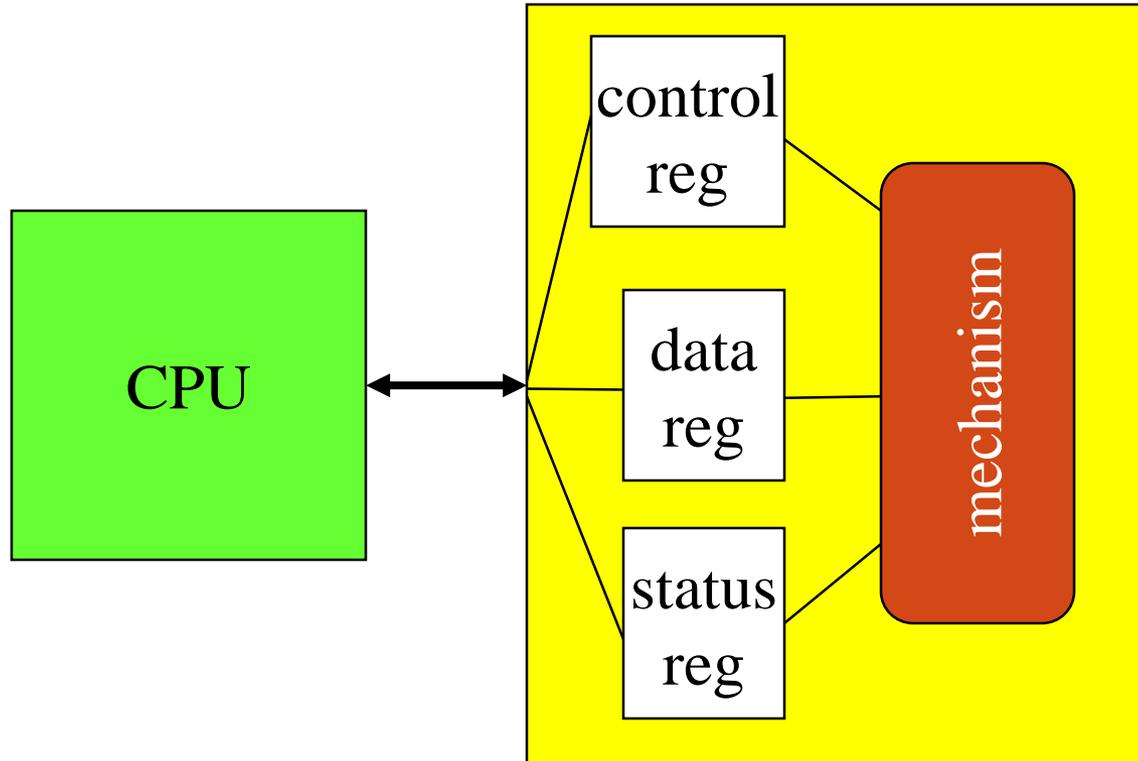


General-Purpose Input/Output

Textbook: Chapter 14 General-Purpose I/O programming

I/O devices

- May include digital and/or non-digital components.
- Typical digital interface to CPU is via addressable registers:



