
BIRZEIT UNIVERSITY

Department of Electrical & Computer Engineering
ENCS4110 – Computer Design Laboratory

**Experiment 09**

# UART Serial Communication

**Date:** November 2025

# 9 UART Serial Communication

## Learning Objectives

After completing this experiment, you will be able to:

- Understand the principles of asynchronous serial communication using UART.
- Differentiate between synchronous and asynchronous communication protocols.
- Configure UART modules on the TM4C123 microcontroller for serial data transmission.
- Calculate and configure baud rate divisors for different communication speeds.
- Implement GPIO alternate function configuration for UART pins.
- Transmit and receive data through UART using polling and FIFO buffers.
- Create a reusable UART driver library with initialization, transmit, and receive functions.
- Interface with a PC terminal application for debugging and user interaction.
- Implement string-based communication protocols for command processing.

## Experiment Overview

Serial communication is one of the most fundamental methods for data exchange between microcontrollers and external devices. The Universal Asynchronous Receiver-Transmitter (UART) is a widely-used hardware communication protocol that enables simple, bidirectional, full-duplex serial data transmission without requiring a shared clock signal between devices.

UART is particularly valuable in embedded systems because of its simplicity, robustness, and widespread support across microcontrollers, sensors, GPS modules, Bluetooth modules, and debugging interfaces. Unlike synchronous protocols (SPI, $I^2C$), UART does not need a clock line — both devices independently use agreed-upon timing parameters (baud rate) to synchronize data transmission.

The TM4C123 microcontroller features eight UART modules (UART0-UART7), each capable of supporting standard baud rates from 300 bps to 5 Mbps. UART0 is commonly used for debugging and communication with PC terminal applications through a USB-to-serial converter integrated into the TM4C123 LaunchPad.

In this experiment, you will:

- Learn the fundamentals of UART frame structure, timing, and baud rate calculation.
- Understand the role of UART hardware registers and FIFOs.
- Configure GPIO pins for UART alternate function (AFSEL, PCTL).
- Implement initialization routines for UART0 with specific baud rates and data formats.
- Transmit characters and strings from the microcontroller to a PC terminal.
- Receive characters and strings from a PC terminal with echo functionality.
- Debug embedded applications using serial output for status messages and variable monitoring.

By the end of this lab, you will be proficient in UART configuration and communication, enabling you to interface the TM4C123 with PCs, sensors, and other microcontrollers for data exchange and debugging.

# Contents

# 1 Theoretical Background

## 1.1 UART Communication Fundamentals

UART (Universal Asynchronous Receiver-Transmitter) is a hardware communication protocol that converts parallel data from a microcontroller's bus into serial format for transmission and vice versa for reception.

### 1.1.1 Synchronous vs. Asynchronous Communication

#### Synchronous Communication

- **Shared clock**: Transmitter and receiver use a common clock signal (e.g., SPI's SCK, I²C's SCL)
- **Advantages**: Higher data rates, precise timing, simpler framing
- **Disadvantages**: Requires additional clock line, limited cable length due to clock skew

#### Asynchronous Communication (UART)

- **No shared clock**: Each device uses its own independent clock
- **Timing synchronization**: Uses start/stop bits and agreed-upon baud rate
- **Advantages**: Only 2 wires needed (TX, RX), longer cable lengths, simpler hardware
- **Disadvantages**: Lower maximum data rates, requires precise clock accuracy, overhead from start/stop bits

### 1.1.2 UART Data Frame Structure

A UART transmission consists of a series of frames, each containing:

1. **Idle State**: Line held HIGH (logic 1) when no data is being transmitted
2. **Start Bit**: A single LOW (logic 0) bit signals the beginning of a frame
3. **Data Bits**: 5 to 9 bits of actual data (typically 8 bits for byte transmission)
4. **Parity Bit** (optional): Even, odd, or no parity for error detection
5. **Stop Bits**: 1, 1.5, or 2 HIGH bits marking the end of the frame

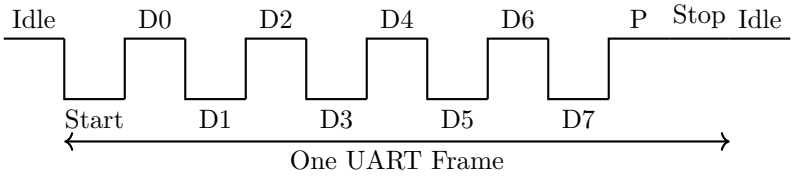


Figure 9.1: UART Frame Structure (8 data bits, 1 parity bit, 1 stop bit)

