2.11 INTERNAL UART0 OF LPC1768

2.11.1 UART COMMUNICATION

Serial communication is a method of transmitting data one bit at a time over a communication channel. It is widely used in microcontroller systems and embedded applications, and the Universal Asynchronous Receiver/Transmitter (UART) is one of the most common hardware modules used to implement serial communication.

UART Overview:

- **UART** is a hardware module that converts parallel data from a microcontroller (or other device) into serial data and vice versa. ○ It works asynchronously, meaning that there is no clock signal shared between the transmitter and receiver.
- Data is transmitted bit by bit, with a start bit, data bits, optional parity bit, and stop bit.

Key Components: ○ **Transmitter (TX)**: Sends data serially to the receiver. ○ **Receiver (RX)**: Receives data serially from the transmitter.

- **Baud Rate**: This is the speed of communication, usually measured in bits per second (bps). Both sender and receiver must use the same baud rate for successful communication. ○ **Start Bit**: The first bit in a transmission that indicates the start of data. It is usually a logic low (0).
- **Data Bits**: These represent the actual data being transmitted. The number of data bits is typically 5, 6, 7, or 8 bits.
- **Parity Bit**: An optional bit used for error detection. It ensures that the number of bits set to 1 in the data is either even or odd. ○ **Stop Bits**: These are used to mark the end of a data frame. Typically, 1, 1.5, or 2 stop bits are used. A stop bit is a logic high (1).

Asynchronous Transmission:

- In asynchronous communication, the data is placed between start bit and stop bit.
- The start bit is always one bit whereas stop bit can be one or two bits. The start bit is always a 0 (low) and stop bit is 1 (high).

- When there is no transfer, the signal is 1 (high), which is referred as mark. The 0 (low) is referred to as space.
- Figure shows Asynchronous serial communication format.

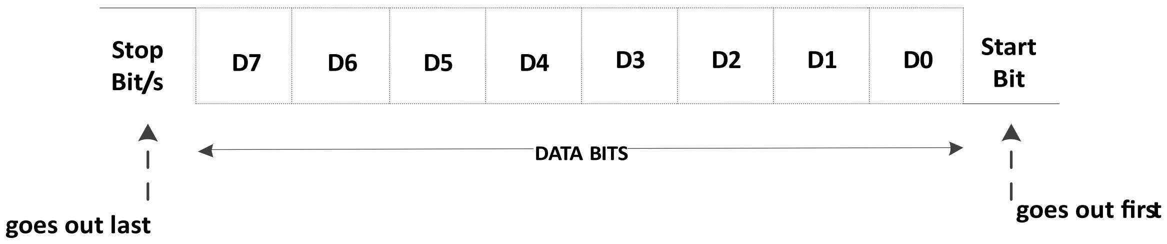


Figure: Asynchronous Serial Communication Format

- Figure shows the ASCII character 'A' framed between start and stop bits. ASCII value of character 'A' is 41h

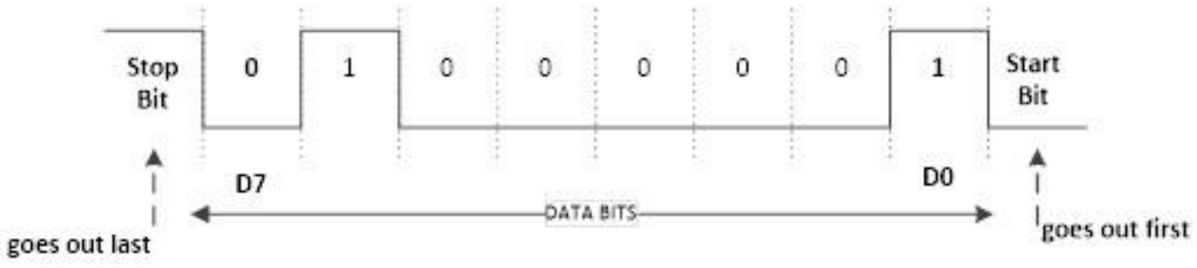


Figure: Framing ASCII 'A' (41H)

- Since UART uses asynchronous communication, there is no shared clock signal between the transmitter and receiver. Instead, both sides agree on the baud rate, and timing is based on this agreed rate.
- The start bit signals the beginning of a byte, and the stop bit(s) indicate the end of the byte.