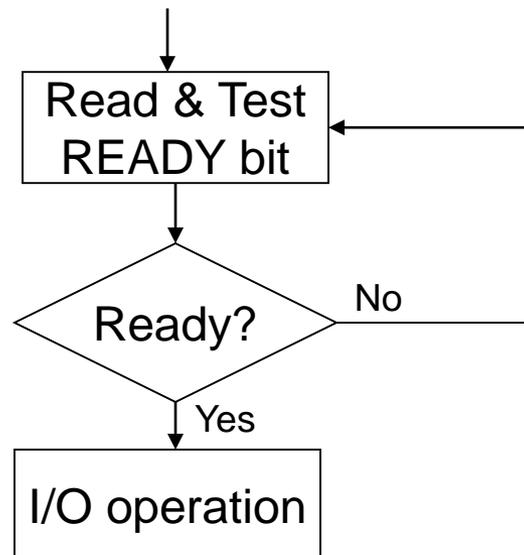
Program-controlled I/O

- Polling

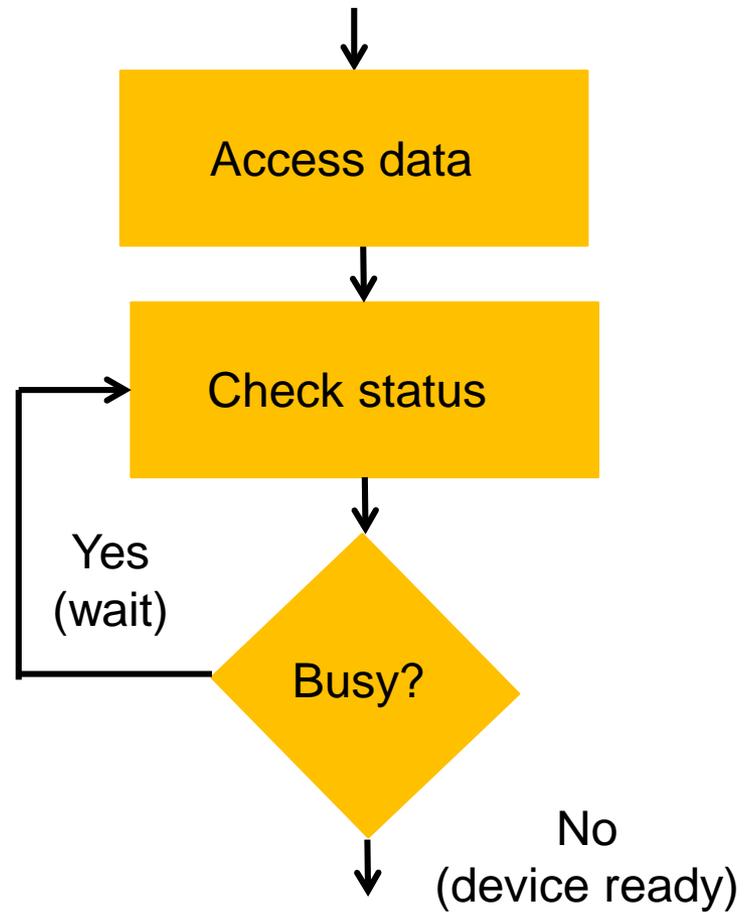
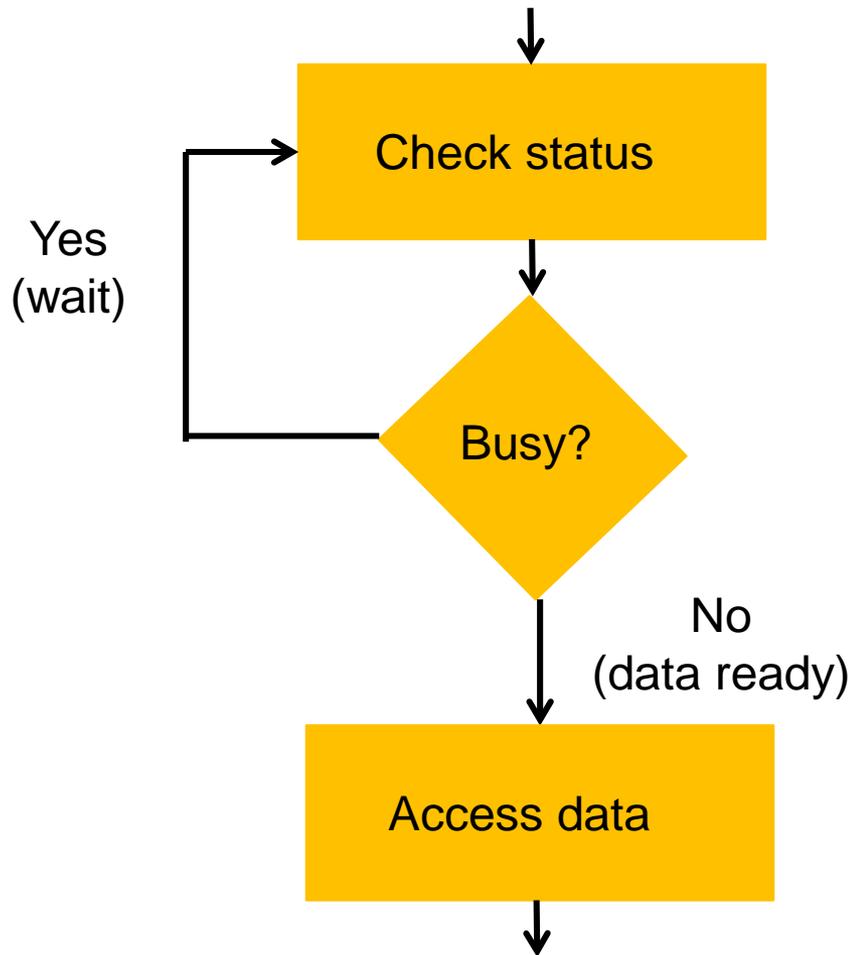
- The CPU checks the status of an I/O device by executing a program (“polling loop”) before initiating I/O operations.