#### Common UART Configuration: 8N1

- 8 data bits
- No parity (N)
- 1 stop bit
- Most widely used configuration for simplicity and efficiency

### 1.1.3 Baud Rate

The **baud rate** is the rate at which symbols (bits) are transmitted per second, measured in bits per second (bps). Common baud rates include:

- **9600 bps**: Slow, highly reliable, long cable lengths

- **115200 bps**: Fast, common for debugging and short cables
- **Other rates**: 4800, 19200, 38400, 57600, 230400, 460800, 921600 bps

Both transmitter and receiver must use the *exact same* baud rate. Even small differences (> 2-3%) can cause communication errors due to timing drift over the duration of a frame.

### 1.1.4 Full-Duplex Communication

UART supports **full-duplex** communication, meaning data can be transmitted and received simultaneously on separate lines:

- **TX (Transmit)**: Output pin for sending data
- **RX (Receive)**: Input pin for receiving data
- **Connection**: TX of device A connects to RX of device B, and vice versa
- **Ground**: Shared ground reference is essential for proper voltage level detection

## 1.2 TM4C123 UART Architecture

The TM4C123 microcontroller includes eight UART modules (UART0-UART7) with advanced features:

### 1.2.1 UART Features

- **Baud rate generation**: Programmable from 300 bps to 5 Mbps
- **Data formats**: 5, 6, 7, 8, or 9 data bits
- **Parity**: Even, odd, or no parity
- **Stop bits**: 1 or 2 stop bits
- **FIFOs**: 16-byte transmit and receive FIFOs with programmable trigger levels
- **Interrupts**: Transmit, receive, overrun, framing error, parity error, break detection
- **DMA support**: Direct Memory Access for high-throughput transfers
- **IrDA and 9-bit modes**: Advanced communication modes

### 1.2.2 UART Pin Mapping

Each UART module is mapped to specific GPIO pins through alternate function configuration:

| UART Module | TX Pin | RX Pin | Notes |
|---|---|---|---|
| UART0 | PA1 | PA0 | Connected to USB-to-serial on LaunchPad |
| UART1 | PB1 | PB0 | Available on expansion headers |
| UART2 | PD7 | PD6 | Available on expansion headers |
| UART3 | PC7 | PC6 | Available on expansion headers |
| UART4 | PC5 | PC4 | Available on expansion headers |
| UART5 | PE5 | PE4 | Available on expansion headers |
| UART6 | PD5 | PD4 | Available on expansion headers |
| UART7 | PE1 | PE0 | Available on expansion headers |

Table 9.1: TM4C123 UART Pin Assignments

In this experiment, we use **UART0** (PA0/PA1) because it is directly connected to the on-board USB-to-serial converter, allowing easy communication with a PC terminal without additional hardware.

## 1.3 UART Registers

The UART modules are controlled through memory-mapped registers. Key registers include:

### 1.3.1 UARTDR — Data Register

**Register:** `UARTx->DR` — UART Data (`Base + 0x000`)
The Data Register is used for both transmitting and receiving data:

- **Write**: Places data in the transmit FIFO (bits 7:0 contain the byte to transmit)
- **Read**: Retrieves data from the receive FIFO (bits 7:0 contain the received byte)
- **Error flags** (bits 11:8 on read): Framing Error (FE), Parity Error (PE), Break Error (BE), Overrun Error (OE)

### 1.3.2 UARTFR — Flag Register

**Register:** `UARTx->FR` — UART Flag Register (`Base + 0x018`)
The Flag Register provides status information about the UART:

| Bit | Name | Description |
|---|---|---|
| 0 | CTS | Clear To Send (hardware flow control, not commonly used) |
| 3 | BUSY | UART busy transmitting data |
| 4 | RXFE | Receive FIFO Empty (1 = empty, 0 = data available) |
| 5 | TXFF | Transmit FIFO Full (1 = full, wait before writing) |
| 6 | RXFF | Receive FIFO Full |
| 7 | TXFE | Transmit FIFO Empty (1 = empty, transmission complete) |

Table 9.2: UARTFR Register Bits

### 1.3.3 UARTIBRD and UARTFBRD — Baud Rate Divisors

**Registers:** `UARTx->IBRD` and `UARTx->FBRD`
The baud rate is determined by dividing the UART clock by a divisor:

$$\text{Baud Rate} = \frac{f_{\text{UART clock}}}{16 \times (\text{IBRD} + \text{FBRD})}$$

