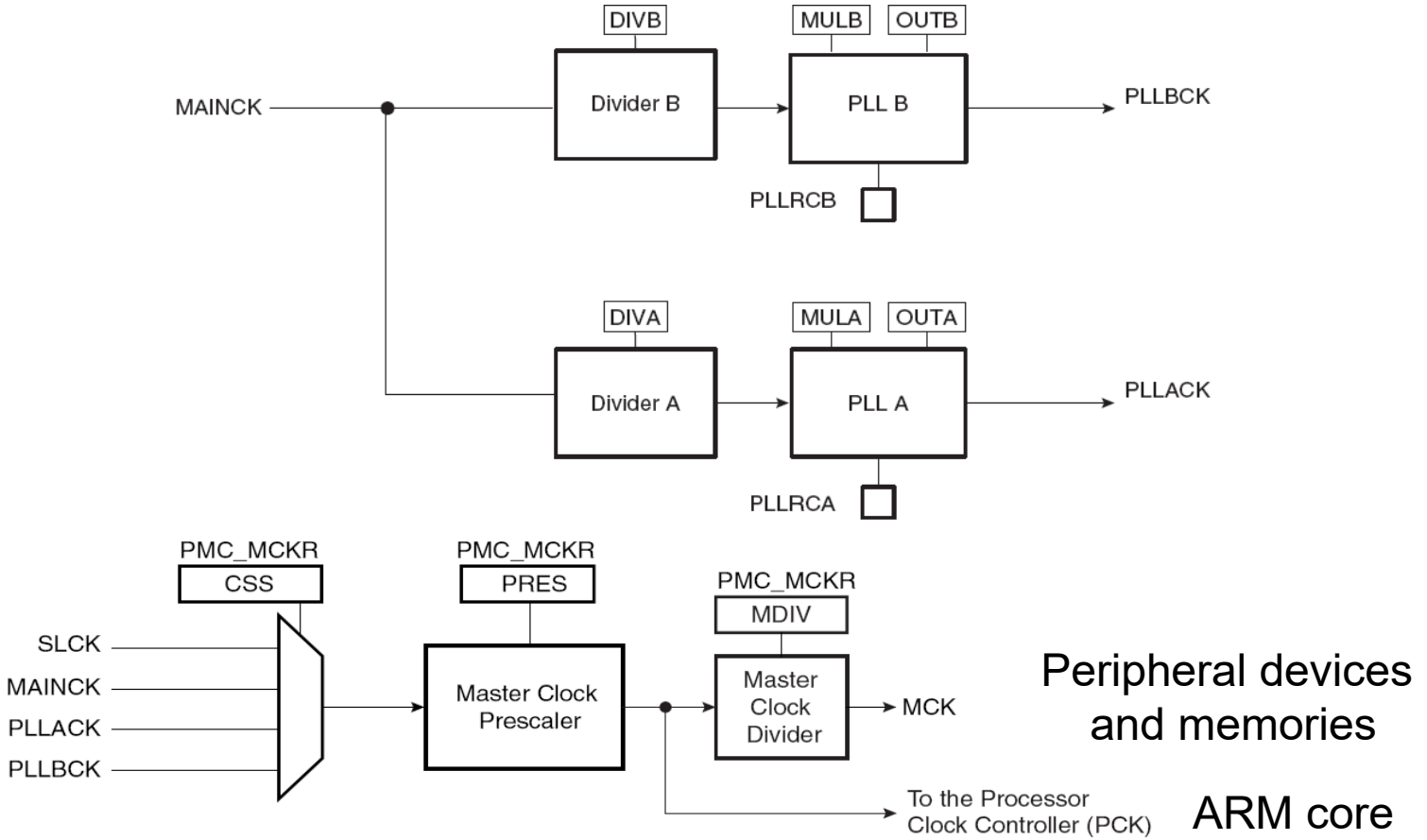



Remapping of FLASH memory is performed after execution of startup code (REMAP register)

# Configuration of Reference Clock – Chapter 28 (1)

After reset processor operates with slow clock with frequency f = 32768 Hz. Slow clock is always active (generated by build-in RC generator).

After reset Phase Locked Loops (PLLs) are disabled.



