GENERAL

# atmega328 uart assembly code example

Published: September 19, 2026 | Index ID: #-

## Introduction

When working with the ATmega328 microcontroller, one of the most common communication methods is UART (Universal Asynchronous Receiver/Transmitter). Whether you're building a simple serial console, creating a sensor network, or interfacing with a computer, mastering UART in assembly gives you fine-grained control over timing, data integrity, and power consumption. In this comprehensive guide, we'll walk through a complete ATmega328 UART assembly code example, explain every register and instruction, and provide practical tips for debugging and optimization. By the end, you'll be comfortable writing your own UART routines in AVR assembly and tailoring them to your project's needs.

## Why Use Assembly for UART on the ATmega328?

High-level languages like C offer convenience, but assembly gives you:

- Precise Timing Control – Essential for high-speed serial communication or when working with custom baud rates.
- Minimal Overhead – No function call overhead or stack usage, leading to smaller code size.
- Direct Register Access – You can manipulate individual bits and flags instantly.

For hobbyists and professionals alike, understanding the assembly foundation of UART enhances debugging skills and opens doors to more advanced microcontroller projects.

## UART Basics on the ATmega328

### What Is UART?

UART is a serial communication protocol that transmits data one bit at a time over a single line. It uses start, stop, and optional parity bits to frame each byte. The ATmega328's UART module, often referred to as USART (Universal Synchronous/Asynchronous Receiver/Transmitter), supports both synchronous and asynchronous modes.

### Key UART Parameters

- Baud Rate – The speed of data transmission (e.g., 9600, 115200).
- Data Bits – Usually 8 bits per byte.
- Parity – None, even, or odd.
- Stop Bits – 1 or 2 stop bits.
- Clock Polarity – Determines when bits are sampled.

In this article, we'll focus on asynchronous mode with 8 data bits, no parity, and 1 stop bit – the most common configuration.

## ATmega328 UART Registers Overview

Understanding the registers is crucial before writing code. Below is a quick reference:

- UBRR0H / UBRR0L – Baud rate registers (high and low).
- UCSR0A – Status register (TXC0, RXC0, UDRE0, etc.).

- UCSR0B – Control register (RXEN0, TXEN0, RXCIE0, TXCIE0, UDRIE0).
- UCSR0C – Control register (UMSEL01/00, UPM01/00, USBS0, UCSZ01/00).
- UDR0 – Data register (read/write).

Each bit in these registers has a specific function, and setting them correctly is the foundation of reliable UART communication.

## Setting Up the UART: Register Configuration

### Choosing a Baud Rate

The ATmega328's datasheet provides a formula to calculate the UBRR value:

**UBRR = (F_CPU / (16 × Baud)) – 1**

For example, with a 16 /MHz clock and a desired baud rate of 9600:

**UBRR = (16,000,000 / (16 × 9600)) – 1 "H 103**

We'll load 103 into UBRR0H and UBRR0L.

### Configuring Data Frame Settings

We want 8 data bits, no parity, and 1 stop bit. In UCSR0C, we set:

- UMSEL01:0 = 0 (Asynchronous)
- UPM01:0 = 0 (Parity disabled)
- USBS0 = 0 (1 stop bit)
- UCSZ01:0 = 3 (8 data bits)

These bits are set by writing to UCSR0C with the appropriate mask.

### Enabling Transmitter and Receiver

In UCSR0B, we enable the receiver (RXEN0) and transmitter (TXEN0). If we plan to use interrupts, we can also enable RXCIE0 and TXCIE0. For a simple polling example, we'll leave interrupts disabled.

## Writing the UART Assembly Code Example

Below is a complete assembly program that initializes the UART and then echoes any received byte back to the sender. The code uses the AVR instruction set and assumes the use of the GNU AVR toolchain. It's written for clarity rather than extreme optimization, making it an excellent learning resource.

### 1. Include Headers and Define Constants

We'll start by defining the CPU frequency and baud rate constants, then calculate the UBRR value.