Where:

- **IBRD**: Integer part of the baud rate divisor (16-bit value)
- **FBRD**: Fractional part of the baud rate divisor (6-bit value, represents fraction/64)

**Calculation Steps:**

1. Calculate divisor: $\text{Divisor} = \frac{f_{\text{UART clock}}}{16 \times \text{Baud Rate}}$
2. Integer part: $\text{IBRD} = \lfloor \text{Divisor} \rfloor$
3. Fractional part: $\text{FBRD} = \text{round}((\text{Divisor} - \text{IBRD}) \times 64 + 0.5)$

**Example: 115200 baud at 50 MHz system clock**

$$\text{Divisor} = \frac{50{,}000{,}000}{16 \times 115{,}200} = \frac{50{,}000{,}000}{1{,}843{,}200} \approx 27.126$$

$$\text{IBRD} = 27, \quad \text{FBRD} = \text{round}(0.126 \times 64 + 0.5) = \text{round}(8.564) = 8$$

### 1.3.4 UARTLCRH — Line Control Register

**Register:** `UARTx->LCRH` — UART Line Control (`Base + 0x02C`)
The Line Control Register configures the frame format:

| Bits | Name | Description |
|---|---|---|
| 0 | BRK | Send break (force TX low) |
| 1 | PEN | Parity Enable (0 = no parity, 1 = parity enabled) |
| 2 | EPS | Even Parity Select (0 = odd, 1 = even) |
| 3 | STP2 | Two Stop Bits (0 = 1 stop bit, 1 = 2 stop bits) |
| 4 | FEN | FIFO Enable (0 = disable FIFOs, 1 = enable FIFOs) |
| 6:5 | WLEN | Word Length: 00 = 5 bits, 01 = 6 bits, 10 = 7 bits, 11 = 8 bits |
| 7 | SPS | Stick Parity Select |

Table 9.3: UARTLCRH Register Bits

**Common configuration (8N1 with FIFOs):**

```
UART0->LCRH = (0x3 << 5) | (1 << 4);  // 8 data bits, FIFOs enabled
// Or: UART0->LCRH = 0x60;
```

### 1.3.5 UARTCTL — Control Register

**Register:** `UARTx->CTL` — UART Control (`Base + 0x030`)
The Control Register enables/disables the UART and its transmit/receive functions:

| Bit | Name | Description |
|---|---|---|
| 0 | UARTEN | UART Enable (0 = disable, 1 = enable) |
| 8 | TXE | Transmit Enable (0 = disable TX, 1 = enable TX) |
| 9 | RXE | Receive Enable (0 = disable RX, 1 = enable RX) |

Table 9.4: UARTCTL Register Key Bits

**Enable UART with TX and RX:**

```
UART0->CTL = (1 << 0) | (1 << 8) | (1 << 9);  // Enable UART, TX, RX
// Or: UART0->CTL = 0x301;
```

### 1.3.6 UARTCC — Clock Configuration

**Register:** `UARTx->CC` — UART Clock Configuration (`Base + 0xFC8`)
Selects the clock source for the UART module:

- `0x0`: System clock (default and most common)
- `0x5`: PIOSC (16 MHz internal oscillator)

## 1.4 UART Configuration Workflow

The following steps configure UART0 for basic serial communication:

1. **Enable clocks**: Enable UART0 and GPIOA clocks in `SYSCTL->RCGCUART` and `SYSCTL->RCGCGPIO`
2. **Wait for clock stabilization**: Check ready bits or insert delay
3. **Disable UART**: Clear `UART0->CTL` during configuration
4. **Configure GPIO for UART alternate function**:
   - Enable alternate function: `GPIOA->AFSEL |= 0x03` (PA0, PA1)
   - Set port control: `GPIOA->PCTL |= 0x11` (UART function on PA0/PA1)
   - Enable digital: `GPIOA->DEN |= 0x03`
5. **Set baud rate divisors**: Write `UART0->IBRD` and `UART0->FBRD`
6. **Configure line control**: Set data bits, parity, stop bits, enable FIFOs in `UART0->LCRH`
7. **Select clock source**: `UART0->CC = 0` (system clock)
8. **Enable UART**: Set `UART0->CTL` to enable UART, TX, and RX

# 2 Procedure

## 2.1 Example: Basic UART Driver Implementation

The following code demonstrates a complete UART driver for UART0 with initialization, character/string transmission, and character/string reception.

