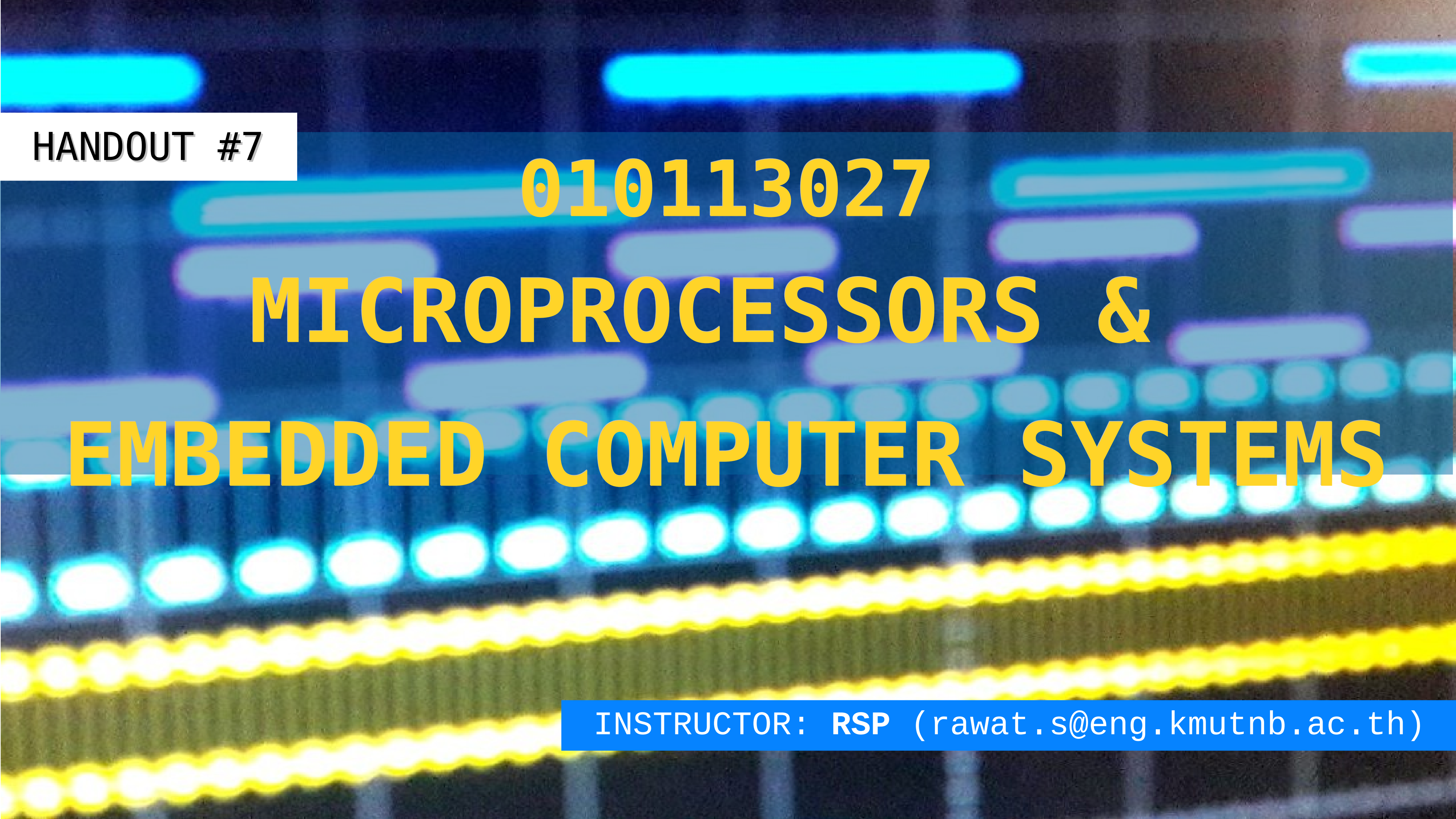


HANDOUT #7

# 010113027 MICROPROCESSORS & EMBEDDED COMPUTER SYSTEMS

INSTRUCTOR: **RSP** (rawat.s@eng.kmutnb.ac.th)

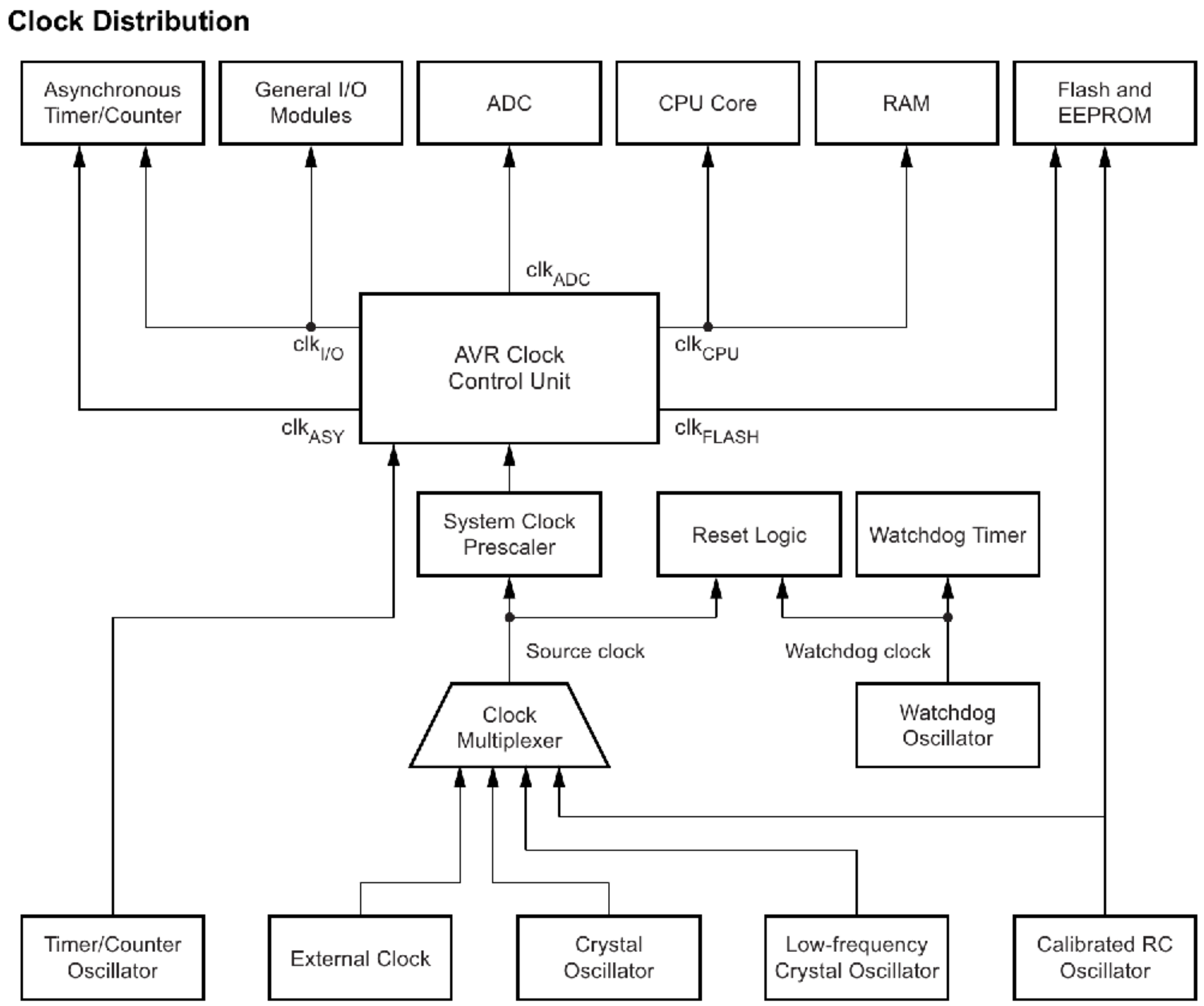
## What We Will Learn...

- AVR Clock Sources
- AVR Low-power / Sleep Modes
- AVR Wakeup Sources
- C Code Demo for Sleep / Wakeup Operations
- USART Peripheral for Asynchronous Serial Data Communication
- USART Configuration
- USART Interrupt Handling
- Using printf() with USART
- C++ Class Implementation for Simple Serial Demo

# AVR Clock Sources

- This block diagram shows **the overall clock system** in the AVR and the **distribution of clock signals**.
- Not all clocks need to be active at the same time.
- To reduce power consumption, **clocks to unused modules** can be halted by selecting appropriate **sleep modes**.

Default clock source: the 8MHz internal RC oscillator with the fuse CKDIV8 programmed, resulting in 1.0MHz system clock.



## AVR Clock Sources

- **CPU Clock:**
  - It drives the AVR core (register file, status register, stack pointer).
  - Halting it stops CPU execution and calculations.
- **I/O Clock:**
  - It is used by peripherals such as Timer/Counters, SPI, and USART.
  - Some interrupts and TWI address detection work asynchronously even when clkI/O is stopped.
- **Flash Clock:**
  - It controls Flash memory access and is ypically active together with the CPU clock.
- **Asynchronous Timer Clock:**
  - It drives Timer/Counter2 from an external clock or 32 kHz crystal.
  - It can be enabled for real-time counting during sleep modes.
- **ADC Clock:**
  - It is a dedicated clock for ADC operation and allows CPU and I/O clocks to be halted to reduce digital noise and improve ADC accuracy.

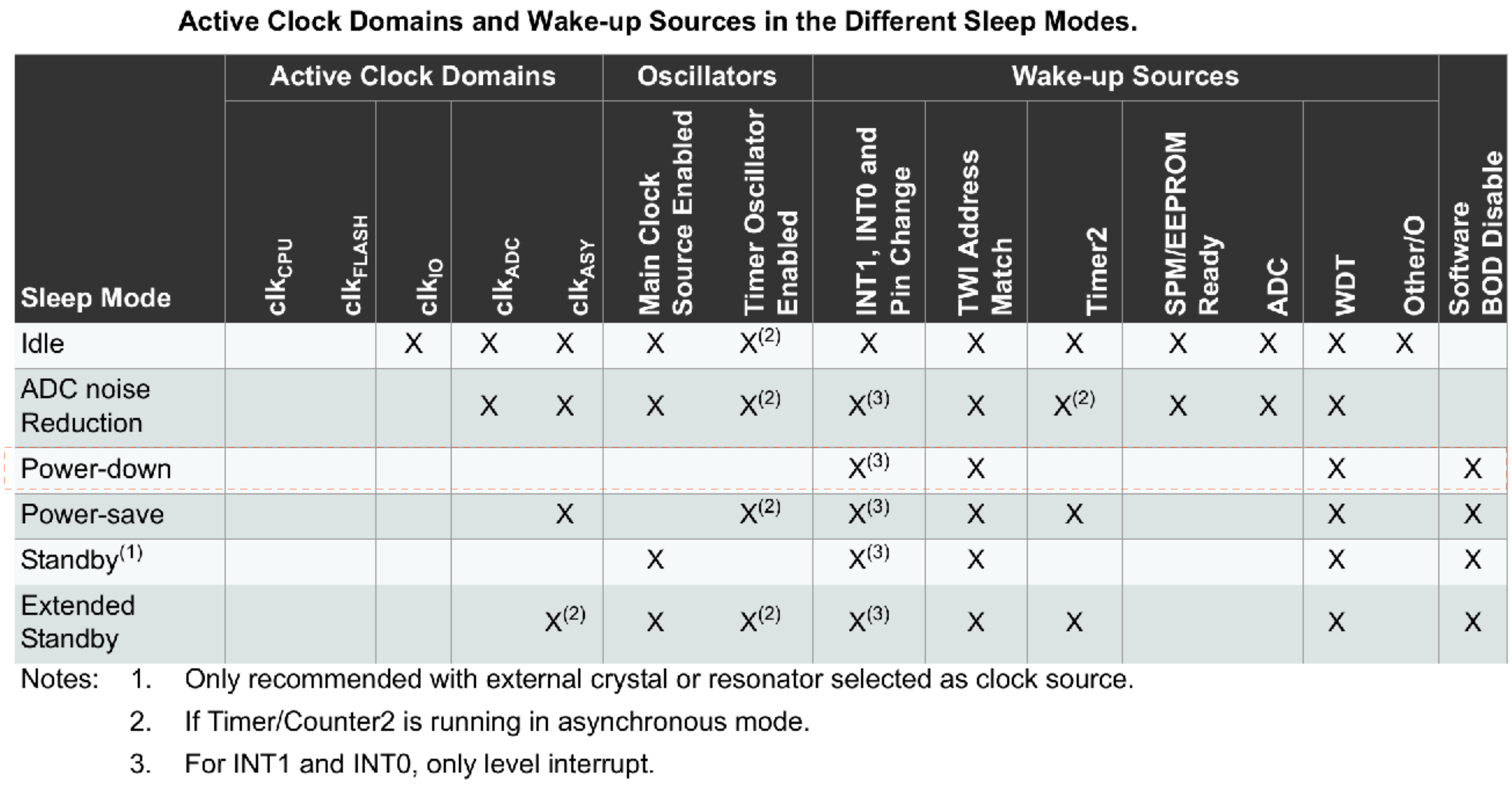
## Low-Power / Sleep Modes in the AVR

- The AVR MCU such as ATmega328P provides **different sleep modes** designed to **reduce power consumption by shutting down parts of the chip** that are not needed while preserving the contents of RAM and registers.
- The MCU can switch from **Active Mode** to a **power-saving sleep mode**.
- There are **six sleep modes**. Each mode provides a different level of power reduction.
  - **Idle**, **ADC Noise Reduction**, **Power-down**, **Power-save**, **Standby**, and **Extended Standby**.
- In all sleep modes, the CPU clock is stopped, preventing instruction execution, while the contents of registers and SRAM are preserved.
- The operation of peripheral modules depends on the selected sleep mode.
- The power-saving level depends on which clock sources and hardware modules are disabled.

## Why Are There Different Sleep Modes?

- Different applications have varying requirements for power saving versus functionality.
- Some situations require retaining peripheral activity (e.g., a timer), while others can shut down almost everything for maximum battery life (for a battery operation).
- The AVR MCU provides multiple modes so designers can balance power consumption and required operation.

# AVR Sleep Modes



**Active Clock Domains and Wake-up Sources in the Different Sleep Modes.**

| Sleep Mode | Active Clock Domains | | | | | Oscillators | | Wake-up Sources | | | | | | | Software BOD Disable |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | $clk_{CPU}$ | $clk_{FLASH}$ | $clk_{IO}$ | $clk_{ADC}$ | $clk_{ASY}$ | Main Clock Source Enabled | Timer Oscillator Enabled | INT1, INT0 and Pin Change | TWI Address Match | Timer2 | SPM/EEPROM Ready | ADC | WDT | Other/O | |
| Idle | | | X | X | X | X | X[2] | X | X | X | X | X | X | X | |
| ADC noise Reduction | | | | X | X | X | X[2] | X[3] | X | X[2] | X | X | X | | |
| Power-down | | | | | | | | X[3] | X | | | | X | | X |
| Power-save | | | | | X | | X[2] | X[3] | X | X | | | X | | X |
| Standby[1] | | | | | | X | | X[3] | X | | | | X | | X |
| Extended Standby | | | | | X[2] | X | X[2] | X[3] | X | X | | | X | | X |

Notes: 1. Only recommended with external crystal or resonator selected as clock source.
2. If Timer/Counter2 is running in asynchronous mode.
3. For INT1 and INT0, only level interrupt.