Frame Structure: A typical UART frame consists of: ○

Start bit (1 bit) ○ **Data bits** (5 to 9 bits, depending on configuration) ○ **Parity bit** (optional, 1 bit) ○ **Stop bit(s)** (1 or 2 bits)

Baud Rate:

- Baud rate is the rate at which data is transmitted, e.g., 9600 bps, 115200 bps, etc. ○ Both the transmitting and receiving UART devices must agree on the baud rate for reliable communication.
- Bit rate is defined as the time required for transmitting one bit. Bit rate is given as

$$\boxed{\mathbf{bitrate} = \frac{\mathbf{1}}{\mathbf{f_{baud}}}}$$

Example: For 9600 bps, the bit rate = 1/9600 = 0.104166 msec

2.11.2 PIN SELECT REGISTER 0 (PINSEL0)

Before using UART0, you need to **configure the pins** using the **PINSEL** registers.

Function	Pin (LQFP100)	Alternate Function
TXD0 (Transmit)	**P0.2**	UART0 TX (Function 1)
RXD0 (Receive)	**P0.3**	UART0 RX (Function 1)

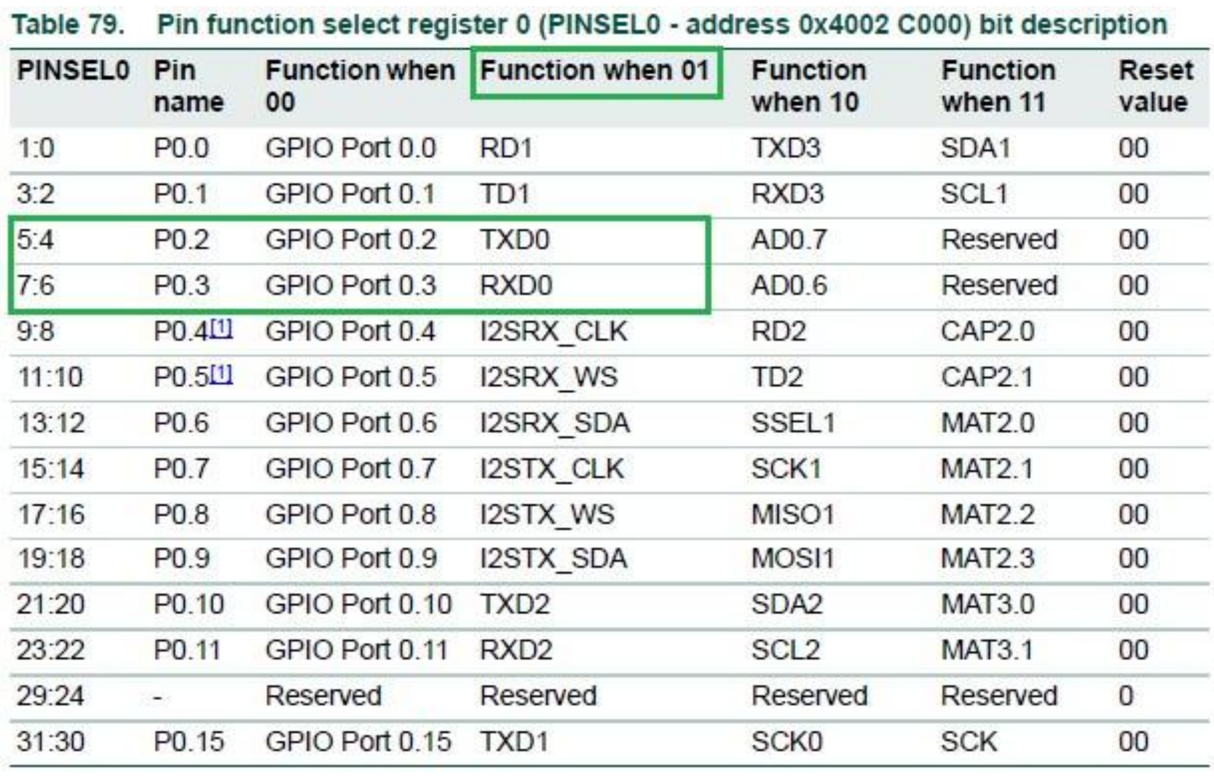


Table 79. Pin function select register 0 (PINSEL0 - address 0x4002 C000) bit description