### 2.1.1 UART Header File

```
#ifndef UART_H
#define UART_H

#include "TM4C123.h"  // Or your MCU's main header

#define U0_TX 2
#define U0_RX 1

void UART0_WriteChar(char c);
void UART0_WriteString(char *str);

char UART0_ReadChar();
void UART0_ReadString(char *buffer, int maxLen);

void UART0_Init();




#endif // UART_H
```

Listing 9.1: UART driver header file (`uart.h`)

### 2.1.2 UART Implementation File

```
#include "uart.h"
#define MAX_STR_LEN 50
// Function to send a single character via UART0
void UART0_WriteChar(char c) {
    // Wait until the transmit FIFO is not full
    // UART0->FR bit 5 (TXFF) = 1 means FIFO is full, so wait until it becomes 0
    while ((UART0->FR & (1 << 5)) != 0);
    // Write the character to the Data Register to transmit
    UART0->DR = c;
}

// Function to send a null-terminated string via UART0
void UART0_WriteString(char *str) {
    // Loop through each character until null terminator
    while (*str) {
        // Send each character using UART0_WriteChar
        UART0_WriteChar(*(str++));
    }
}

// UART0 initialization function
```

```c
void UART0_Init() {
    // Enable clock to UART0 module (bit 0)
    SYSCTL->RCGCUART |= (1 << 0);
    // Enable clock to GPIO Port A (bit 0)
    SYSCTL->RCGCGPIO |= (1 << 0);

    // Enable alternate function on PA0 (RX) and PA1 (TX)
    GPIOA->AFSEL |= U0_RX | U0_TX;
    // Configure PA0 and PA1 pins for UART function in Port Control Register
    // U0_RX corresponds to PA0 (bits 3:0) and U0_TX corresponds to PA1 (bits 7:4)
    GPIOA->PCTL |= (1 << 0) | (1 << 4);
    // Enable digital function for PA0 and PA1 pins
    GPIOA->DEN |= U0_RX | U0_TX;

    // Disable UART0 while configuring
    UART0->CTL = 0;
    // Set integer baud rate divisor for 115200 baud with 50MHz clock
    UART0->IBRD = 27;
    // Set fractional baud rate divisor
    UART0->FBRD = 8;
    // Configure Line Control for 8 data bits, no parity, one stop bit, and FIFOs
    enabled
    // 0x60 = 0b01100000: bit 6 (FEN) = 1 enable FIFO, bits 5-6 (WLEN) = 11 for 8
    bits
    UART0->LCRH = 0x60;
    // Use system clock for UART
    UART0->CC = 0;
    // Enable UART0, TX and RX
    // Bit 0 = UARTEN, bit 8 = TXE, bit 9 = RXE
    UART0->CTL = (1 << 0) | (1 << 8) | (1 << 9);
}

char UART0_ReadChar(void) {
    while (UART0->FR & (1 << 4));  // Wait while RX FIFO empty
    return (char)(UART0->DR & 0xFF);
}

void UART0_ReadString(char *buffer, int maxLen) {
    int i = 0;
    char c;

    while (i < (maxLen - 1)) {   // Leave space for null terminator
        c = UART0_ReadChar();

        // Echo character back (optional)
        UART0_WriteChar(c);

        if (c == '\r' || c == '\n') {  // End of input
      UART0_WriteString("\r\n");
            break;
        }

        buffer[i++] = c;
    }

    buffer[i] = '\0';  // Null terminate the string
```

```
}
```

Listing 9.2: UART driver implementation (`uart.c`)

### 2.1.3 Main Application

```
#include <stdio.h>
#include "TM4C123.h"
#include "uart.h"

int main(void) {
    UART0_Init();
    UART0_WriteString("Hello World!\r\n");  // Send greeting with newline

    while (1) {
    char buff[16];
    UART0_ReadString(buff, 16);

    UART0_WriteString("Recived: ");
    UART0_WriteString(buff);
    UART0_WriteString("\r\n");
    }
}
```

Listing 9.3: Main application using UART driver (`main.c`)

## 2.2 Code Explanation

**UART Initialization** The `UART0_Init()` function configures UART0 for 115200 baud, 8N1 format with FIFOs enabled:

1. Enables UART0 and GPIOA clocks
2. Configures PA0 (RX) and PA1 (TX) for UART alternate function using AFSEL, PCTL, and DEN
3. Disables UART0 during configuration
4. Sets baud rate divisors: IBRD = 27, FBRD = 8 for 115200 baud at 50 MHz
5. Configures line control: 8 data bits (`WLEN = 0x3`), FIFOs enabled (`FEN = 1`)
6. Selects system clock as UART clock source
7. Enables UART0 with transmit and receive functions

**Transmitting Data** The `UART0_WriteChar()` function sends a single character:

- Polls the TXFF flag (bit 5) in the FR register
- Waits while TXFF = 1 (transmit FIFO full)
- Writes character to DR register when FIFO has space
- Hardware automatically handles framing (start bit, data bits, stop bit)

The `UART0_WriteString()` function sends a null-terminated string by repeatedly calling `UART0_WriteChar()`.