- Example

“READY” signal (active high) of a device is accessible through an “input port”

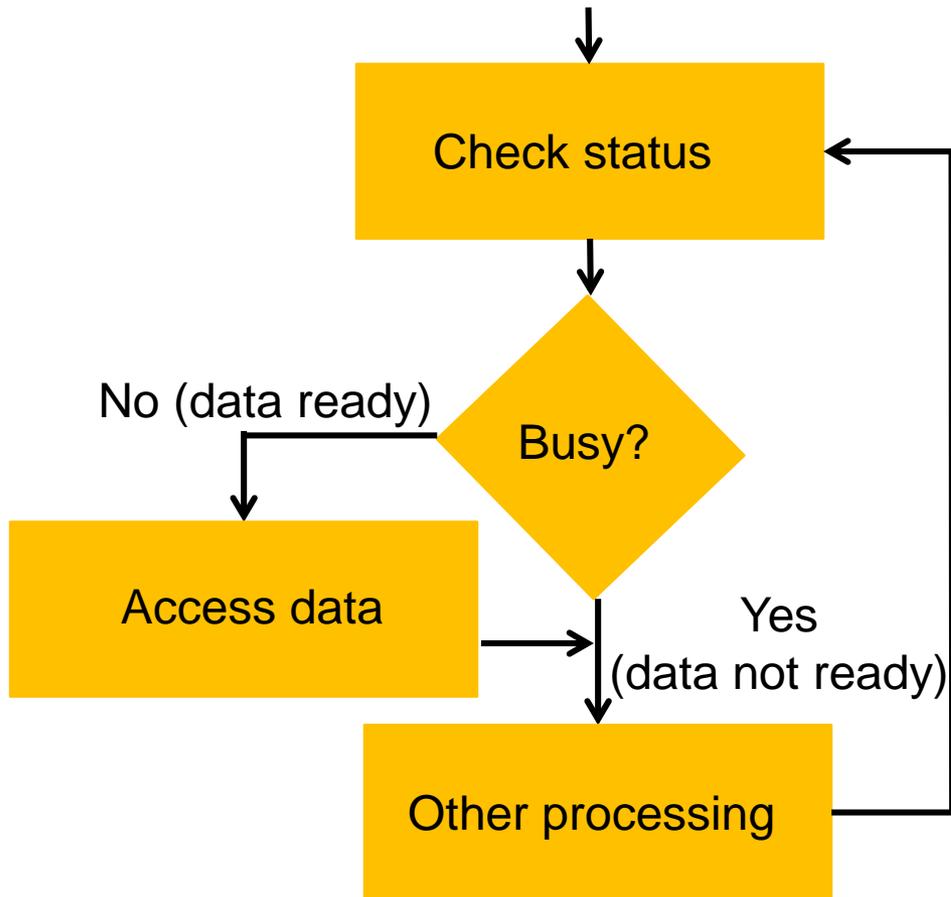


Program-controlled (busy-wait) I/O