## AVR Sleep Modes

**Idle Mode:**

- This mode preserves most peripherals and only halts the CPU — useful if still keeping  timers or serial interfaces running.

**Power-down Mode: Current consumption < 1uA**

- It disables almost all clocks and peripherals and is the most power-efficient — but only certain wake-up sources remain enabled.
  - Clock stopped: CPU, Flash, all synchronous peripherals, main oscillator
  - SRAM and register contents preserved
- **Wake-up sources:**
  - External interrupts: INT0 / INT1, PCINT
  - WDT (periodic wakeup)
  - TWI (I2C address match in slave mode)
  - External reset pin

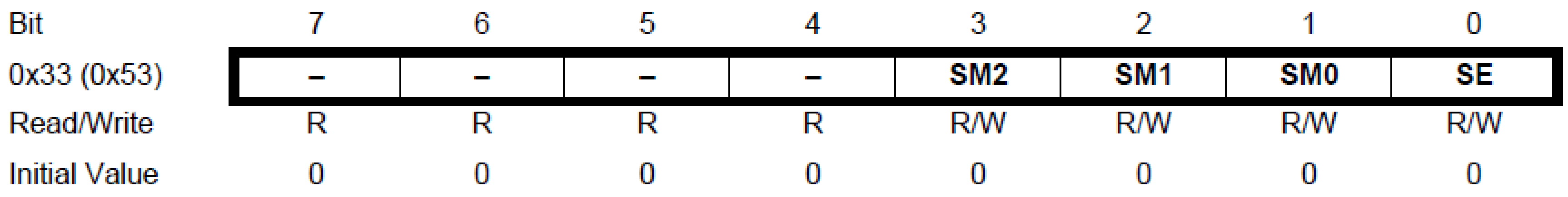
## AVR Sleep Modes

### Power-save Mode

- This mode is identical to Power-down, with one exception: it allows a timer to run asynchronously even while the rest of the system sleeps — useful when a periodic wake-up is needed.
- **Timer/Counter2** is still running.
  - The device can wake up from Timer2 interrupt.
  - Useful for periodic wakeup with Timer2.
  - But only in **asynchronous mode** with external 32.768 kHz crystal on TOSC1/TOSC2 pins.
  - If **Timer/Counter2** is not running, power-down mode is recommended instead of power-save mode.

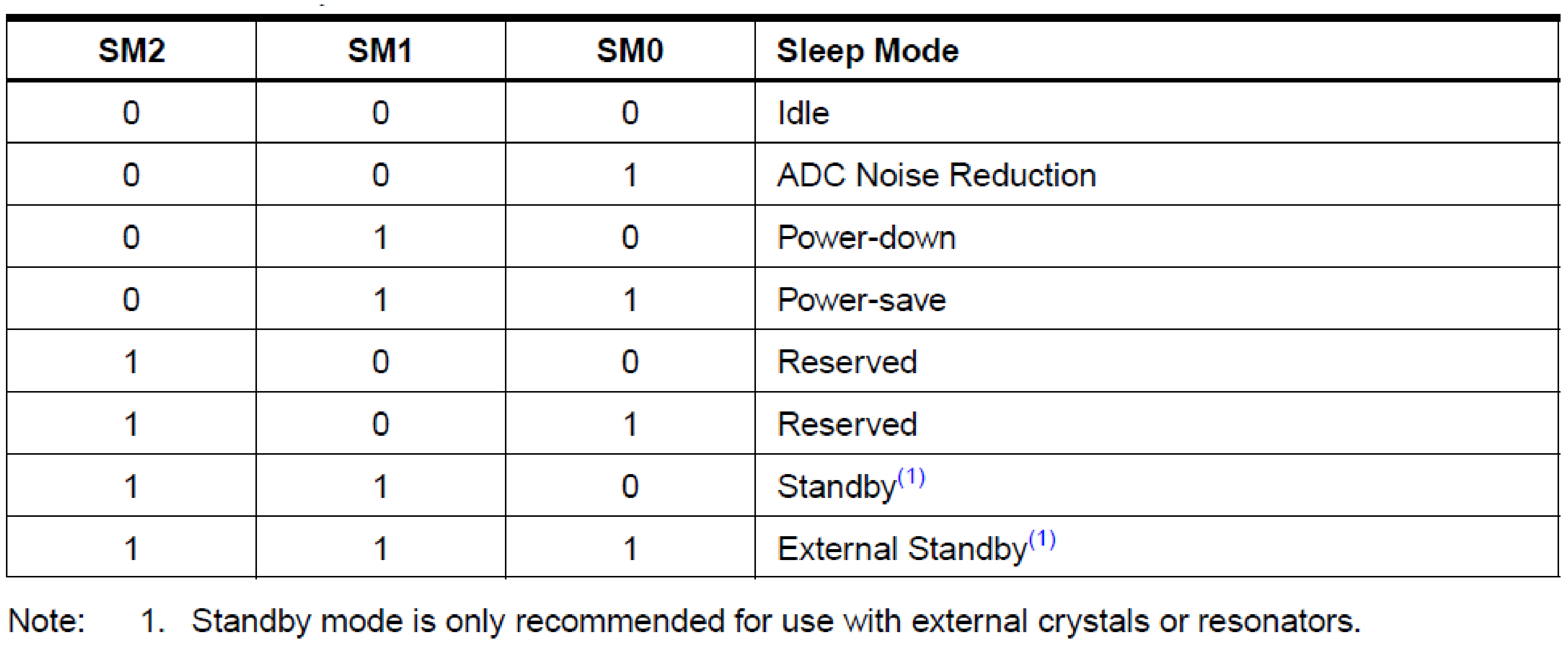
## Registers Related To Sleep Modes

### Sleep Mode Control Register (SMCR)



| Bit | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|---|
| 0x33 (0x53) | – | – | – | – | SM2 | SM1 | SM0 | SE |
| Read/Write | R | R | R | R | R/W | R/W | R/W | R/W |
| Initial Value | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |

The sleep mode control register contains control bits for power management.

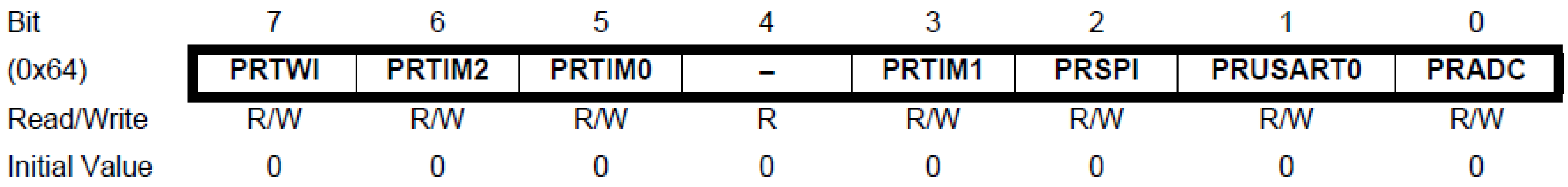


| SM2 | SM1 | SM0 | Sleep Mode |
|---|---|---|---|
| 0 | 0 | 0 | Idle |
| 0 | 0 | 1 | ADC Noise Reduction |
| 0 | 1 | 0 | Power-down |
| 0 | 1 | 1 | Power-save |
| 1 | 0 | 0 | Reserved |
| 1 | 0 | 1 | Reserved |
| 1 | 1 | 0 | Standby(1) |
| 1 | 1 | 1 | External Standby(1) |

Note: 1. Standby mode is only recommended for use with external crystals or resonators.

## Registers Related To Sleep Modes

### PRR – Power Reduction Register



| Bit | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|---|
| (0x64) | PRTWI | PRTIM2 | PRTIM0 | – | PRTIM1 | PRSPI | PRUSART0 | PRADC |
| Read/Write | R/W | R/W | R/W | R | R/W | R/W | R/W | R/W |
| Initial Value | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |

This register can be programmed to disable internal peripheral modules (using clock gating / shutdown) :

- **PRTWI** bit: Power Reduction **TWI (I2C) module**
- **PRTIM2** bit: Power Reduction **Timer/Counter2 module**
- **PRTIM0** bit: Power Reduction **Timer/Counter0 module**
- **PRTIM1** bit: Power Reduction **Timer/Counter1 module**
- **PRSPI** bit: Power Reduction **Serial Peripheral Interface (SPI) module**
- **PRUSART0** bit: Power Reduction **USART0 module**
- **PRADC** bit: Power Reduction **ADC module**

For example, if the **PRADC** bit is set to 1, the ADC is disabled, and if the **PRUSART0** bit is set to 1, the **USART0** module is disabled.

```
PRR |= (1<<PRADC);    // disable ADC
PRR |= (1<<PRUSART0); // disable USART0
PRR |= (1<<PRTIM1);   // disable Timer1
```

## Port Pins in Sleep Modes

- Before entering a sleep mode, all I/O port pins should be configured to minimize power consumption. Ensure that no pins source or sink current by driving resistive loads.
- For pins used as analog inputs, the **digital input buffer should be disabled** to reduce power consumption and minimize noise during analog operation.
- **Digital input buffers** on unused or analog pins can be disabled by setting the corresponding bits in the **Digital Input Disable Registers** (**DIDR0** and **DIDR1**).

  On AVR, the **DIDR registers** control the digital input buffer on certain pins.

  - **DIDR0**: ADC pins (ADC0–ADC5)
  - **DIDR1**: Analog Comparator pins (AIN0, AIN1)

## Steps to Enter a Sleep Mode

- Disable unused peripheral modules. Peripheral shutdown is controlled by the **Power Reduction Register (PRR)**.
  - In power-down mode, almost everything is automatically stopped by hardware.
- Select the desired sleep mode by setting the **SM2:0 bits** in **SMCR** to one of the six mode codes (Idle, ADC Noise Reduction, Power-down, etc.).
- Set the **Sleep Enable (SE) bit** in **SMCR** to 1 to enable the execution of the SLEEP instruction.
- Execute the **SLEEP** instruction. This causes the MCU to enter the selected sleep mode.

## How To Wake Up MCU from Sleep

- When the ATmega328P is in a sleep mode, enabled interrupt sources or reset events will wake the MCU.
- Possible wake-up sources which depend on which sleep mode is used:
  - External interrupts (e.g., INT0/INT1, pin change interrupts)
  - Watchdog Timer (WDT) interrupt or reset
  - Timer overflow or compare match (depending on mode)
  - Two-Wire Serial Interface (TWI or I2C) address match
  - ADC conversion complete (in certain modes)
  - External reset

## How To Wake Up MCU from Sleep

When the ATmega328P is in a sleep mode, enabled interrupt sources or reset events can wake the MCU.

**Wake-up by interrupt**

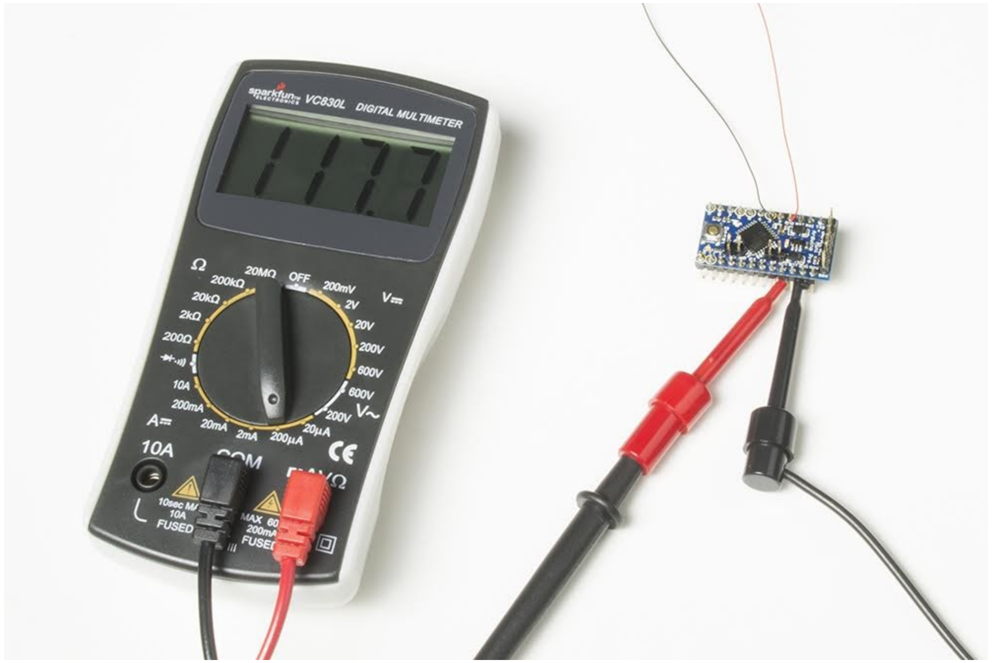
- CPU wakes (SRAM, registers, and I/O registers are retained).
- Interrupt Service Routine (ISR) executes.
- Execution then continues after SLEEP.

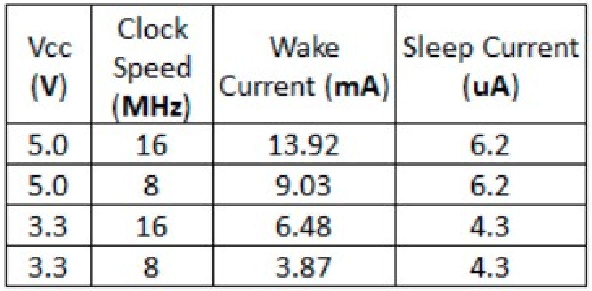
**Wake-up by reset**

- Program restarts from the Reset Vector.
- SRAM and registers are not guaranteed to be preserved.