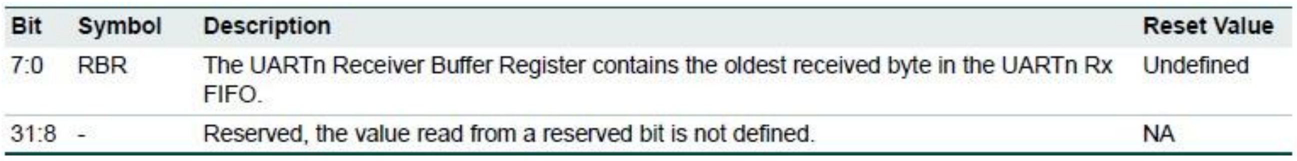
PINSEL0	Pin name	Function when 00	Function when 01	Function when 10	Function when 11	Reset value
1:0	P0.0	GPIO Port 0.0	RD1	TXD3	SDA1	00
3:2	P0.1	GPIO Port 0.1	TD1	RXD3	SCL1	00
5:4	P0.2	GPIO Port 0.2	TXD0	AD0.7	Reserved	00
7:6	P0.3	GPIO Port 0.3	RXD0	AD0.6	Reserved	00
9:8	P0.4[1]	GPIO Port 0.4	I2SRX_CLK	RD2	CAP2.0	00
11:10	P0.5[1]	GPIO Port 0.5	I2SRX_WS	TD2	CAP2.1	00
13:12	P0.6	GPIO Port 0.6	I2SRX_SDA	SSEL1	MAT2.0	00
15:14	P0.7	GPIO Port 0.7	I2STX_CLK	SCK1	MAT2.1	00
17:16	P0.8	GPIO Port 0.8	I2STX_WS	MISO1	MAT2.2	00
19:18	P0.9	GPIO Port 0.9	I2STX_SDA	MOSI1	MAT2.3	00
21:20	P0.10	GPIO Port 0.10	TXD2	SDA2	MAT3.0	00
23:22	P0.11	GPIO Port 0.11	RXD2	SCL2	MAT3.1	00
29:24	-	Reserved	Reserved	Reserved	Reserved	0
31:30	P0.15	GPIO Port 0.15	TXD1	SCK0	SCK	00

By default the pins are configured as GPIO pins. Configure for TXD0 and RXD0

```
LPC_PINCON->PINSEL0 |= (1 << 4) | (1 << 6);
```

2.11.3 UARTN RECEIVER BUFFER REGISTER

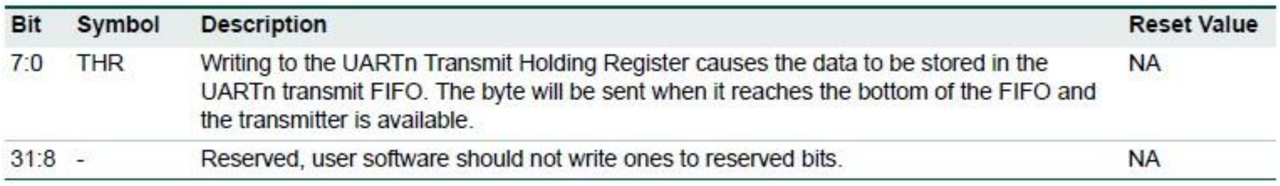
The **Receiver Buffer Register (RBR)** in the **LPC1768** stores the data received through **UART0**. When a byte is received via UART, it is stored in the **RBR** register, and it must be read before receiving more data.



Bit	Symbol	Description	Reset Value
7:0	RBR	The UARTn Receiver Buffer Register contains the oldest received byte in the UARTn Rx FIFO.	Undefined
31:8	-	Reserved, the value read from a reserved bit is not defined.	NA

2.11.4 UARTN TRANSMIT HOLDING REGISTER

The **Transmit Holding Register (THR)** in the LPC1768 is used to send data through a UART interface. When a byte is written to **THR**, it is moved to the **Transmit Shift Register (TSR)**, and then it is transmitted through the **TXD (Transmit Data) pin**.

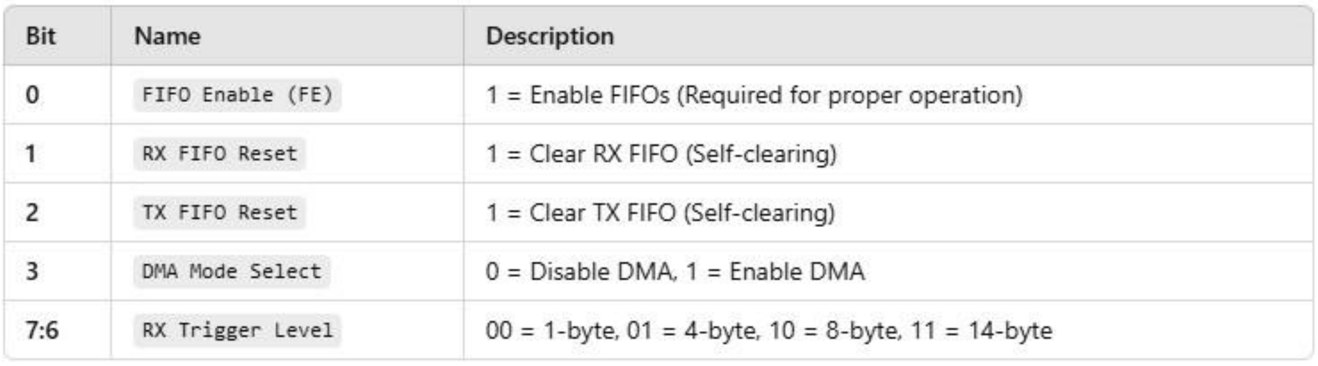


Bit	Symbol	Description	Reset Value
7:0	THR	Writing to the UARTn Transmit Holding Register causes the data to be stored in the UARTn transmit FIFO. The byte will be sent when it reaches the bottom of the FIFO and the transmitter is available.	NA
31:8	-	Reserved, user software should not write ones to reserved bits.	NA