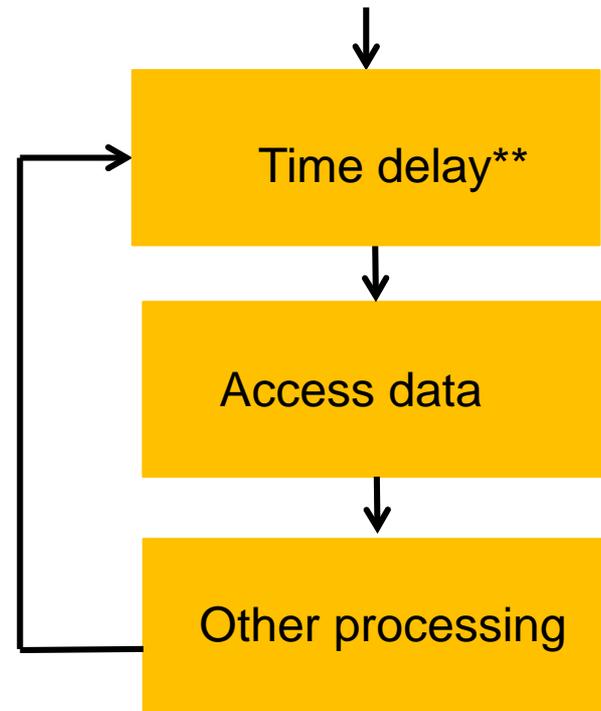


Other I/O options

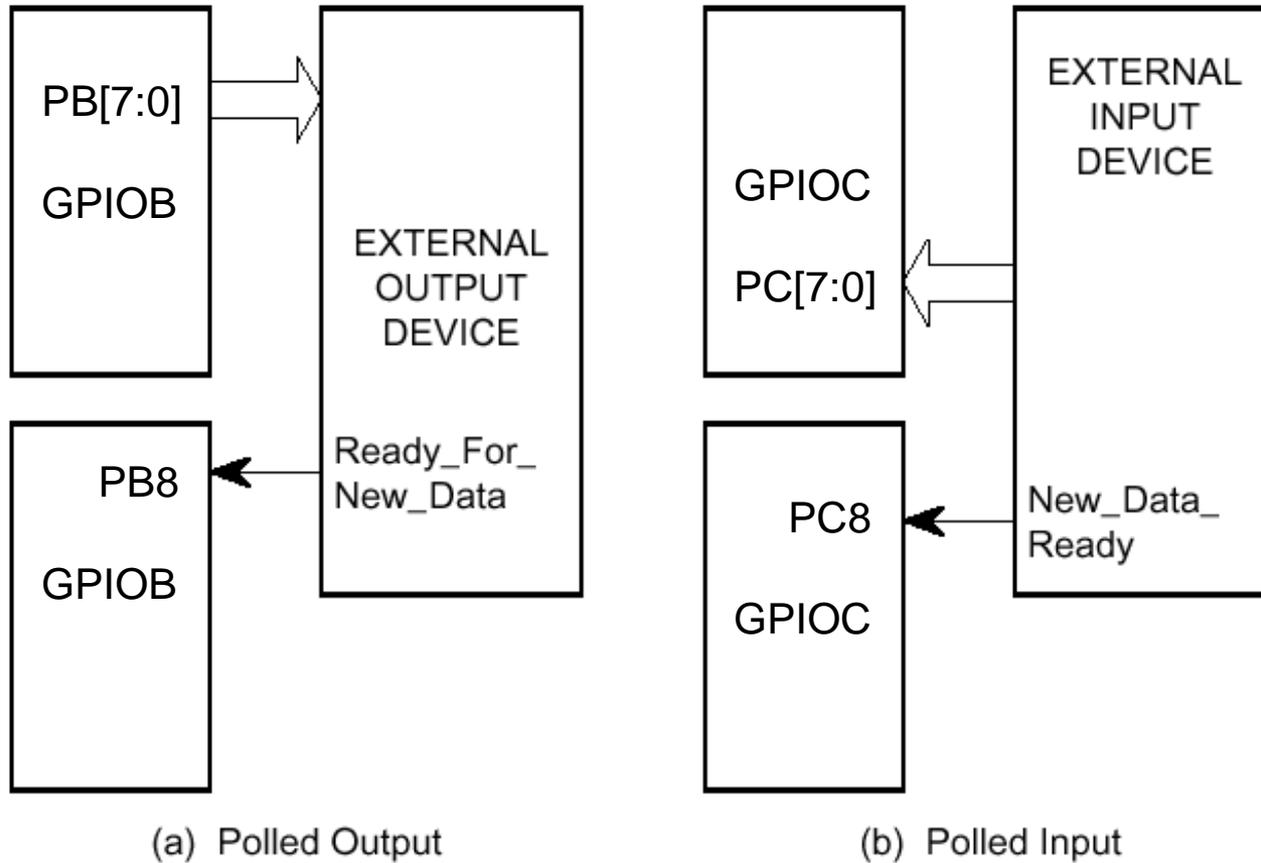
Periodically check device,
without “blocking” the program.