## Reducing Arduino Power Consumption

Using the Low Power library for Arduno: https://github.com/rocketscream/Low-Power/





| Vcc (V) | Clock Speed (MHz) | Wake Current (mA) | Sleep Current (uA) |
|---|---|---|---|
| 5.0 | 16 | 13.92 | 6.2 |
| 5.0 | 8 | 9.03 | 6.2 |
| 3.3 | 16 | 6.48 | 4.3 |
| 3.3 | 8 | 3.87 | 4.3 |

Source: https://learn.sparkfun.com/tutorials/reducing-arduino-power-consumption/all

## Code Example: Sleep & Wake Up by WDT

The behavior of the example code is as follows (for **ATmega328P**):

- The **WDT** is used and configured to generate an interrupt (without causing a WDT reset) when a **WDT timeout** occurs.
- The **WDT prescaler** is selected to obtain a WDT timeout period of approximately 1 second.
- After the **WDT** is enabled, the **MCU** enters sleep mode by executing the **SLEEP** instruction, and program execution is halted until the MCU is woken up.
- Each time a **WDT** timeout occurs, the **MCU** is woken up from sleep mode and resumes operation. It also turns off the LED.
- The **MCU** also responds to the interrupt by invoking the **ISR for the WDT**, which is programmed to turn on the **LED**.
  - The ISR definition for a **WDT interrupt** using avr-gcc is as follows:

    ```
    ISR(WDT_vect) { … }
    ```

## Code Example: Sleep & Wake Up by WDT

```
#include <avr/io.h>
#include <avr/interrupt.h>
#include <util/delay.h>

#define nop()   __asm__ __volatile__("nop")
#define wdr()   __asm__ __volatile__("wdr")
#define sleep() __asm__ __volatile__("sleep")

#define LED_DDR   DDRB
#define LED_PORT  PORTB
#define LED_BIT   PB5

volatile uint8_t wakeup_count = 0;

void WDT_stop(void) {
    cli();                      // Disable global interrupts
    wdr();                      // Reset WDT
    MCUSR &= ~(1 << WDRF); // Clear WDT reset flag
    // Enable configuration changes
    WDTCSR |= (1 << WDCE) | (1 << WDE);
    WDTCSR = 0x00;              // Disable WDT
    sei();                      // Enable global interrupts
}
// Code continues on the next page...
```

## Code Example: Sleep & Wake Up by WDT

```c
void WDT_init(void) {
    cli();                      // Disable global interrupts
    MCUSR &= ~(1 << WDRF);  // Clear WDT reset flag
    // Enable configuration changes
    WDTCSR |= (1 << WDCE) | (1 << WDE);
    // WDT interrupt mode, ~1 s timeout
    // WDP2 + WDP1 = approx. 1 second
    WDTCSR = (1 << WDIE) | (1 << WDP2) | (1 << WDP1);
    wdr();  // Reset WDT
    sei();  // Enable global interrupts
}

ISR(WDT_vect) {
    wakeup_count++; // Increment wakeup count
    LED_PORT |= (1 << LED_BIT); // Turn on LED
}
```

## Code Example: Sleep & Wake Up by WDT

```
int main(void) {
    // Configure LED pins as outputs
    LED_DDR  |=  (1 << LED_BIT);
    LED_PORT &= ~(1 << LED_BIT);
    _delay_ms(1000);
    WDT_init(); // Initialize Watchdog
    while (1) {
        // Enter power-down sleep mode
        SMCR = (1 << SM1) | (1 << SE);
        sleep(); // Enter sleep mode
        nop();

        _delay_ms(25);
        LED_PORT &= ~(1 << LED_BIT); // Turn off LED
        if (wakeup_count >= 10) {
            LED_PORT |= (1 << LED_BIT); // Turn on LED
            WDT_stop(); // Stop WDT
            while(1);
        }
    }
}
```

## Code Example: Sleep & Wake Up by INT0

The behavior of the example code is as follows (for ATmega328P)

- **External Interrupt Request 0 (INT0)**, which corresponds to pin **PD2** (**Arduino Uno / Nano D3 pin**), is used and connected to an active-low push-button circuit.
- The interrupt trigger condition for **INT0** is configured as **low-level**, consistent with the active-low button connection.
- The **MCU** enters sleep mode, during which the LED remains off.
- The **MCU** is woken up by the **INT0 interrupt** and continues execution, blinking the **LED** for 3 times, before returning to sleep mode.
- The associated **ISR** clears the **INT0 enable bit** in the **EIMSK** register to prevent the interrupt from triggering again before the **MCU** enters sleep mode the next time.

## Code Example: Sleep & Wake Up by INT0

```
#include <avr/io.h>
#include <avr/interrupt.h>
#include <avr/sleep.h>
#include <util/delay.h>

#define nop()   __asm__ __volatile__("nop")

volatile uint8_t woke_up = 0;

ISR(INT0_vect) { // INT0 ISR
  EIMSK &= ~(1 << INT0);   // Disable INT0 immediately
  woke_up = 1;             // Set flag
}

static void peripherals_disable(void) {
  // Disable ADC
  ADCSRA &= ~(1 << ADEN);
  // Disable Analog Comparator
  ACSR |= (1 << ACD);
  // Disable digital input buffers on ADC pins
  DIDR0 = 0x3F;
}
// Code continues on the next page...
```

## Code Example: Sleep & Wake Up by INT0

```
static void gpio_init(void) {
  DDRB = (1 << DDB5);
  PORTB = 0x00; // Outputs low for all PORTB pins
  // PC0..PC5 unused: outputs low
  DDRC  = 0x3F;
  PORTC = 0x00;
  // PD2/INT0 input with pull-up; output low for other pins
  DDRD  = 0xFF;
  DDRD &= ~(1 << DDD2);
  PORTD =  (1 << PORTD2); // Enable pull-up on INT0
}

static void enable_int0(void) {
  // Low-level trigger on INT0
  EICRA &= ~(1 << ISC00);
  EICRA &= ~(1 << ISC01);
  EIFR  |=  (1 << INTF0); // Clear flag
  EIMSK |=  (1 << INT0);  // Enable INT0
}
// Code continues on the next page...
```

## Code Example: Sleep & Wake Up by INT0

```c
static void toggle_led(uint8_t times) {
  for (uint8_t i = 0; i < times; i++) {
    PINB = (1 << PB5);
    _delay_ms(50);
  }
}

static void enter_sleep(void) {
  set_sleep_mode(SLEEP_MODE_PWR_DOWN);
  sleep_enable();
  sleep_bod_disable(); // Disable BOD
  sei();
  sleep_cpu();
  sleep_disable();
}
```

```c
int main(void) {
  cli();
  gpio_init();
  peripherals_disable();
  enable_int0();
  sei();

  while (1) {
    woke_up = 0;
    enter_sleep();
    while (!(PIND & (1 << PIND2))) {
      _delay_ms(5);
    }
    enable_int0();
    if (woke_up) {
      toggle_led(6);
    }
  }
}
```

## Code Example: Sleep & Wake Up by PCINT2

```
#include <avr/io.h>
#include <avr/interrupt.h>
#include <avr/sleep.h>
#include <util/delay.h>

volatile uint8_t woke_up = 0;

ISR(PCINT2_vect) {
  if (!(PIND & (1 << PD2))) { // Is PD2 low?
    woke_up = 1; // Set the flag
  }
}

static void peripherals_disable(void) {
  ADCSRA &= ~(1 << ADEN); // ADC off
  ACSR   |=  (1 << ACD);  // Analog comparator off
  DIDR0   =  0x3F;        // Disable ADC digital inputs
}

// Code continues on the next page...
```

## Code Example: Sleep & Wake Up by PCINT2

```
static void gpio_init(void) {
  // PB5 (LED) output low
  DDRB  = (1 << DDB5);
  PORTB = 0x00;
  // PC0..PC5 unused: outputs low
  DDRC  = 0x3F;
  PORTC = 0x00;
  // PD2 input with pull-up, others output low
  DDRD  = 0xFB; // 0b1111_1011 (only PD2 is input)
  PORTD = (1 << PORTD2); // pull-up on PD2
}

static void pcint_init(void) {
  PCICR  |=  (1 << PCIE2);   // Enable PCINT[23:16]
  PCMSK2 |=  (1 << PCINT18); // Enable PCINT18 (PD2)
  PCIFR  |=  (1 << PCIF2);   // Clear pending flag
}

// Code continues on the next page...
```

## Code Example: Sleep & Wake Up by PCINT2

```
static void toggle_led(uint8_t times) {
  while (times--) {
    PINB = (1 << PB5);
     _delay_ms(50);
  }
}

static void enter_sleep(void) {
  set_sleep_mode(SLEEP_MODE_PWR_DOWN);
  sleep_enable();
  sleep_bod_disable(); // Disable BOD
  sei();
  sleep_cpu();
  sleep_disable();
}
```

```
int main(void) {
    cli();
    gpio_init();
    peripherals_disable();
    pcint_init();
    sei();

    while (1) {
      woke_up = 0;
      enter_sleep(); // Enter sleep mode
      if (woke_up) {
        toggle_led(6); // Blink the LED
      }
    }
}
```

## USART Peripheral

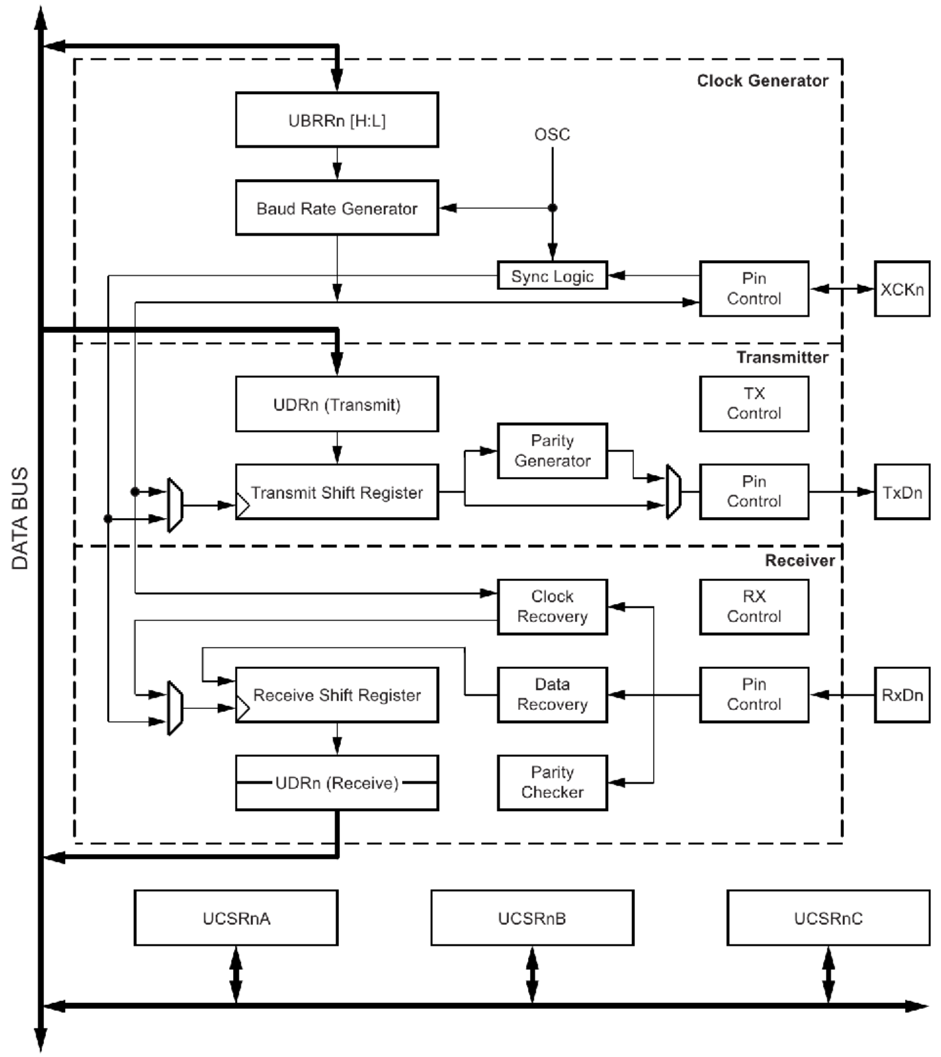
- One common interface is the **USART** (*Universal Synchronous and Asynchronous Receiver and Transmitter*).
- The **ATmega328P** contains **one USART module** (**USART0**).
  - Other AVR devices may provide multiple USARTs (for example, **ATmega2560** has four).
- **USART0** features
  - Performs **bit-serial (bit-by-bit) data communication.**
  - Supports both **asynchronous** and **synchronous** operation.
  - Supports **full-duplex** communication (independent **TX** and **RX** lines).

# USART Block Diagram

In this simplified block diagram, the following hardware components are shown:

- USART-related registers
- USART interface pins (TXD, RXD, XCK)
- Internal data path

USART Block Diagram

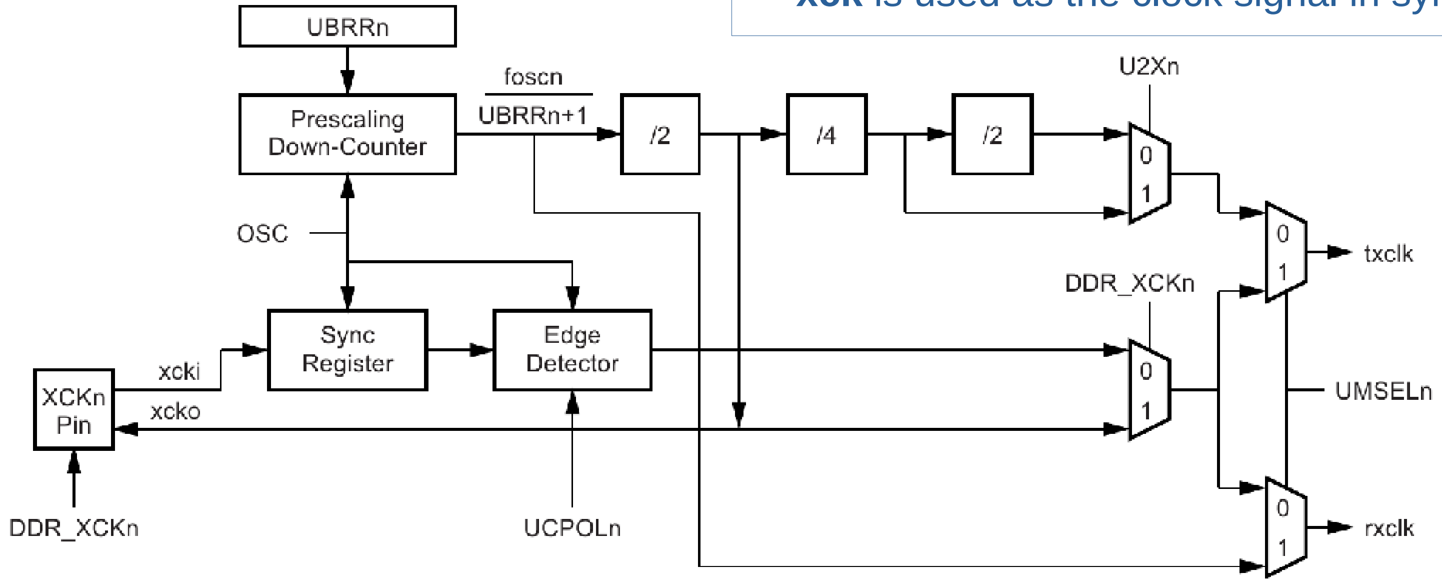


# USART Block Diagram

Note:

- **rxclk** runs at a higher frequency than **txclk**.
- **xck** is used as the clock signal in synchronous mode.