**Example:**

<code>\n.equ F_CPU = 16000000\n.equ BAUD = 9600\n.equ UBRR_VAL = (F_CPU / (16 * BAUD)) - 1\n</code>

### 2. Set Up the Stack

Even though our example is simple, it's good practice to initialize the stack pointer.

<code>\nldi r16, high(RAMEND)\nout SPL, r16\nldi r16, low(RAMEND)\nout SPL, r16\n</code>

### 3. UART Initialization Routine

This routine writes the calculated UBRR value to UBRR0L and UBRR0H, configures UCSR0C for 8 N 1, and enables the transmitter and receiver.

<code>\nUART_INIT:\n ; Load UBRR value\n ldi r16, low(UBRR_VAL)\n out UBRR0L, r16\n ldi r16, high(UBRR_VAL)\n out UBRR0H, r16\n\n ; Configure frame format: 8 data bits, no parity, 1 stop bit\n ldi r16, (1

### 4. Transmit Byte Routine

The routine waits until the UDRE0 flag is set, indicating that the transmit buffer is empty, then writes the byte to UDR0.

<code>\nUART_TX:\n ; Wait for empty transmit buffer\n sbis UCSR0A, UDRE0\n rjmp UART_TX\n\n ; Load data into UDR0\n out UDR0, r16\n ret\n</code>

### 5. Receive Byte Routine

This routine waits until a byte is available (RXC0 flag set), then reads it from UDR0.

<code>\nUART_RX:\n ; Wait for data to be received\n sbis UCSR0A, RXC0\n rjmp UART_RX\n\n ; Read data from UDR0\n in r16, UDR0\n ret\n</code>

### 6. Main Loop – Echo Received Bytes

In the main loop, we continuously receive a byte and immediately transmit it back.

<code>\nMAIN_LOOP:\n rcall UART_RX ; Receive a byte into r16\n rcall UART_TX ; Transmit the same byte\n rjmp MAIN_LOOP\n</code>

### 7. Full Program Assembly

Combining all the pieces, here’s the complete assembly listing.

<code>\n.include "m328pdef.inc"\n\n.equ F_CPU = 16000000\n.equ BAUD = 9600\n.equ UBRR_VAL = (F_CPU / (16 * BAUD)) - 1\n\n.org 0x0000\n rjmp RESET\n\nRESET:\n ; Initialize stack\n ldi r16, high(RAMEND)\n out SPL, r16\n ldi r16, low(RAMEND)\n out SPL, r16\n\n ; Initialize UART\n rcall UART_INIT\n\nMAIN_LOOP:\n rcall UART_RX\n rcall UART_TX\n rjmp MAIN_LOOP\n\nUART_INIT:\n ldi r16, low(UBRR_VAL)\n out UBRR0L, r16\n ldi r16, high(UBRR_VAL)\n out UBRR0H, r16\n ldi r16, (1

## Step by Step Explanation of the Code

### 1. Header and Constants

The .include directive pulls in the ATmega328 register definitions. Defining F_CPU and BAUD makes the code portable – you can change the clock speed or baud rate without hunting down hard coded numbers.

### 2. Reset Vector and Stack Setup

The program starts at address 0x0000, the reset vector. We set the stack pointer to the top of SRAM (RAMEND) to avoid overwriting data.

### 3. UART Initialization

## Baca Juga (Artikel Terkait)

- **advanced engineering electromagnetics solution manual edition hardcover by balanis constantine a published by wiley**

  URL: http://dtrainderailed.com/advanced-engineering-electromagnetics-solution-manual-edition-hardcover-by-balanis-constantine-a-published-by-wiley.pdf

- **advanced engineering electromagnetics solutions manual**

  URL: http://dtrainderailed.com/advanced-engineering-electromagnetics-solutions-manual.pdf

- **advanced engineering electromagnetics wiley 1989 grading**

  URL: http://dtrainderailed.com/advanced-engineering-electromagnetics-wiley-1989-grading.pdf

- **advanced engineering fluid mechanics by biswas**

  URL: http://dtrainderailed.com/advanced-engineering-fluid-mechanics-by-biswas.pdf