**Wait long enough to
ensure data is ready



Output (a) and input (b) polling.



Busy/wait (polling) output subroutine

```
;Simplest method for synchronizing CPU and device  
;r0 contains a character to output to a device  
#define OUT_STATUS 0x1000 ;8-bit status reg.  
#define OUT_CHAR 0x1004 ;8-bit data reg.
```

OutChar

```
ldr r1,=OUT_STATUS ;point to status reg.  
w ldrb r2,[r1] ;read status register  
tst r2,#0x01 ;check ready bit (bit 0)  
beq w ;repeat until ready=1  
ldr r1,=OUT_CHAR ;point to data reg.  
strb r0,[r1] ;send char to data reg.  
bx lr ;return
```

Busy/wait output C example

```
/* transmit a character string */
/* OUT_CHAR and OUT_STATUS are device addresses

current_char = mystring;           //char string ptr
while (*current_char != '\0') //more to send?
{
    OUT_CHAR = *current_char; //write a character
    while (OUT_STATUS != 0); //wait while busy
    current_char++;           //point to next char
}
```

Busy/wait input subroutine

```
;return character from a device in r0
#define IN_STATUS    0x1000 ;8-bit status reg.
#define IN_CHAR      0x1004 ;8-bit data reg.

CharIn
    ldr    r1,=IN_STATUS    ;point to status reg.
w ldrb r2,[r1]            ;read status register
tst  r2,#0x01           ;check received bit
beq  w                  ;repeat until received=1
    ldr    r1,=in_CHAR     ;point to data reg.
ldrb r0,[r1]           ;read char from data reg.
    bx    lr              ;return
```

Copy data from input device to output device

```
while (TRUE) {  
    /* read */  
    while (IN_STATUS == 0); //repeat until ready  
    achar = IN_DATA; //read data  
    /* write */  
    while (OUT_STATUS != 0); //repeat until ready  
    OUT_DATA = achar; //write data  
}
```