**Clock Generation Logic, Block Diagram**



Signal description:

| | |
|---|---|
| **txclk** | Transmitter clock (Internal Signal). |
| **rxclk** | Receiver base clock (Internal Signal). |
| **xcki** | Input from XCK pin (internal Signal). Used for synchronous slave operation. |
| **xcko** | Clock output to XCK pin (Internal Signal). Used for synchronous master operation. |
| **fosc** | XTAL pin frequency (System Clock). |

## Asynchronous vs. Synchronous Modes

### Asynchronous mode

- No dedicated clock, only **TXD** and **RXD** lines
- Both devices must match the same baud rate and frame format.
- Full-duplex point-to-point communication
- Uses start/stop bits + data bits

### Synchronous mode

- Uses dedicated clock (**XCK**), **TXD** and **RXD** (3-wire)
- Operates in master/slave configuration
  - Master generates the clock.
  - Both devices shift data simultaneously with the clock
  - No start/stop bits

## USART0 Peripheral

**USART0** supports four operating modes:

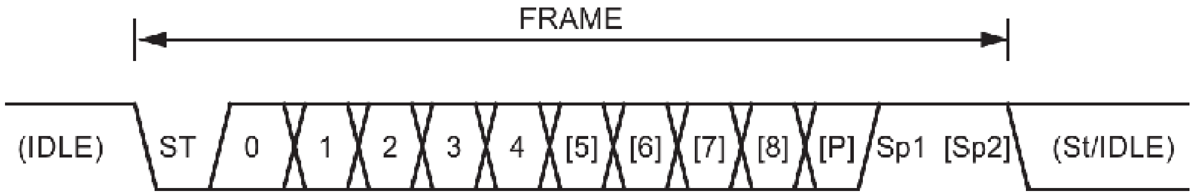
- Asynchronous normal and double (2x) speed
- Synchronous master only: **Master SPI Mode (MSPIM), no Slave-Select (SS) pin**
  - The **ATmega328P** also includes a dedicated **SPI** (*Serial Peripheral Interface*) module.
- In practice, **USART0** is most commonly used for **asynchronous UART-style communication** (e.g., PC serial, USB-to-serial adapters).
- On **Arduino Uno / Nano board**:
  - **Arduino D0 / PD0 pin = RXD** (Receive, digital input)
  - **Arduino D1 / PD1 pin = TXD** (Transmit, digital output)
- These pins are also connected to the on-board **USB-to-Serial bridge chip** for sketch uploading and serial monitoring:
  - **ATmega16U2** on official Uno boards
  - **CH34x or FT232** on many Nano / clones

## USART: Asynchronous Mode

- No clock signal is transmitted.
- Both devices must agree on the **baud rate**.
- **Data frame** format consists of:
  - 1 start bit (always logic **0** or **LOW**)
  - 5–9 data bits
  - optional parity bit (even or odd parity)
  - 1 or 2 stop bits (always logic **1** or **HIGH**)
  - **8N1** = 8-bit data, no parity bit, one stop bit
- In the **Idle state** or when no data is transmitted:
  - **TXD = HIGH (logic 1)**
  - **RXD = HIGH (logic 1**)
  - This is called the **MARK state**.

## UART Data Frame

**Frame Formats**



| | |
|---|---|
| **St** | Start bit, always low. |
| **(n)** | Data bits (0 to 8). |
| **P** | Parity bit. Can be odd or even. |
| **Sp** | Stop bit, always high. |
| **IDLE** | No transfers on the communication line (RxDn or TxDn). An IDLE line must be high. |

## USART: Asynchronous Mode

Steps to use the **USART0** in TX and RX mode.

- Configure **USART baud rate** via the **UBRR0** register (16-bit).
  - **UBRR0** = UART Baud Rate Generator

$$\text{Baudrate (Normal Speed)} = \frac{f_{CPU}}{16 \cdot (UBRR0 + 1)}$$

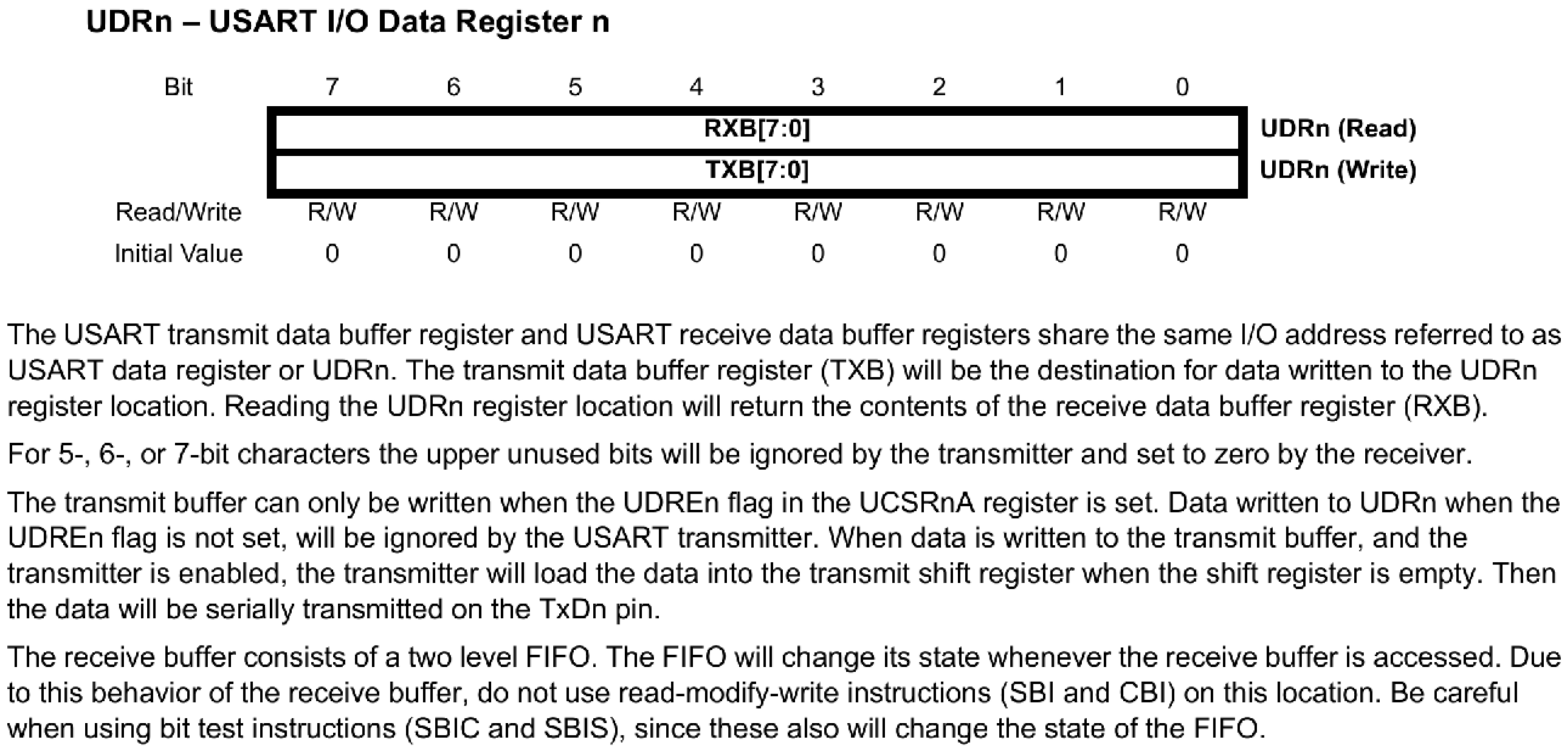
$$\text{Baudrate (Double Speed)} = \frac{f_{CPU}}{8 \cdot (UBRR0 + 1)}$$

- Configure the **UCSR0A** register
  - Normal Speed: **U20X0** = 0
  - Double Speed: **U20X0** = 1
- Configure **UCSR0B** and **UCSR0C** registers
  - Set the frame format
  - Choose frame format (e.g. 8N1: 8 data bits, 1 stop bit)
- Enable transmitter and receiver.
  - Enable Tx: **TXEN0** = 1 in **UCSR0B**
  - Enable Rx: **RXEN0** = 1 in **UCSR0B**
- Read or write the **UDR0** register (TX or RX Data Buffer Register)

## USART: Asynchronous Mode

- **Polling method**
  - **RX operation**:
    - Use the **RXC0 (Receive Complete) bit** in **UCSR0A** to check if a data byte has been received.
    - Read from **UDR0** if **RXC0** is 1.
  - **TX operation**:
    - Wait until transmit register is free, then writes the next data byte.
    - Write to **UDR0** if **UDRE0 (Data Register Empty)** bit in **UCSR0A** is 1.
- **Interrupt-driven method**
  - **RX operation**:
    - Enable RX interrupt and implement the ISR: `USART_RX_vect`
  - **TX operation:**
    - Enable TX interrupt and implement the ISR: `USART_TX_vect`

# USART0 Register: UDR0



**UDRn – USART I/O Data Register n**

| Bit | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 | |
|---|---|---|---|---|---|---|---|---|---|
| | RXB[7:0] | | | | | | | | UDRn (Read) |
| | TXB[7:0] | | | | | | | | UDRn (Write) |
| Read/Write | R/W | R/W | R/W | R/W | R/W | R/W | R/W | R/W | |
| Initial Value | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | |

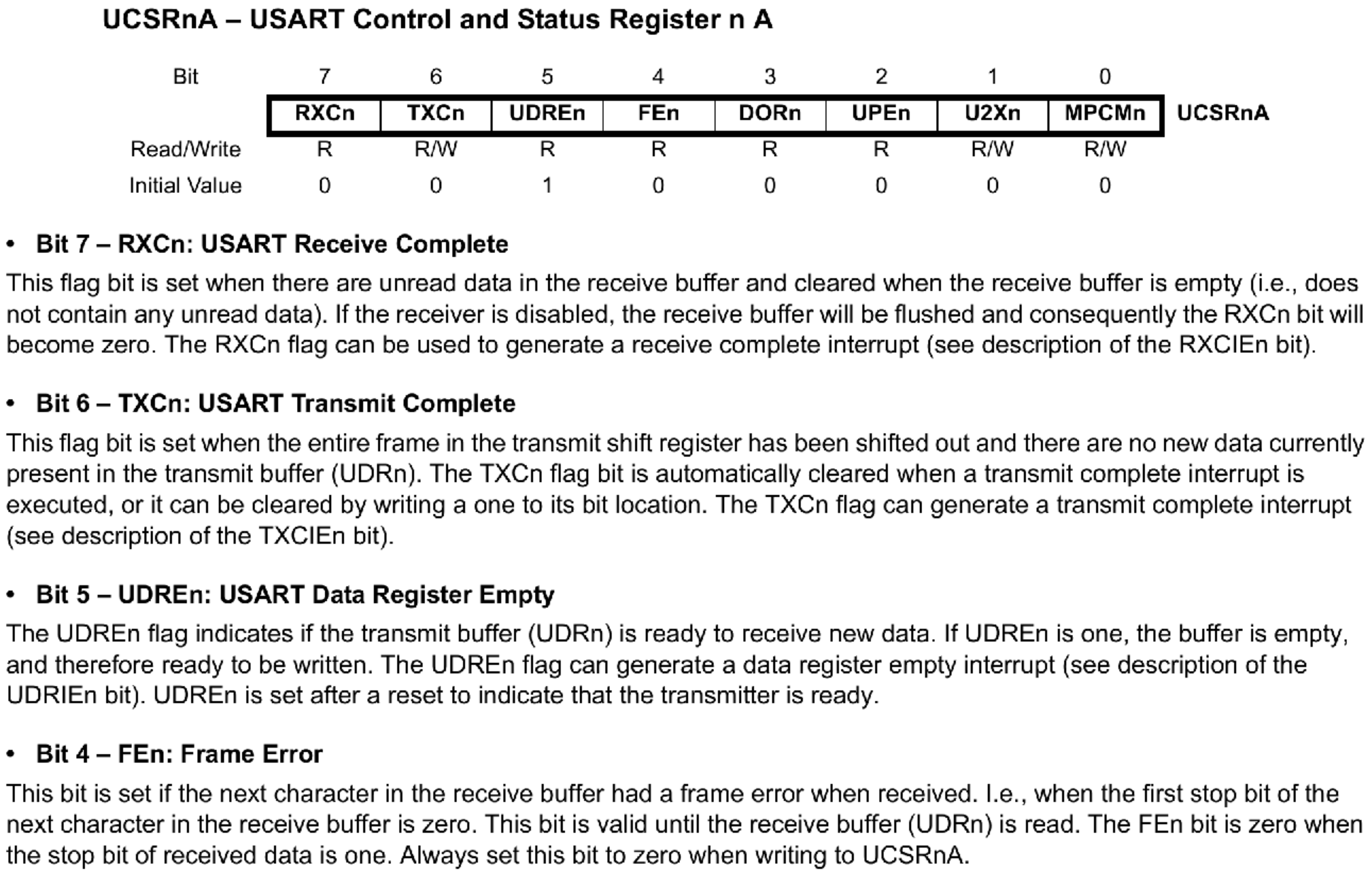
The USART transmit data buffer register and USART receive data buffer registers share the same I/O address referred to as USART data register or UDRn. The transmit data buffer register (TXB) will be the destination for data written to the UDRn register location. Reading the UDRn register location will return the contents of the receive data buffer register (RXB).

For 5-, 6-, or 7-bit characters the upper unused bits will be ignored by the transmitter and set to zero by the receiver.

The transmit buffer can only be written when the UDREn flag in the UCSRnA register is set. Data written to UDRn when the UDREn flag is not set, will be ignored by the USART transmitter. When data is written to the transmit buffer, and the transmitter is enabled, the transmitter will load the data into the transmit shift register when the shift register is empty. Then the data will be serially transmitted on the TxDn pin.

The receive buffer consists of a two level FIFO. The FIFO will change its state whenever the receive buffer is accessed. Due to this behavior of the receive buffer, do not use read-modify-write instructions (SBI and CBI) on this location. Be careful when using bit test instructions (SBIC and SBIS), since these also will change the state of the FIFO.

## USART0 Register: UCSR0A



**UCSRnA – USART Control and Status Register n A**

| Bit | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 | |
|---|---|---|---|---|---|---|---|---|---|
| | RXCn | TXCn | UDREn | FEn | DORn | UPEn | U2Xn | MPCMn | UCSRnA |
| Read/Write | R | R/W | R | R | R | R | R/W | R/W | |
| Initial Value | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | |

- **Bit 7 – RXCn: USART Receive Complete**

This flag bit is set when there are unread data in the receive buffer and cleared when the receive buffer is empty (i.e., does not contain any unread data). If the receiver is disabled, the receive buffer will be flushed and consequently the RXCn bit will become zero. The RXCn flag can be used to generate a receive complete interrupt (see description of the RXCIEn bit).

- **Bit 6 – TXCn: USART Transmit Complete**

This flag bit is set when the entire frame in the transmit shift register has been shifted out and there are no new data currently present in the transmit buffer (UDRn). The TXCn flag bit is automatically cleared when a transmit complete interrupt is executed, or it can be cleared by writing a one to its bit location. The TXCn flag can generate a transmit complete interrupt (see description of the TXCIEn bit).