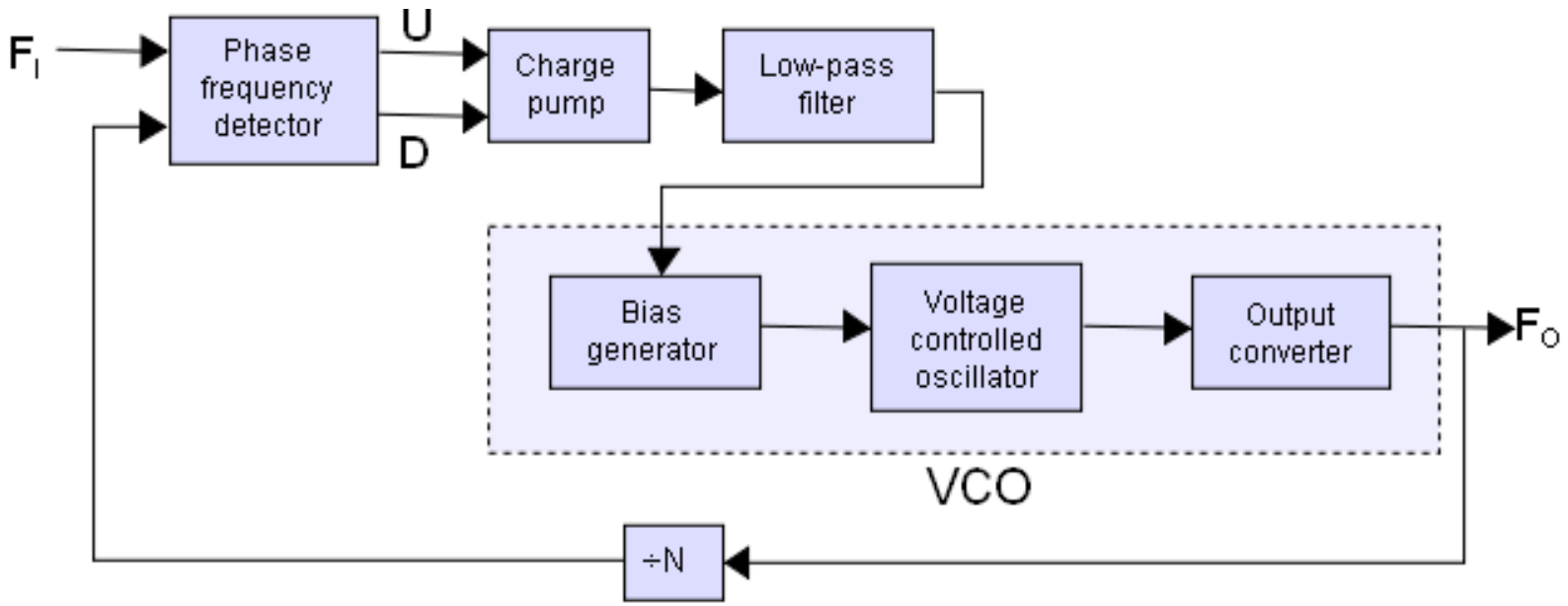
# Generator with PLL (1)

**Phase Locked Loop (PLL)** – electronic circuit based on feedback used for automatic regulation of generator frequency. PLL is used in frequency synthesisers, heterodyne RF receivers, frequency sources and frequency multipliers.

**PLL generator is composed of:**

- Reference generator (quartz generator),
- Phase detector,
- Low pass filter,
- Voltage Controlled Oscillator (VCO),
- Feedback loop with suitable frequency divider.

# Generator with PLL (2)




High frequency signal generated by VCO is connected to output (Fo). The same signal is connected to divider (/N). The output of divider is connected to phase detector (PD) and compared with stable reference signal. Phase difference of reference signal and output signal divided by N is used to control VCO. Feedback loop strive to obtain synchronous signals (phase error equal to 0). Low pass filter is required to make the loop stable.

# Configuration of Reference Clock (2)

**Procedure to turn on PLL generator:**

1. Turn on quartz generator. Wait until frequency will be stable (bit PMC_MOSCS).
2. Configure PLLA, f = 16 367 660*110/9 = ~200 MHz. After turning on PLLA wait until PPLA will be stable (bit PMC_LOCKA) and frequency will be also stable (bit PMC_MCKRDY).
3. Switch processor to use PLLA (in example additional divider by 2), bit AT91C_PMC_CSS_PLLA_CLK. Wait until frequency will be stable.