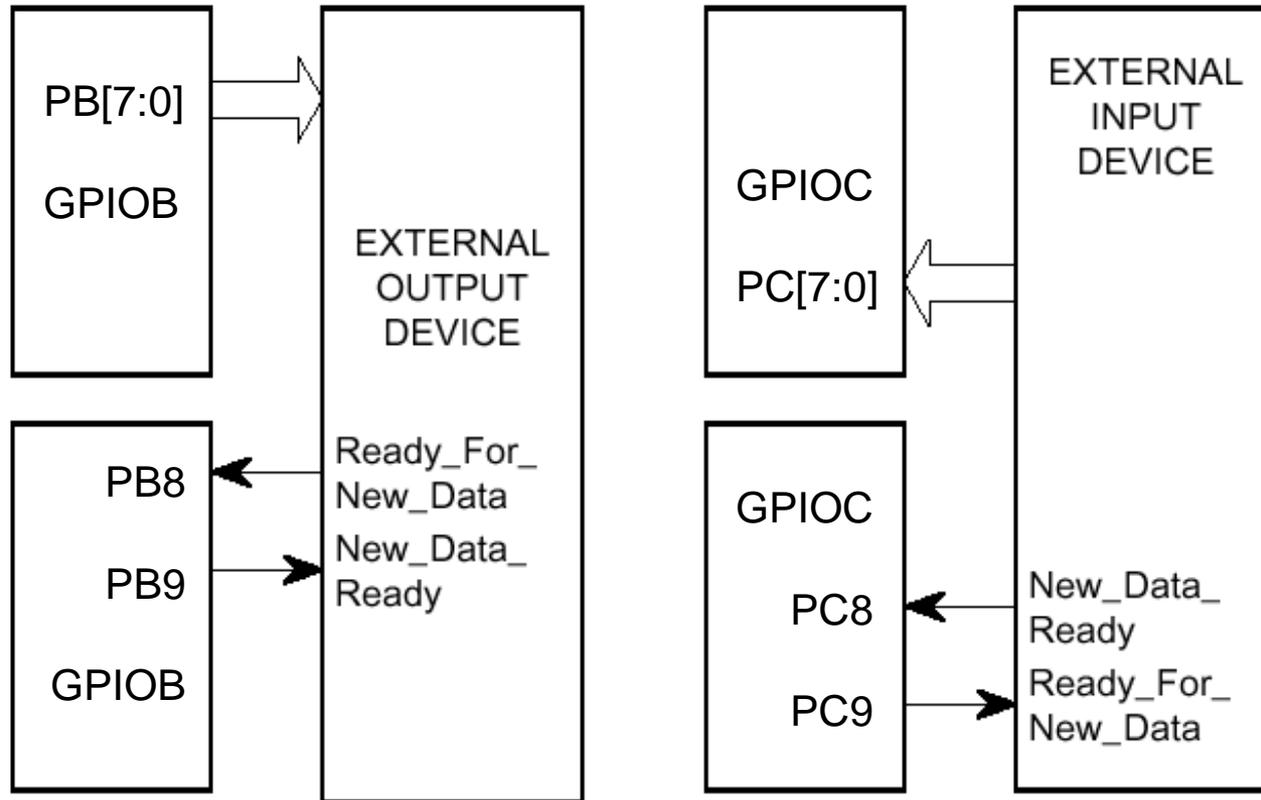
NOTE:

Above assumes **all 8 bits** of IN_STATUS = 0 when ready.

Normally we need to test a **single bit** (IN_STATUS contains multiple bits):

```
while ((IN_STATUS & 0x01) == 0)
```

Hardware handshaking I/O



(a) Handshake Output

(b) Handshake Input

Example: Handshaking I/O Software

; Initialization: write to MODER to configure
; OUTPUT Device: PB7-PB0 outputs, PB8 input, PB9 output
; INPUT Device: PC7-PC0 inputs, PC8 input, PC9 output

; Handshaking output data to PB[7-0] – assume data in r2

; Wait until Ready_For_New_Data is 1

```
ldr    r0,=GPIOB      ;point to register block
SPIN1: ldrh   r1,[r0,#IDR] ;read GPIOB_IDR
      tst    r1,#0x0100 ;test PB8 (Ready)
      beq    SPIN1      ;repeat until ready
```

; Output the data

```
strb   r2,[r0,#ODR]    ;write to PB[7:0]
```

; Strobe handshaking bit New_Data_Ready to signal new data

```
mov    r1,#0x0200      ;select bit PB9
strh   r1,[r0,#BSRRL]  ;New_Data_Ready=1
strh   r1,[r0,#BSRRH]  ;New_Data_Ready=0
```

(Continued next slide)