**Receiving Data** The `UART0_ReadChar()` function receives a single character:

- Polls the RXFE flag (bit 4) in the FR register
- Waits while RXFE = 1 (receive FIFO empty)
- Reads character from DR register when data is available
- Masks to 8 bits (`0xFF`) to extract data byte

The `UART0_ReadString()` function reads characters until Enter (`\r` or `\n`), with optional echo.

**Main Application** The main program demonstrates:

1. UART initialization
2. Sending a greeting message
3. Echo loop: reading a string, echoing it back with a prefix

## 2.3 Using PuTTY for Serial Communication

To interact with the TM4C123 via UART0, use a terminal emulator like PuTTY:

1. Download and install PuTTY from https://www.putty.org/
2. Connect the TM4C123 LaunchPad to your PC via USB
3. Open Windows Device Manager and find the COM port under "Ports (COM & LPT)" (e.g., COM3)
4. Launch PuTTY and configure:
   - Connection type: Serial
   - Serial line: COMx (replace with your COM port)
   - Speed: 115200 (or your configured baud rate)
5. Click "Open" to establish connection
6. Type text and press Enter to send strings to the microcontroller
7. View echoed responses and debug output

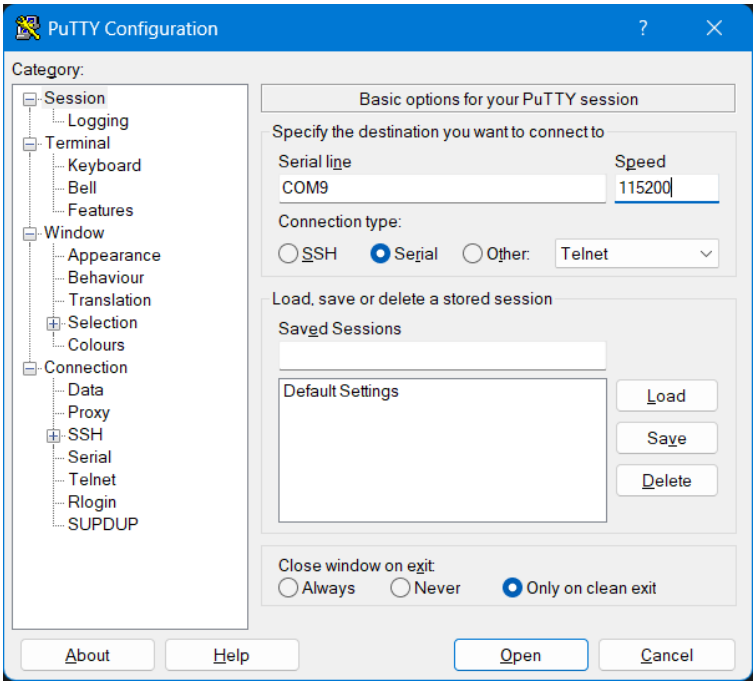


Figure 9.2: PuTTY Serial Configuration

**Tip**: Enable "Local echo" in PuTTY settings (Terminal → Line discipline options) to see your typed characters even before the microcontroller echoes them back.

## 2.4 Tasks

### 2.4.1 Task 1: LED Control via UART Commands

Write a program that receives single-character commands via UART to control the on-board LEDs.

**Requirements:**

- Initialize UART0 and configure GPIO Port F for LEDs (PF1/Red, PF2/Blue, PF3/Green)
- Receive single characters via UART0
- Implement command processing:
  - `'r'` or `'R'`: Toggle RED LED
  - `'b'` or `'B'`: Toggle BLUE LED
  - `'g'` or `'G'`: Toggle GREEN LED
  - `'a'` or `'A'`: Turn all LEDs ON
  - `'x'` or `'X'`: Turn all LEDs OFF
- Echo back confirmation messages (e.g., "RED LED toggled")
- Send error message for unrecognized commands

**Hint:**

```
char cmd = UART0_ReadChar();
switch (cmd) {
    case 'r':
    case 'R':
        GPIOF->DATA ^= (1 << 1);  // Toggle RED
        UART0_WriteString("RED LED toggled\r\n");
        break;
    // ... other cases
    default:
        UART0_WriteString("Unknown command\r\n");
}
```

### 2.4.2 Task 2: ADC Value Monitoring

Create a program that reads the on-board temperature sensor (or potentiometer) using ADC and continuously sends the values to a PC terminal.