**Konfiguracja PLLA:**

```
AT91C_BASE_PMC->
                    PMC_PLLAR = AT91C_CKGR_SRCA    |    /* programming PLL */
                    AT91C_CKGR_OUTA_2              |    /* electrical parameters  */
                    (0x3F << 8)                    |    /* counter = 63 */
                    (AT91C_CKGR_MULA & (0x6D << 16)) |  /* multiplier 109 */
                    (AT91C_CKGR_DIVA & 9);              /* divider 9 */
```

f ref = 16 367 660 Hz

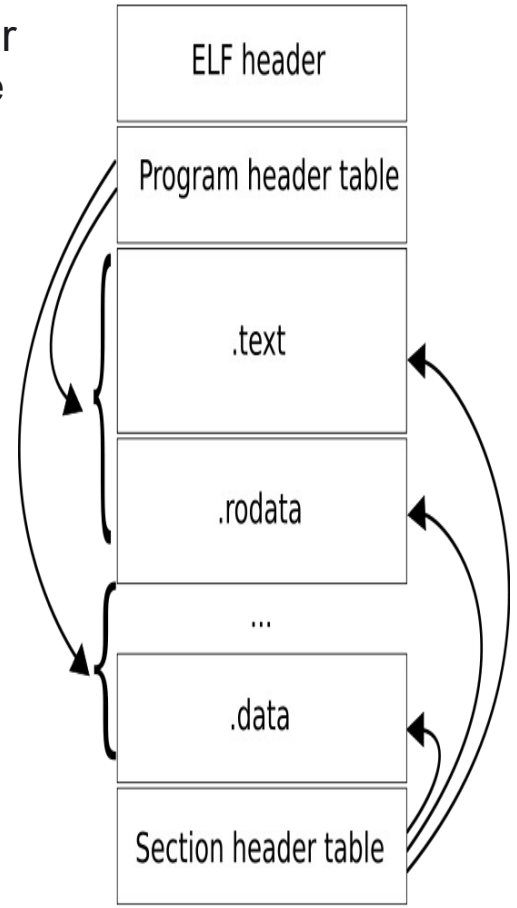
fout = fref * (MULA+1) / DIVA = 16 MHz * 110 / 9 => 200 MHz

$f_{MCK}$ = fout / 2 => ~100 MHz

# Analysis of lowlevel.c file

# COFF vs ELF

- **COFF (Common Object File Format) –** specification of format for executable, object code, and shared library computer files used on Unix systems. COFF was introduced to substitute old **a.out format**. COFF is used also on other platforms, e.g. Windows. Currently, ELF standard is promoted instead of COFF.
- **ELF (Executable and Linkable Format) –** common standard file format for executables, object code, shared libraries, and core dumps used on different architectures, e.g.: x86 family, PowerPC, OpenVMS, BeOS, PlayStation Portable, PlayStation 2, PlayStation 3, Wii, Nintendo DS, GP2X, AmigaOS 4 and Symbian OS v9.
- Usufull tools:
  - readelf
  - elfdump
  - objdump