Example: handshaking I/O software

; Handshaking input data from PC[7-0], return data in r2

; Wait until New_Data_Ready is 1

```
ldr    r0,=GPIOC      ;point to register block
SPIN1: ldrh   r1,[r0,#IDR] ;read GPIOB_IDR
      tst    r1,#0x0100 ;test PC8 (Ready)
      beq    SPIN1      ;repeat until ready
```

; Input data

```
ldrb   r2,[r0,#ODR] ;read PC[7:0]
```

; Strobe handshaking bit New_Data_Ready to signal data received

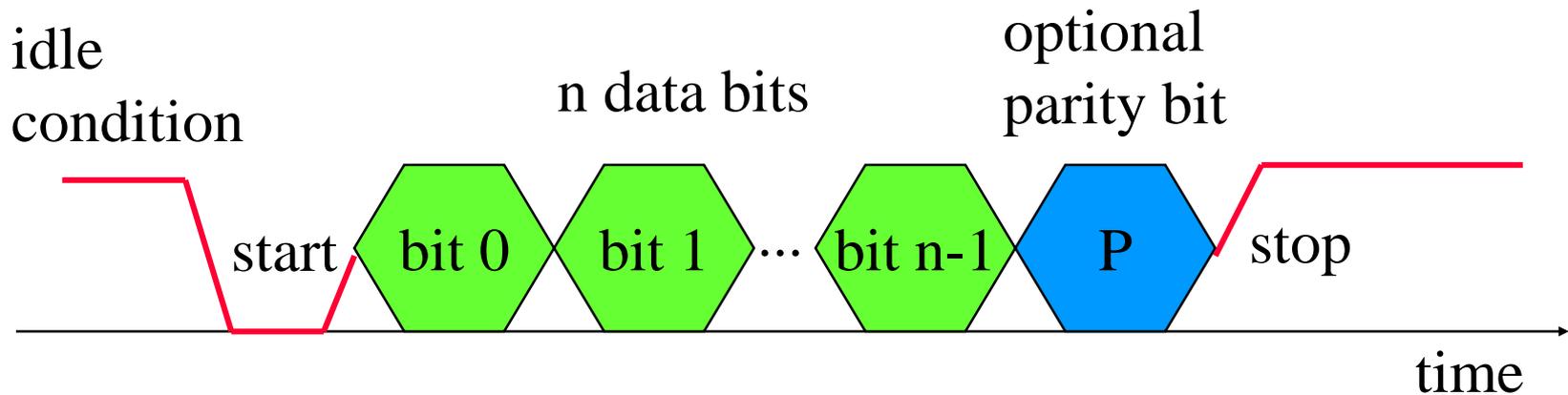
```
mov    r1,#0x0200 ;select bit PC9
strh   r1,[r0,#BSRRL] ;New_Data_Ready=1
strh   r1,[r0,#BSRRH] ;New_Data_Ready=0
```