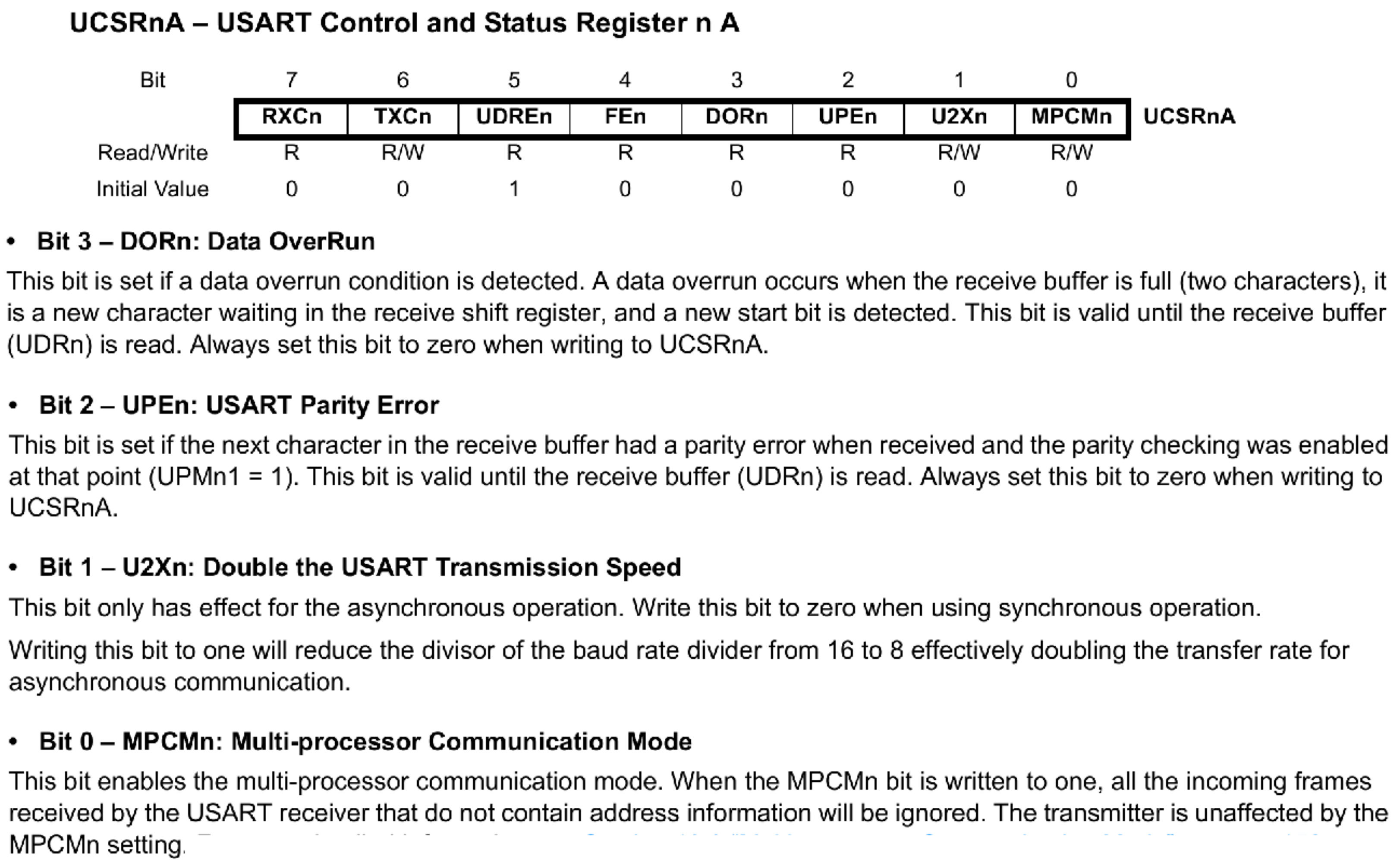
- **Bit 5 – UDREn: USART Data Register Empty**

The UDREn flag indicates if the transmit buffer (UDRn) is ready to receive new data. If UDREn is one, the buffer is empty, and therefore ready to be written. The UDREn flag can generate a data register empty interrupt (see description of the UDRIEn bit). UDREn is set after a reset to indicate that the transmitter is ready.

- **Bit 4 – FEn: Frame Error**

This bit is set if the next character in the receive buffer had a frame error when received. I.e., when the first stop bit of the next character in the receive buffer is zero. This bit is valid until the receive buffer (UDRn) is read. The FEn bit is zero when the stop bit of received data is one. Always set this bit to zero when writing to UCSRnA.

# USART0 Register: UCSR0A



**UCSRnA – USART Control and Status Register n A**

| Bit | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 | |
|---|---|---|---|---|---|---|---|---|---|
| | RXCn | TXCn | UDREn | FEn | DORn | UPEn | U2Xn | MPCMn | UCSRnA |
| Read/Write | R | R/W | R | R | R | R | R/W | R/W | |
| Initial Value | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | |

- **Bit 3 – DORn: Data OverRun**

This bit is set if a data overrun condition is detected. A data overrun occurs when the receive buffer is full (two characters), it is a new character waiting in the receive shift register, and a new start bit is detected. This bit is valid until the receive buffer (UDRn) is read. Always set this bit to zero when writing to UCSRnA.

- **Bit 2 – UPEn: USART Parity Error**

This bit is set if the next character in the receive buffer had a parity error when received and the parity checking was enabled at that point (UPMn1 = 1). This bit is valid until the receive buffer (UDRn) is read. Always set this bit to zero when writing to UCSRnA.

- **Bit 1 – U2Xn: Double the USART Transmission Speed**

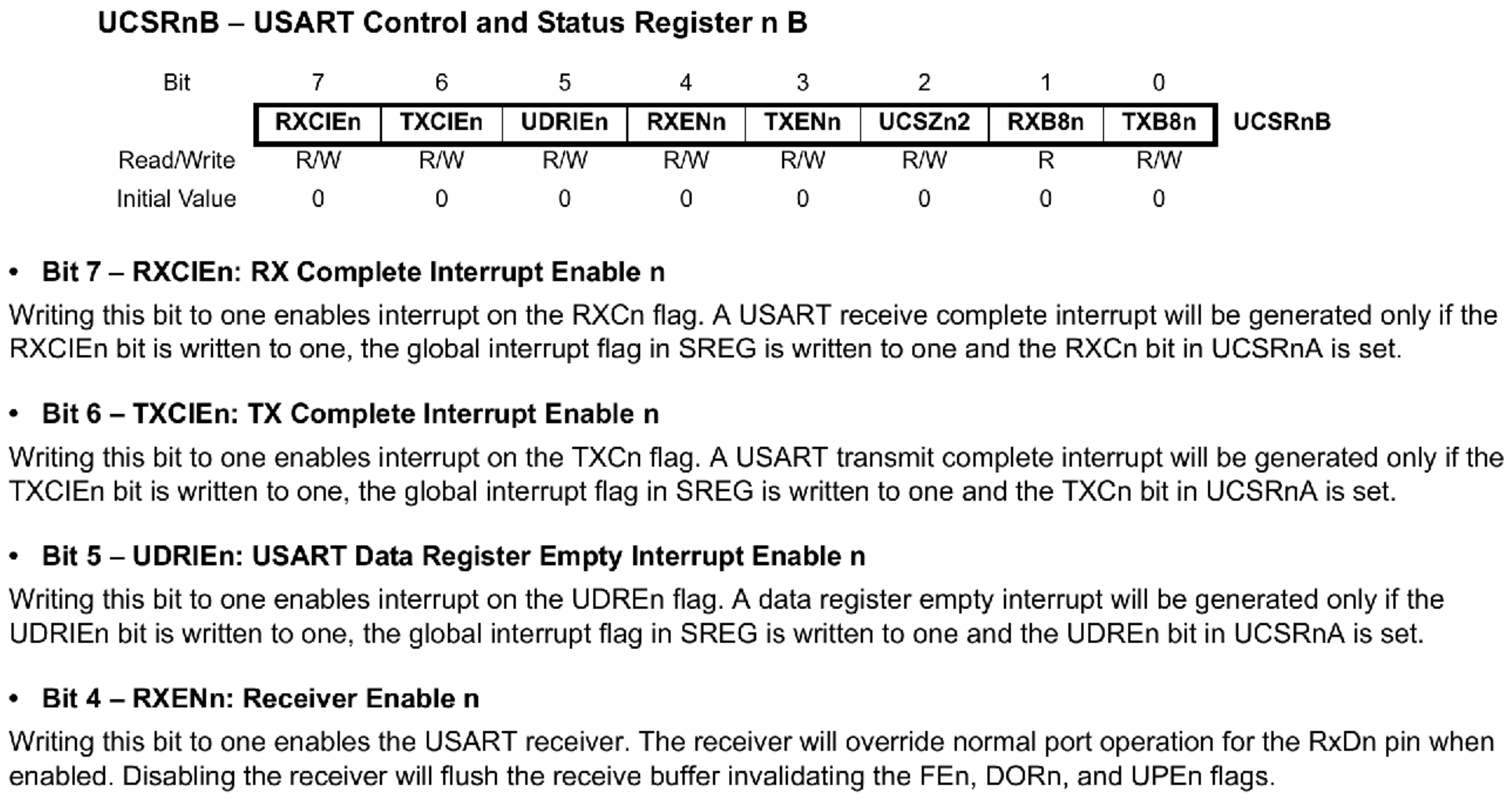
This bit only has effect for the asynchronous operation. Write this bit to zero when using synchronous operation.

Writing this bit to one will reduce the divisor of the baud rate divider from 16 to 8 effectively doubling the transfer rate for asynchronous communication.

- **Bit 0 – MPCMn: Multi-processor Communication Mode**

This bit enables the multi-processor communication mode. When the MPCMn bit is written to one, all the incoming frames received by the USART receiver that do not contain address information will be ignored. The transmitter is unaffected by the MPCMn setting.

## USART0 Register: UCSR0B



**UCSRnB – USART Control and Status Register n B**

| Bit | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 | |
|---|---|---|---|---|---|---|---|---|---|
| | RXCIEn | TXCIEn | UDRIEn | RXENn | TXENn | UCSZn2 | RXB8n | TXB8n | UCSRnB |
| Read/Write | R/W | R/W | R/W | R/W | R/W | R/W | R | R/W | |
| Initial Value | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | |

- **Bit 7 – RXCIEn: RX Complete Interrupt Enable n**

Writing this bit to one enables interrupt on the RXCn flag. A USART receive complete interrupt will be generated only if the RXCIEn bit is written to one, the global interrupt flag in SREG is written to one and the RXCn bit in UCSRnA is set.

- **Bit 6 – TXCIEn: TX Complete Interrupt Enable n**

Writing this bit to one enables interrupt on the TXCn flag. A USART transmit complete interrupt will be generated only if the TXCIEn bit is written to one, the global interrupt flag in SREG is written to one and the TXCn bit in UCSRnA is set.

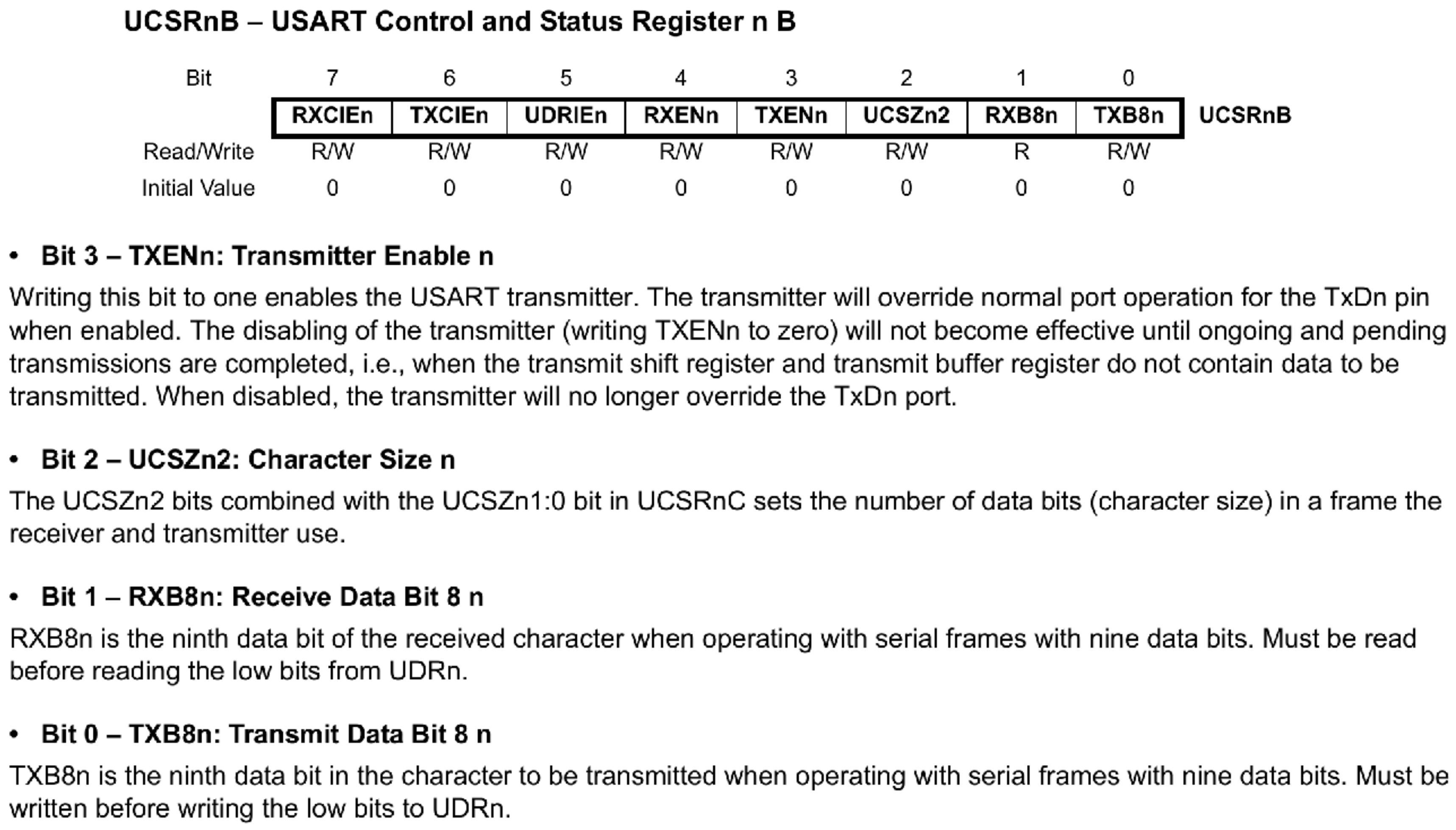
- **Bit 5 – UDRIEn: USART Data Register Empty Interrupt Enable n**

Writing this bit to one enables interrupt on the UDREn flag. A data register empty interrupt will be generated only if the UDRIEn bit is written to one, the global interrupt flag in SREG is written to one and the UDREn bit in UCSRnA is set.

- **Bit 4 – RXENn: Receiver Enable n**

Writing this bit to one enables the USART receiver. The receiver will override normal port operation for the RxDn pin when enabled. Disabling the receiver will flush the receive buffer invalidating the FEn, DORn, and UPEn flags.