Źródłó: wikipedia

# Linker Script

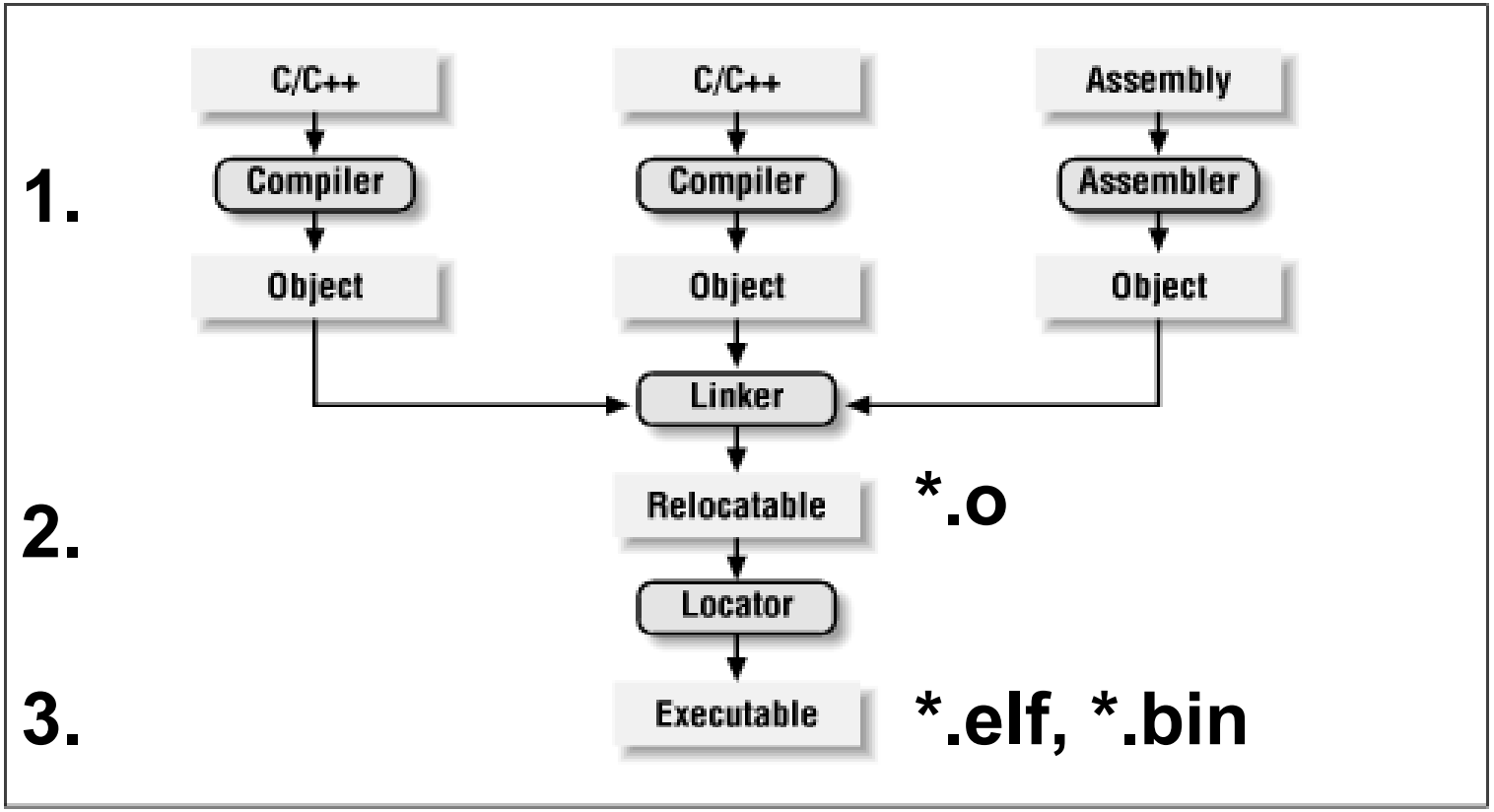
```
/* elf32-littlearm.lds for ARM At91SAM9263 */
OUTPUT_FORMAT ("elf32-littlearm", "elf32-littlearm", "elf32-littlearm")
OUTPUT_ARCH (arm)
ENTRY (reset_handler)
/*#include "project.h"*/
SECTIONS
{
  . = 0x1.0000;                 /* base address */
        .text : { (.text) }     /* code section */
  . = 0x800.0000;               /* base address */
        .data : { (.data) }     /* initialized data */
        .bss : { *(.bss) }      /* uninitialized data */
}

LED_test: $(OBJS)
$(LD) $(LDFLAGS) -Ttext 0x20000000 -Tdata 0x300000 -n -o $(OUTFILE_SDRAM).elf $
(OBJS)
```

# Linker script for ARM AT91SAM9263 processor

```
SECTIONS {
.text : {
  _stext = .;
        *(.text)        /* program code */
        *(.rodata)    /* read-only data (constants) */
        *(.rodata*)
        . = ALIGN(4);
  _etext = . ; }
/* all initialized .data  that go into FLASH  */
.data : AT ( ADDR (.text) + SIZEOF (.text) ) {
  _sdata = .;
        *(.vectors)  /* vectors table */
         (.data)      /* initialized data */
  _edata = .; }
/* all uninitialized .bss that go into FLASH  */
.bss (NOLOAD) : {
        . = ALIGN(4);
  _sbss = .;
        *(.bss)                    /* uninitialized data */
  _ebss = .; } }
end = .;
```

```
CROSS_COMPILE=arm-elf-
LD=$(CROSS_COMPILE)gcc
LDFLAGS+=-nostartfiles -Wl,--cref
LDFLAGS+=-lc -lgcc
LDFLAGS+=-T elf32-littlearm.lds
OBJS  = cstartup.o
OBJS+= lowlevel.o main.o

LED_test: $(OBJS)
$(LD) $(LDFLAGS) -Ttext 0x20000000
                 -Tdata 0x300000 -n -o
      $(OUTFILE_LED_test).elf $(OBJS)
```

# Compilation Process




- 1 phase – compilation of source files → relocable binary files
- 2 phase – linking of relocable files → relocable binary file
- 3 phase – generation of executable file (with assigned addresses)