Example: UART

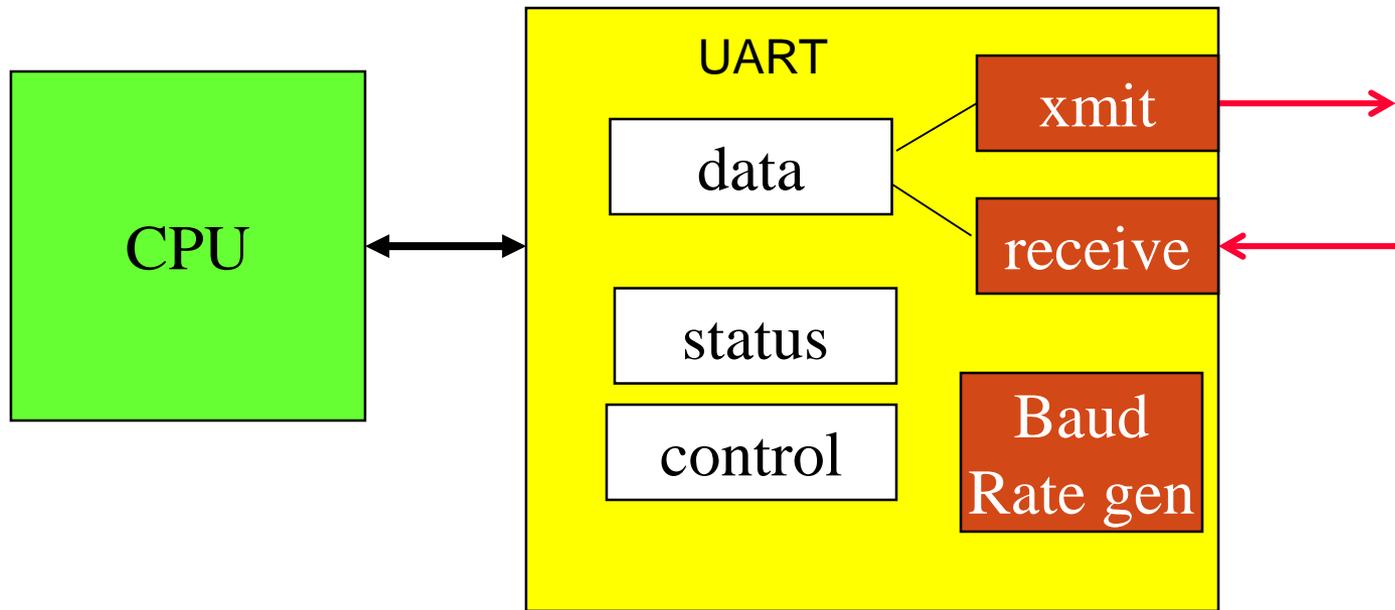
- **Universal asynchronous receiver transmitter (UART)** : provides serial communication.
- Transmit/receive one byte at a time
 - Usually full duplex (transmit/receive concurrently)
- Multiple UARTs are integrated into most microcontrollers
- Allows many communication parameters to be programmed.
- External drivers/receivers often used to provide desired voltage/current levels
 - Example: RS-232
 - Logic 1 voltage [-3v...-12v]
 - Logic 0 voltage [+3v...+12v]
 - Typically for communication up to about 50 feet

Asynchronous serial communication

- Characters are transmitted separately, framed by start and stop bits, with optional parity bit for error detection:



UART CPU interface



Serial communication parameters selected via UART control registers

- Number of bits per character (8 or 9 bits**).
- Parity generation/checking.
 - Enable/disable (** P = 8th/9th “data bit” if enabled)
 - Type of parity: Even or Odd.
- Length of stop bit: 1 or 2 bit periods (1/2 periods also possible)
- Baud rate generator
 - Baud rate = #bits per second received/transmitted.
 - Received data is oversampled at 16x or 8x the baud rate.
- Enable/disable the transmitter and/or receiver.
- Enable/disable interrupts triggered by various conditions.
 - Receiver not empty, transmitter empty, error detected, etc.

Status register indicates various UART conditions/state

- **RXNE**: Receive register not empty
 - Newly-received data available
 - Resets when the data is read from the data register
- **TXE**: Transmitter register empty
 - Ready to accept new data
 - Resets when data is written to the data register
- **TC**: Transmission complete
 - All data has been transmitted
 - Resets if data transmission pending or in progress
- **FE, OE, PE, NE** – errors detected in received data
 - Framing error: incorrect STOP bit (data did not fit within the “frame”)
 - Overrun error: data register overwritten by new data before current data read
 - Parity error: received parity did not match programmed parity
 - Noise error: logic 1 detected in the START bit

UART software

- To transmit data:
 - Verify that the transmitter can accept new data
 - $TXE = 1$ (transmitter buffer empty)
 - Write data to the data register
- To receive data:
 - Detect that a new byte has been received
 - $RXNE = 1$ (receiver not empty)
 - Read data from the data register
- TXE/RXNE detection:
 - Software polls the status bits
 - Interrupt triggered by the status bits
 - Direct Memory Access (DMA) triggered by the status bits, to move data directly between data register and memory