## USART0 Register: UCSR0B



**UCSRnB – USART Control and Status Register n B**

| Bit | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 | |
|---|---|---|---|---|---|---|---|---|---|
| | RXCIEn | TXCIEn | UDRIEn | RXENn | TXENn | UCSZn2 | RXB8n | TXB8n | UCSRnB |
| Read/Write | R/W | R/W | R/W | R/W | R/W | R/W | R | R/W | |
| Initial Value | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | |

- **Bit 3 – TXENn: Transmitter Enable n**

Writing this bit to one enables the USART transmitter. The transmitter will override normal port operation for the TxDn pin when enabled. The disabling of the transmitter (writing TXENn to zero) will not become effective until ongoing and pending transmissions are completed, i.e., when the transmit shift register and transmit buffer register do not contain data to be transmitted. When disabled, the transmitter will no longer override the TxDn port.

- **Bit 2 – UCSZn2: Character Size n**

The UCSZn2 bits combined with the UCSZn1:0 bit in UCSRnC sets the number of data bits (character size) in a frame the receiver and transmitter use.

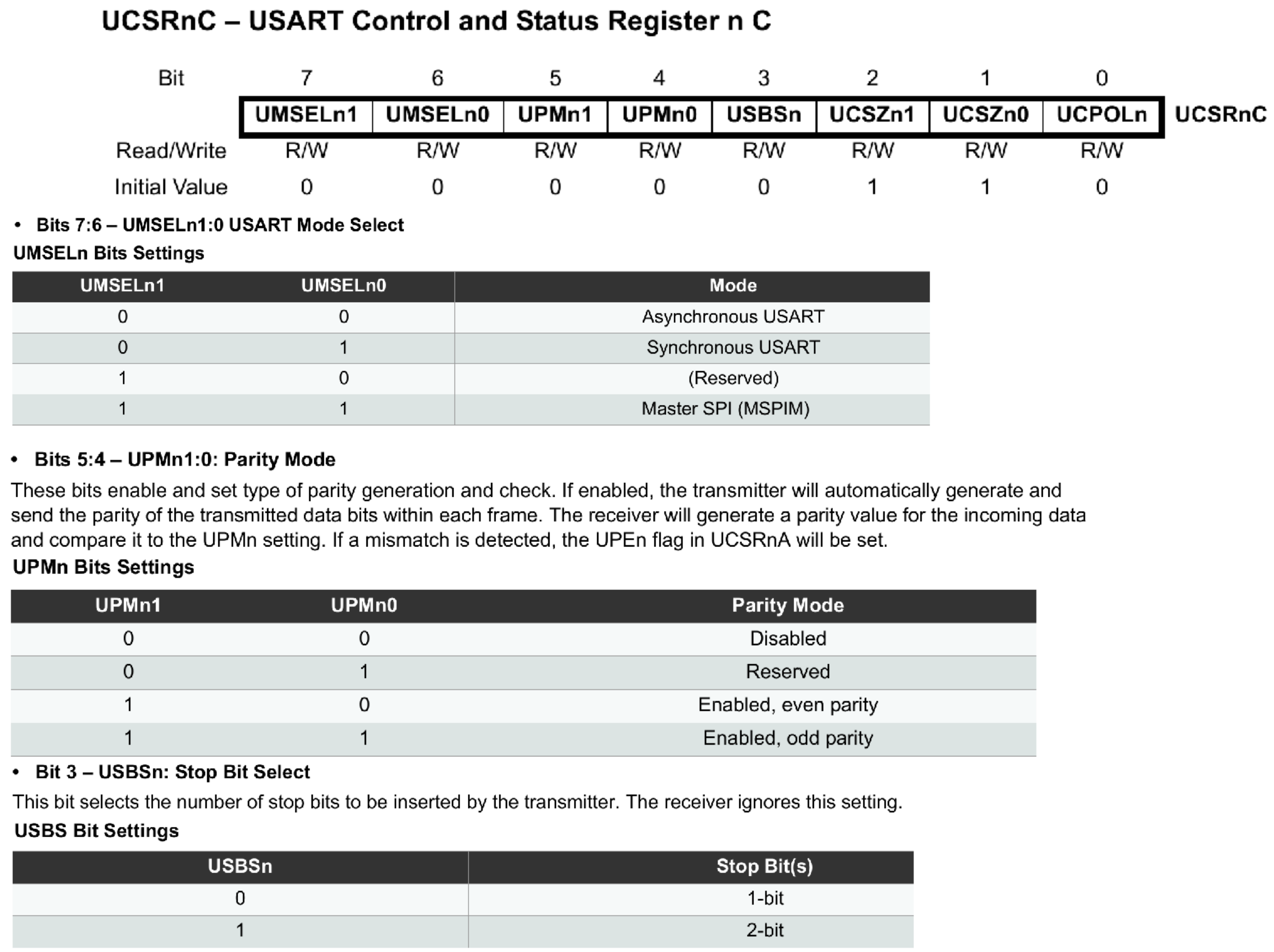
- **Bit 1 – RXB8n: Receive Data Bit 8 n**

RXB8n is the ninth data bit of the received character when operating with serial frames with nine data bits. Must be read before reading the low bits from UDRn.

- **Bit 0 – TXB8n: Transmit Data Bit 8 n**

TXB8n is the ninth data bit in the character to be transmitted when operating with serial frames with nine data bits. Must be written before writing the low bits to UDRn.

## USART0 Register: UCSR0C



### UCSRnC – USART Control and Status Register n C

| Bit | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 | |
|---|---|---|---|---|---|---|---|---|---|
| | UMSELn1 | UMSELn0 | UPMn1 | UPMn0 | USBSn | UCSZn1 | UCSZn0 | UCPOLn | UCSRnC |
| Read/Write | R/W | R/W | R/W | R/W | R/W | R/W | R/W | R/W | |
| Initial Value | 0 | 0 | 0 | 0 | 0 | 1 | 1 | 0 | |

- **Bits 7:6 – UMSELn1:0 USART Mode Select**

**UMSELn Bits Settings**

| UMSELn1 | UMSELn0 | Mode |
|---|---|---|
| 0 | 0 | Asynchronous USART |
| 0 | 1 | Synchronous USART |
| 1 | 0 | (Reserved) |
| 1 | 1 | Master SPI (MSPIM) |

- **Bits 5:4 – UPMn1:0: Parity Mode**

These bits enable and set type of parity generation and check. If enabled, the transmitter will automatically generate and send the parity of the transmitted data bits within each frame. The receiver will generate a parity value for the incoming data and compare it to the UPMn setting. If a mismatch is detected, the UPEn flag in UCSRnA will be set.

**UPMn Bits Settings**

| UPMn1 | UPMn0 | Parity Mode |
|---|---|---|
| 0 | 0 | Disabled |
| 0 | 1 | Reserved |
| 1 | 0 | Enabled, even parity |
| 1 | 1 | Enabled, odd parity |

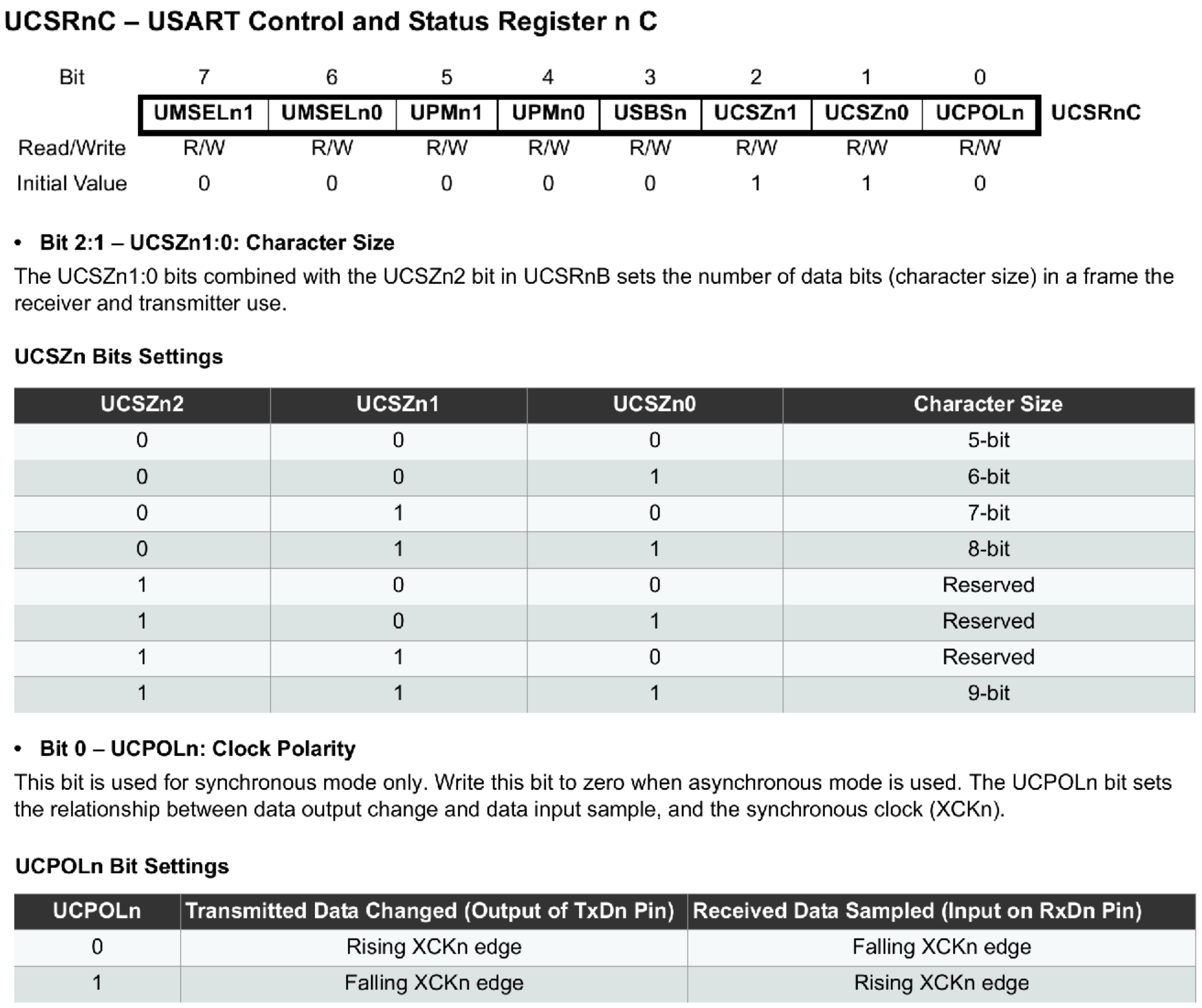
- **Bit 3 – USBSn: Stop Bit Select**

This bit selects the number of stop bits to be inserted by the transmitter. The receiver ignores this setting.

**USBS Bit Settings**

| USBSn | Stop Bit(s) |
|---|---|
| 0 | 1-bit |
| 1 | 2-bit |

## USART0 Register: UCSR0C



**UCSRnC – USART Control and Status Register n C**

| Bit | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 | |
|---|---|---|---|---|---|---|---|---|---|
| | UMSELn1 | UMSELn0 | UPMn1 | UPMn0 | USBSn | UCSZn1 | UCSZn0 | UCPOLn | UCSRnC |
| Read/Write | R/W | R/W | R/W | R/W | R/W | R/W | R/W | R/W | |
| Initial Value | 0 | 0 | 0 | 0 | 0 | 1 | 1 | 0 | |

- **Bit 2:1 – UCSZn1:0: Character Size**

The UCSZn1:0 bits combined with the UCSZn2 bit in UCSRnB sets the number of data bits (character size) in a frame the receiver and transmitter use.

**UCSZn Bits Settings**

| UCSZn2 | UCSZn1 | UCSZn0 | Character Size |
|---|---|---|---|
| 0 | 0 | 0 | 5-bit |
| 0 | 0 | 1 | 6-bit |
| 0 | 1 | 0 | 7-bit |
| 0 | 1 | 1 | 8-bit |
| 1 | 0 | 0 | Reserved |
| 1 | 0 | 1 | Reserved |
| 1 | 1 | 0 | Reserved |
| 1 | 1 | 1 | 9-bit |

- **Bit 0 – UCPOLn: Clock Polarity**

This bit is used for synchronous mode only. Write this bit to zero when asynchronous mode is used. The UCPOLn bit sets the relationship between data output change and data input sample, and the synchronous clock (XCKn).

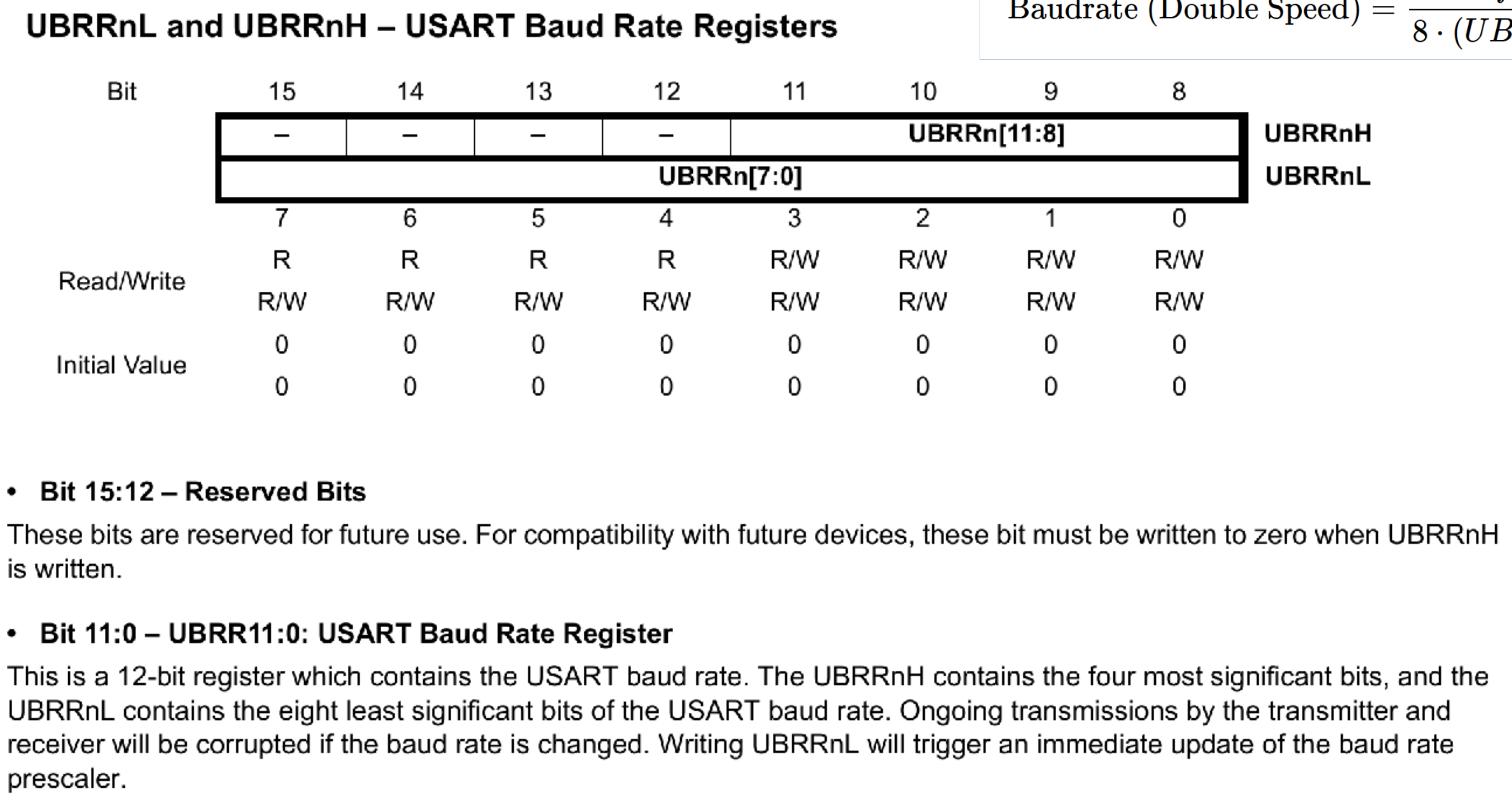
**UCPOLn Bit Settings**

| UCPOLn | Transmitted Data Changed (Output of TxDn Pin) | Received Data Sampled (Input on RxDn Pin) |
|---|---|---|
| 0 | Rising XCKn edge | Falling XCKn edge |
| 1 | Falling XCKn edge | Rising XCKn edge |