**Requirements:**

- Initialize UART0 for serial communication
- Initialize ADC0 to read the internal temperature sensor (ADC0 sequencer 3)
- Use a timer (SysTick or GPTM) to trigger ADC sampling every 1 second
- Convert ADC value to temperature in degrees Celsius using the formula:

$$T_C = 147.5 - \frac{(\text{ADC\_Value} \times 3.3 \times 100)}{4096}$$

- Format and send the temperature reading via UART (e.g., "Temp: 25.3 C")
- Use `sprintf()` to format floating-point values into strings

**Hint:**

```
#include <stdio.h>

char buffer[50];
float temp = 147.5 - ((adcValue * 3.3 * 100) / 4096.0);
sprintf(buffer, "Temperature: %.1f C\r\n", temp);
```

```
UART0_WriteString(buffer);
```

### 2.4.3 Task 3: Simple Command-Line Interface

Implement a simple command-line interface (CLI) that accepts multi-character commands and arguments.

**Requirements:**

- Display a welcome message and prompt (">") after initialization
- Read complete strings (terminated by Enter) using `UART0_ReadString()`
- Parse commands with arguments (e.g., "LED RED ON", "DELAY 500")
- Implement command processing:
  - `LED <color> <ON|OFF>`: Control specific LED
  - `BLINK <color> <count>`: Blink LED specified number of times
  - `STATUS`: Report current LED states
  - `HELP`: Display available commands
- Send confirmation or error messages
- Display prompt after each command execution

**Hint:** Use `strcmp()` and `strtok()` for string parsing:

```
#include <string.h>

char buffer[50];
UART0_ReadString(buffer, 50);

char *cmd = strtok(buffer, " ");
if (strcmp(cmd, "LED") == 0) {
    char *color = strtok(NULL, " ");
    char *state = strtok(NULL, " ");
    // Process LED command
} else if (strcmp(cmd, "STATUS") == 0) {
    // Report status
}
UART0_WriteString("> ");  // Display prompt
```

## 2.5 Testing and Debugging

### 2.5.1 Common Issues and Solutions

**No output in terminal**

- **Cause**: Wrong COM port, incorrect baud rate, or UART not initialized
- **Solution**: Verify COM port in Device Manager, ensure baud rates match (115200), check UART initialization code

**Garbled characters**

- **Cause**: Baud rate mismatch between microcontroller and terminal
- **Solution**: Recalculate IBRD and FBRD values, verify system clock frequency, ensure terminal baud rate matches code

**Missing characters or corrupted data**

- **Cause**: FIFO overrun, insufficient processing speed, timing issues
- **Solution**: Enable FIFOs, process received data promptly, consider interrupt-driven reception for high data rates

**Characters not echoing back**

- **Cause**: RX not configured, GPIO alternate function not set correctly
- **Solution**: Verify GPIOA->AFSEL, GPIOA->PCTL, GPIOA->DEN settings, ensure RXE bit is set in UART0->CTL

### 2.5.2 Debugging Strategy

1. **Test transmission first**: Send fixed strings to verify TX is working before implementing RX
2. **Use LED indicators**: Toggle LEDs after UART initialization and during character transmission/reception
3. **Verify clock configuration**: Ensure system clock is 50 MHz as expected
4. **Check register values**: Use debugger to inspect UART registers (CTL, FR, IBRD, FBRD, LCRH)
5. **Test with simple echo**: Start with single-character echo before implementing complex string processing
6. **Monitor FIFO flags**: Check TXFF and RXFE flags to ensure proper FIFO operation

### 2.5.3 Advanced UART Features

For more sophisticated applications, consider exploring:

- **Interrupt-driven I/O**: Use UART interrupts instead of polling for better CPU efficiency
- **DMA transfers**: Offload large data transfers to DMA controller
- **Hardware flow control**: Implement RTS/CTS for reliable high-speed communication
- **9-bit mode**: Support address/data distinction in multi-processor systems
- **LIN and IrDA modes**: Special communication protocols for automotive and infrared applications