## USART0 Register: UBRR0

$$\text{Baudrate (Normal Speed)} = \frac{f_{CPU}}{16 \cdot (UBRR0 + 1)}$$

$$\text{Baudrate (Double Speed)} = \frac{f_{CPU}}{8 \cdot (UBRR0 + 1)}$$



**UBRRnL and UBRRnH – USART Baud Rate Registers**

| Bit | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | |
|---|---|---|---|---|---|---|---|---|---|
| | – | – | – | – | UBRRn[11:8] | | | | UBRRnH |
| | UBRRn[7:0] | | | | | | | | UBRRnL |
| | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 | |
| Read/Write | R | R | R | R | R/W | R/W | R/W | R/W | |
| | R/W | R/W | R/W | R/W | R/W | R/W | R/W | R/W | |
| Initial Value | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | |
| | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | |

- **Bit 15:12 – Reserved Bits**

These bits are reserved for future use. For compatibility with future devices, these bit must be written to zero when UBRRnH is written.

- **Bit 11:0 – UBRR11:0: USART Baud Rate Register**

This is a 12-bit register which contains the USART baud rate. The UBRRnH contains the four most significant bits, and the UBRRnL contains the eight least significant bits of the USART baud rate. Ongoing transmissions by the transmitter and receiver will be corrupted if the baud rate is changed. Writing UBRRnL will trigger an immediate update of the baud rate prescaler.

## Using printf() via UART

- Using the **AVR libc** and **AVR-GCC compiler toolchain**, the standard C `printf()` function can be redirected to USART by creating a custom `usart_putchar()` function and using:

  ```
  fdevopen(&usart_putchar, NULL);
  ```

- The user-defined usart_putchar() uses the **USART** module in asynchronous mode to send chars.
- This links **stdout** (standard output stream) to **USART** so that calls to `printf()` send text bytes to the **USART**.

## Code Example: USART – Serial TX Mode

**Code Snippet 1**

```
#define F_CPU 16000000UL
#include <avr/io.h>
#include <avr/interrupt.h>
#include <util/delay.h>
#include <stdio.h>

#define SPEED_2X
#define BAUD  115200

#ifdef SPEED_2X
#define UBRR_VALUE ((F_CPU / (8UL * BAUD)) - 1)
#else
#define UBRR_VALUE ((F_CPU / (16UL * BAUD)) - 1)
#endif

void usart_init(void) {
    // Set baud rate
    UBRR0H = (uint8_t)(UBRR_VALUE >> 8);
    UBRR0L = (uint8_t)UBRR_VALUE;
#ifdef SPEED_2X
    UCSR0A = (1 << U2X0);     // Enable double speed
#else
    UCSR0A = 0;
#endif
    UCSR0C = (1 << UCSZ01) | (1 << UCSZ00); // Use 8N1
    UCSR0B = (1 << RXEN0)  | (1 << TXEN0);  // Enable both TX & RX
}
// Code continues on the next page...
```

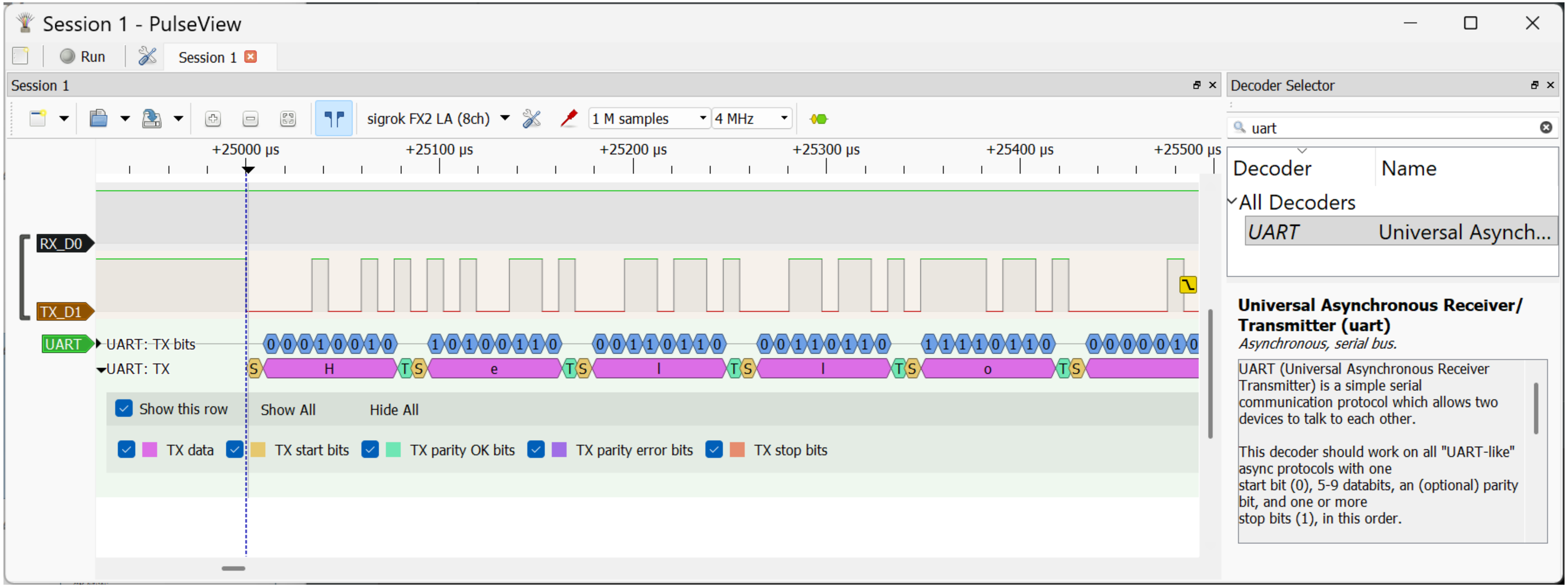
## Code Example: USART – Serial TX Mode

**Code Snippet 2**

```
void usart_putchar(char c) {
    while (!(UCSR0A & (1 << UDRE0))); // Wait TX buffer empty
    UDR0 = c; // Write char to the data register
}

void usart_send_string(const char *s) {
    while (*s) {
        usart_putchar(*s++);
    }
}

char usart_getchar(void) {
    while (!(UCSR0A & (1 << RXC0))); // Wait for RX incoming byte
    return UDR0;
}
```

```
int main(void) {
    usart_init();
    while (1) {
        usart_send_string("Hello from ATmega328P\r\n");
        _delay_ms(1000);   // 1 second blocking delay
    }
}
```

# USB Logic Analyzer + PulseView / UART Protocol Analyzer

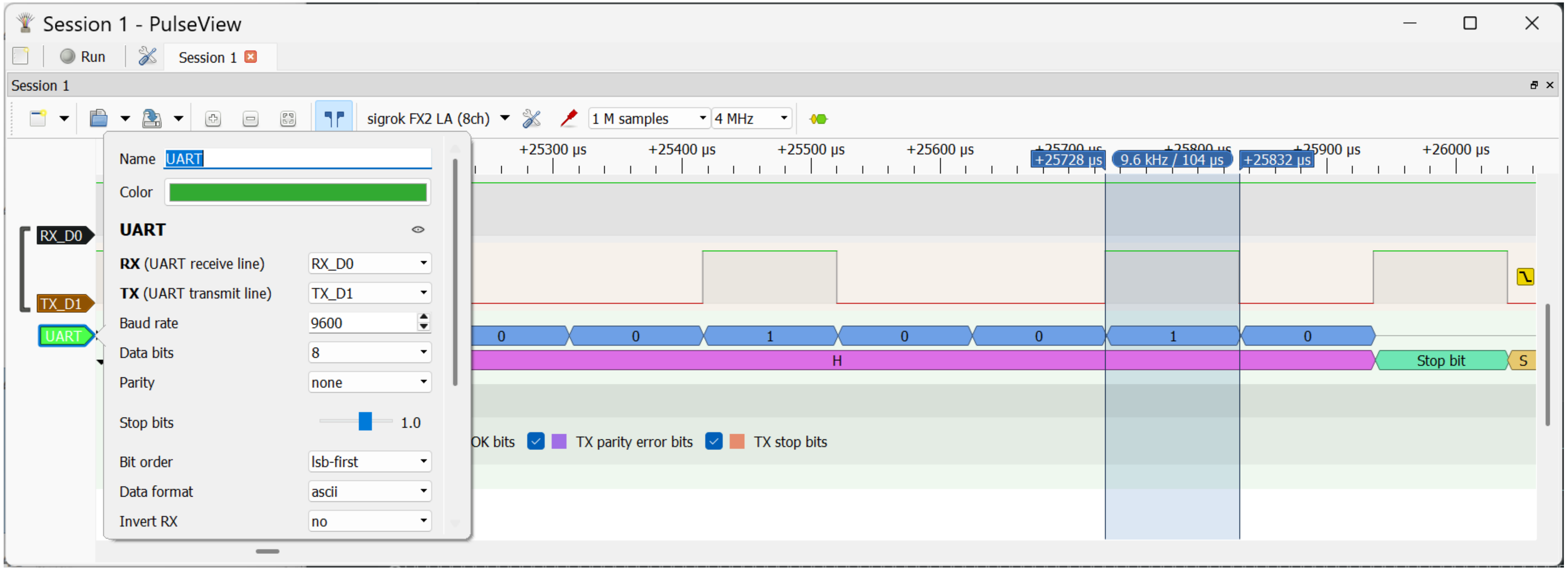


**Universal Asynchronous Receiver/ Transmitter (uart)**
*Asynchronous, serial bus.*

UART (Universal Asynchronous Receiver Transmitter) is a simple serial communication protocol which allows two devices to talk to each other.

This decoder should work on all "UART-like" async protocols with one start bit (0), 5-9 databits, an (optional) parity bit, and one or more stop bits (1), in this order.

# USB Logic Analyzer + PulseView / UART Protocol Analyzer



Session 1 - PulseView

Run | Session 1

sigrok FX2 LA (8ch) | 1 M samples | 4 MHz

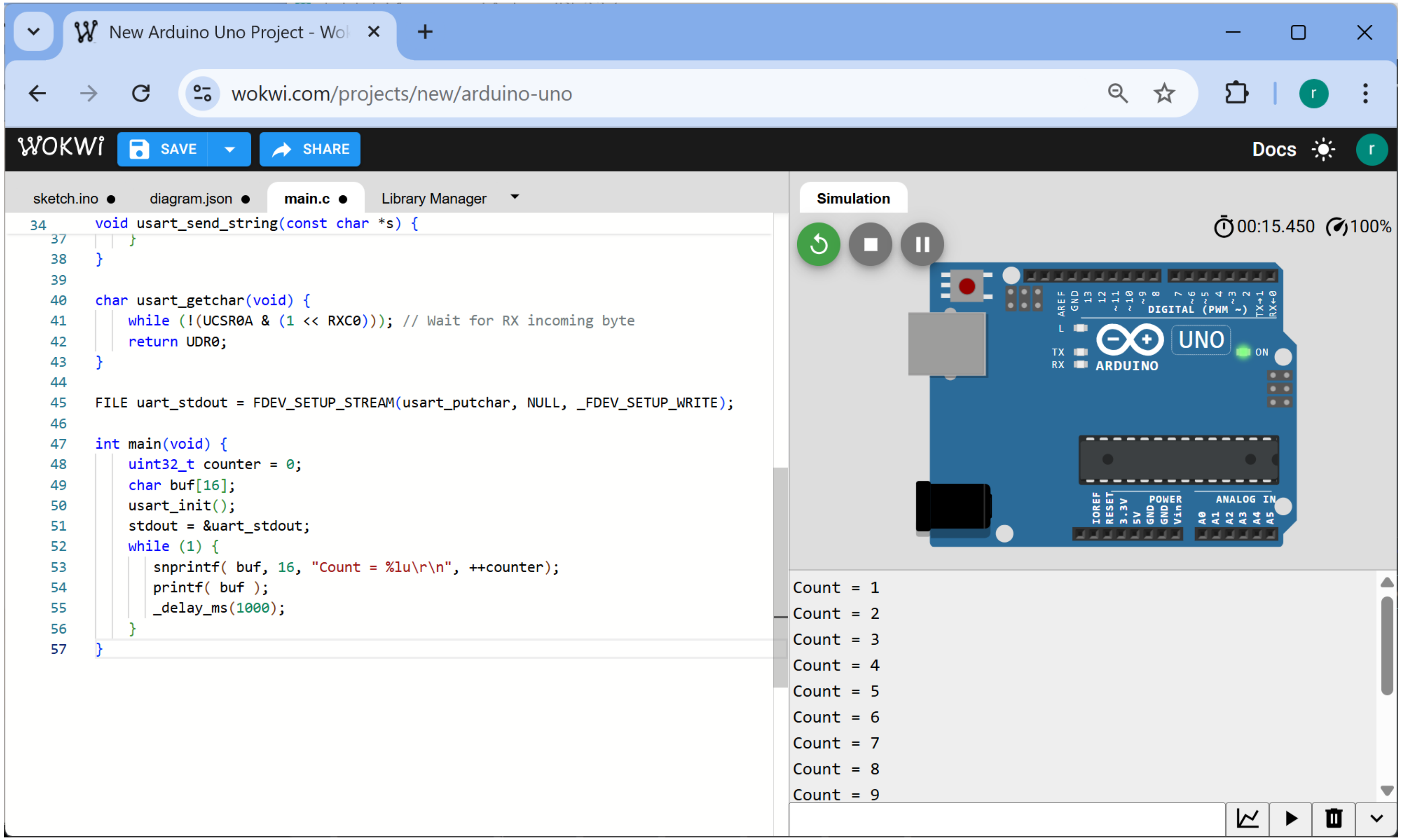
| | |
|---|---|
| Name | UART |
| Color | 🟩 |
| **UART** | |
| **RX** (UART receive line) | RX_D0 |
| **TX** (UART transmit line) | TX_D1 |
| Baud rate | 9600 |
| Data bits | 8 |
| Parity | none |
| Stop bits | 1.0 |
| Bit order | lsb-first |
| Data format | ascii |
| Invert RX | no |

**Baudrate: 9600**

## Code Example: printf() via USART

```
// Use Code Snippets 1 & 2

FILE uart_stdout = FDEV_SETUP_STREAM(usart_putchar, NULL, _FDEV_SETUP_WRITE);

int main(void) {
    uint32_t counter = 0;
    char buf[16];
    usart_init();
    stdout = &uart_stdout;
    while (1) {
       snprintf( buf, 16, "Count = %lu\r\n", ++counter);
       printf( buf );
       _delay_ms(1000);
    }
}
```

# Wokwi Simulator

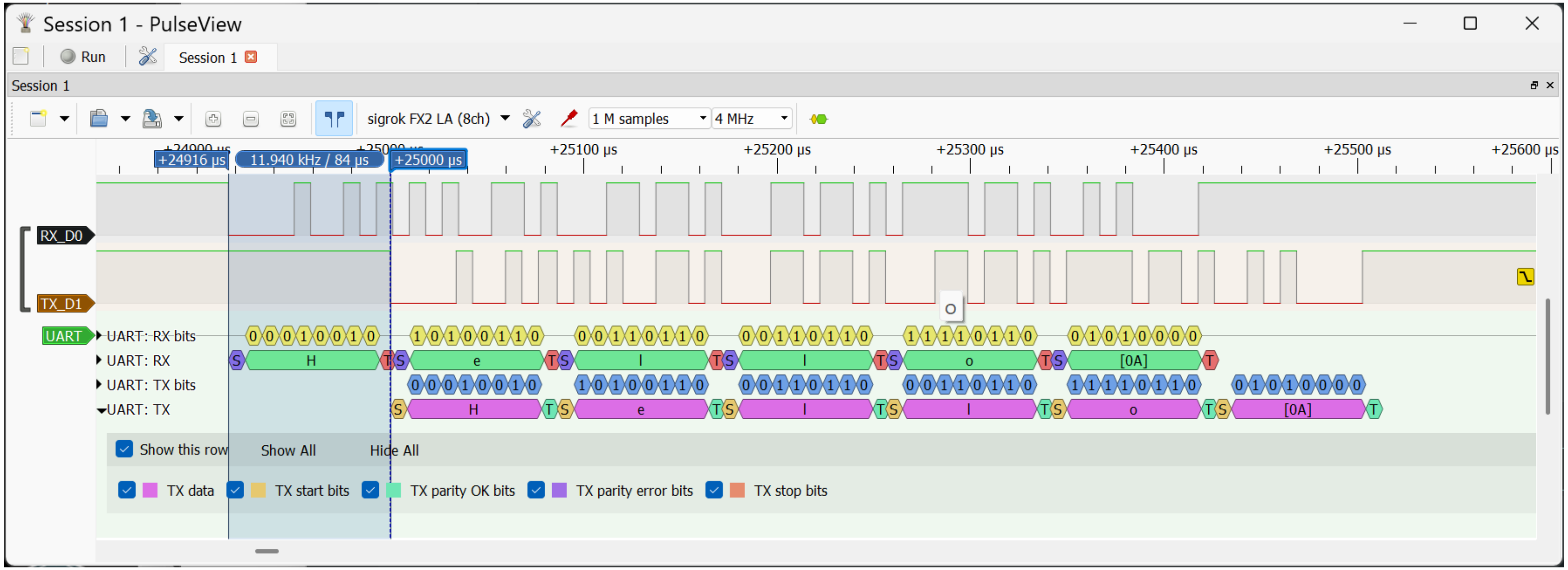


```c
void usart_send_string(const char *s) {
    }
}

char usart_getchar(void) {
    while (!(UCSR0A & (1 << RXC0))); // Wait for RX incoming byte
    return UDR0;
}

FILE uart_stdout = FDEV_SETUP_STREAM(usart_putchar, NULL, _FDEV_SETUP_WRITE);

int main(void) {
    uint32_t counter = 0;
    char buf[16];
    usart_init();
    stdout = &uart_stdout;
    while (1) {
        snprintf( buf, 16, "Count = %lu\r\n", ++counter);
        printf( buf );
        _delay_ms(1000);
    }
}
```

```
Count = 1
Count = 2
Count = 3
Count = 4
Count = 5
Count = 6
Count = 7
Count = 8
Count = 9
```

# Code Example: USART – Loopback Test (Polling)

```c
// Use Code Snippets 1 & 2
int main(void) {
    usart_init();
    while (1) { // Serial loopback test
        usart_putchar( usart_getchar() ); // Blocking calls
    }
}
```



## Code Example: USART – TX Mode (Interrupt-Driven)

```c
// Use Code Snippets 1 & 2

volatile const char *tx_buf = 0;
volatile uint8_t tx_busy = 0;

// Start TX using interrupt-driven method
void usart_send_string_async(const char *s) {
    tx_busy = 1;
    tx_buf = s;
    // Enable Data Register Empty interrupt
    UCSR0B |= (1 << UDRIE0);
}

// ISR for USART Data Register Empty
ISR(USART_UDRE_vect) {
    if (*tx_buf) {
       UDR0 = *tx_buf++; // Send next byte
    } else { // no more data to send
       tx_busy = 0;
       // Disable TX interrupt
       UCSR0B &= ~(1 << UDRIE0);
       tx_buf = 0;
    }
}
```

```c
int main(void) {
  const char *msg = "Hello from ATmega328\r\n";
  usart_init();

  // Enable UDRE interrupt
  UCSR0B |= (1 << UDRIE0);
  // Enable global interrupts
  sei();

  while (1) {
     if (!tx_busy) {
       usart_send_string_async( msg );
       _delay_ms(100);
     }
  }
}
```

## Code Example: USART – RX Buffer & RX Interrupt

```
// Use Code Snippets 1 & 2
#define UART_BUFSIZE (64)

volatile uint8_t rx_buf[UART_BUFSIZE];
volatile uint8_t rx_buf_tail = 0,
                 rx_buf_head = 0;

ISR(USART_RX_vect) { // ISR for UART-RX
  // Read the received data byte
  uint8_t data = UDR0;
  // Increment the head pointer
  uint8_t next_head
           = (rx_buf_head + 1) % UART_BUFSIZE;
  // If the buffer is not full,
  // save data byte into buffer.
  if (next_head != rx_buf_tail) {
    rx_buf[rx_buf_head] = data;
    rx_buf_head = next_head;
  }
}
```

```
uint8_t is_uart_buffer_empty() {
  return (rx_buf_head == rx_buf_tail);
}


uint8_t uart_read(void) {
  if (is_uart_buffer_empty()) {
    return 0;
  }
  uint8_t data = rx_buf[rx_buf_tail];
  rx_buf_tail = (rx_buf_tail + 1) % UART_BUFSIZE;
  return data;
}
```

## Code Example: USART – RX Buffer & RX Interrupt

```
int main(void) {
    usart_init();

    UCSR0B |= (1 << RXCIE0); // Enable USART RX interrupt
    sei();

    while (1) {
      if (!is_uart_buffer_empty()) {
        // Read one data byte from buffer and write it to UART
        usart_putchar( uart_read() );
      }
      _delay_ms(1);
    }
}
```

## Code Example: C++ Class for Simple Serial

**SimpleSerial.h**

```
#ifndef SIMPLE_SERIAL_H
#define SIMPLE_SERIAL_H

#include <avr/io.h>
#include <avr/interrupt.h>
#include <stdint.h>

#define UART_BUFSIZE 64

class SimpleSerial {
  public:
    SimpleSerial() {}  // Default constructor
    void begin(uint32_t baud);
    void write(char c);
    void print(const char* s);
    char read();
    bool available();
};

#endif
```

## Code Example: C++ Class for Simple Serial

**SimpleSerial.cpp**

```
#include "SimpleSerial.h"

static volatile uint8_t rx_buf[UART_BUFSIZE];
static volatile uint8_t rx_buf_head = 0;
static volatile uint8_t rx_buf_tail = 0;

void SimpleSerial::begin(uint32_t baud) {
    uint16_t ubrr_value = (F_CPU / (8UL * baud)) - 1;
    UBRR0H = (uint8_t)(ubrr_value >> 8);
    UBRR0L = (uint8_t)ubrr_value;
    UCSR0A = (1<<U2X0); // Double speed
    UCSR0C = (1<<UCSZ01) | (1<<UCSZ00); // 8N1
    // Enable the "RX Complete" interrupt
    UCSR0B = (1<<RXEN0) | (1<<TXEN0) | (1<<RXCIE0);
    sei(); // Enable global interrupts
}
void SimpleSerial::write(char c) {
    while (!(UCSR0A & (1 << UDRE0)));
    UDR0 = c;
}
```

## Code Example: C++ Class for Simple Serial

**SimpleSerial.cpp**

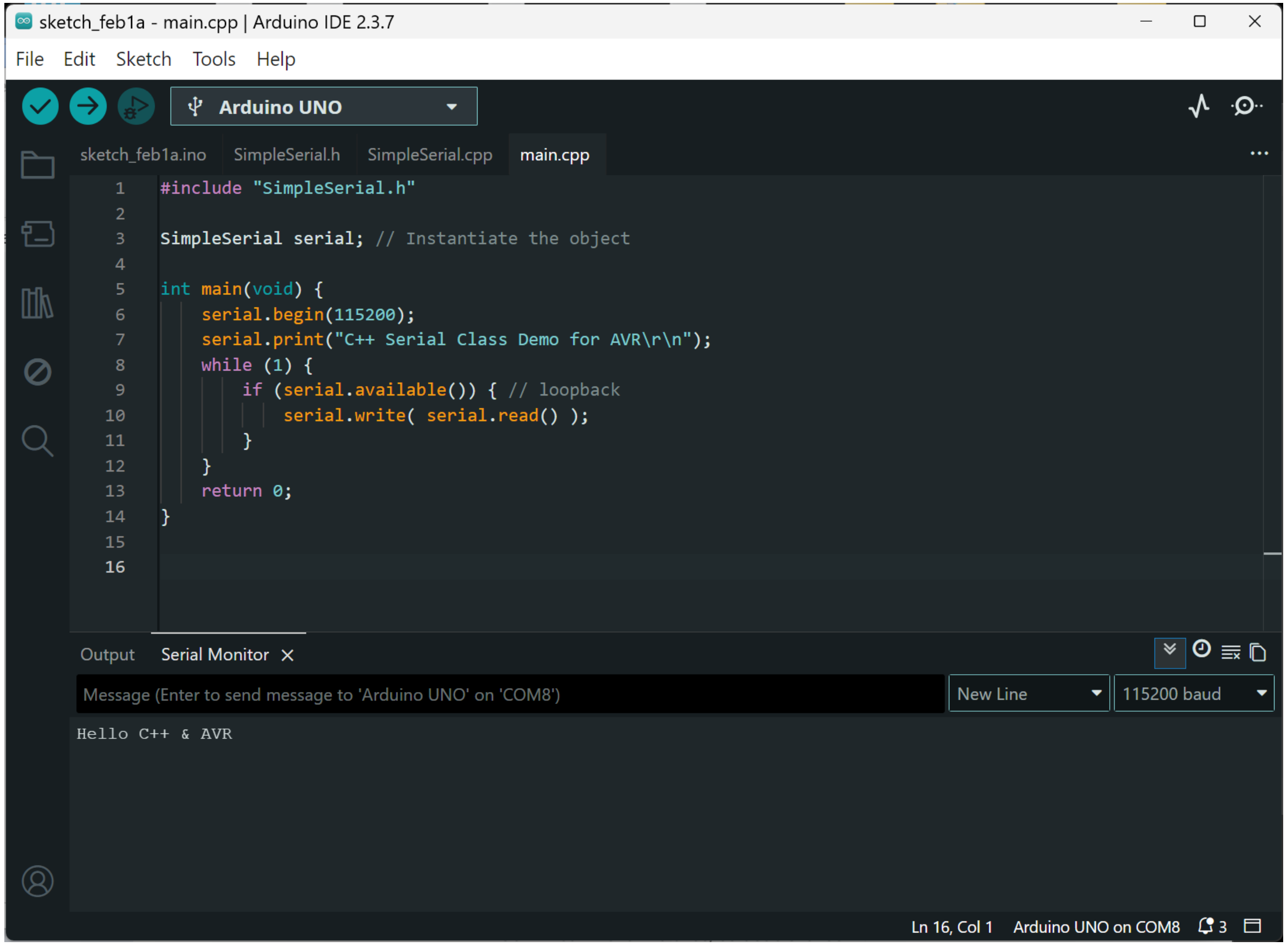
```
void SimpleSerial::print(const char* s) {
    while (*s) { write(*s++); }
}

bool SimpleSerial::available() { // Check if the RX buffer has data
    return (rx_buf_head != rx_buf_tail);
}

char SimpleSerial::read() {  // Pull data from the buffer
    if (rx_buf_head == rx_buf_tail) { return 0; }
    uint8_t data = rx_buf[rx_buf_tail];
    rx_buf_tail = (rx_buf_tail + 1) % UART_BUFSIZE;
    return (char)data;
}
ISR(USART_RX_vect) { // ISR for RX receive
    uint8_t data = UDR0;
    uint8_t next_head = (rx_buf_head + 1) % UART_BUFSIZE;
    if (next_head != rx_buf_tail) { // Check whether buffer not full
        rx_buf[rx_buf_head] = data;
        rx_buf_head = next_head;
    }
}
```

## Code Example: C++ Class for Simple Serial

main.cpp

```
#include "SimpleSerial.h"

SimpleSerial serial; // Instantiate the object

int main(void) {
    serial.begin(115200);
    serial.print("C++ Serial Class Demo for AVR\r\n");
    while (1) {
        if (serial.available()) { // serial loopback
            serial.write( serial.read() );
        }
    }
    return 0;
}
```

## Code Example: C++ Class for Simple Serial


sketch_feb1a - main.cpp | Arduino IDE 2.3.7
File Edit Sketch Tools Help
Arduino UNO
sketch_feb1a.ino SimpleSerial.h SimpleSerial.cpp main.cpp
Output Serial Monitor
Ln 16, Col 1 Arduino UNO on COM8 3

```cpp
#include "SimpleSerial.h"

SimpleSerial serial; // Instantiate the object

int main(void) {
    serial.begin(115200);
    serial.print("C++ Serial Class Demo for AVR\r\n");
    while (1) {
        if (serial.available()) { // loopback
            serial.write( serial.read() );
        }
    }
    return 0;
}

```

Message (Enter to send message to 'Arduino UNO' on 'COM8')  New Line  115200 baud

```
Hello C++ & AVR
```

In Arduino IDE, you can have multiple source code files:

- sketch.ino (empty)
- SimpleSerial.h
- SimpleSerial.cpp
- main.cpp