2.11.5 UARTN FIFO CONTROL REGISTER (FCR)

The FIFO Control Register (FCR) in the LPC1768 configures the FIFO (First In, First Out) buffers for UART communication. The FIFO helps in buffering received and transmitted data, reducing CPU overhead by handling multiple bytes at once.

Bit Fields of FCR Register

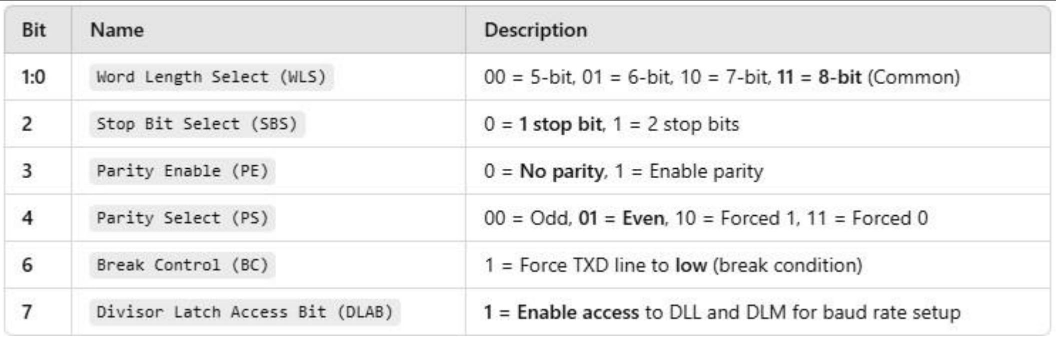


Bit	Name	Description
0	`FIFO Enable (FE)`	1 = Enable FIFOs (Required for proper operation)
1	`RX FIFO Reset`	1 = Clear RX FIFO (Self-clearing)
2	`TX FIFO Reset`	1 = Clear TX FIFO (Self-clearing)
3	`DMA Mode Select`	0 = Disable DMA, 1 = Enable DMA
7:6	`RX Trigger Level`	00 = 1-byte, 01 = 4-byte, 10 = 8-byte, 11 = 14-byte

```
LPC_UART0->FCR = (1 << 0) | (1 << 1) | (1 << 2);
```

2.11.6 UART LINE CONTROL REGISTER (LCR)

The **Line Control Register (LCR)** in the **LPC1768** configures the **data format** for UART communication, including data length, stop bits, parity, and access to baud rate settings.



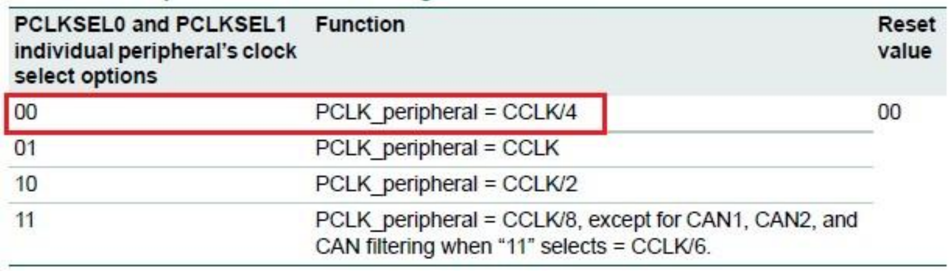
Bit	Name	Description
1:0	`Word Length Select (WLS)`	00 = 5-bit, 01 = 6-bit, 10 = 7-bit, **11** = **8-bit** (Common)
2	`Stop Bit Select (SBS)`	0 = **1 stop bit**, 1 = 2 stop bits
3	`Parity Enable (PE)`	0 = **No parity**, 1 = Enable parity
4	`Parity Select (PS)`	00 = Odd, **01** = **Even**, 10 = Forced 1, 11 = Forced 0
6	`Break Control (BC)`	1 = Force TXD line to **low** (break condition)
7	`Divisor Latch Access Bit (DLAB)`	**1** = **Enable access** to DLL and DLM for baud rate setup

```
// 8-bit character length, 1 stop bit, no parity, enable DLAB
   LPC_UART0->LCR = (1 << 0) | (1 << 1) | (1 << 7);
```

2.11.7 PCLK SELECTION FOR UART0

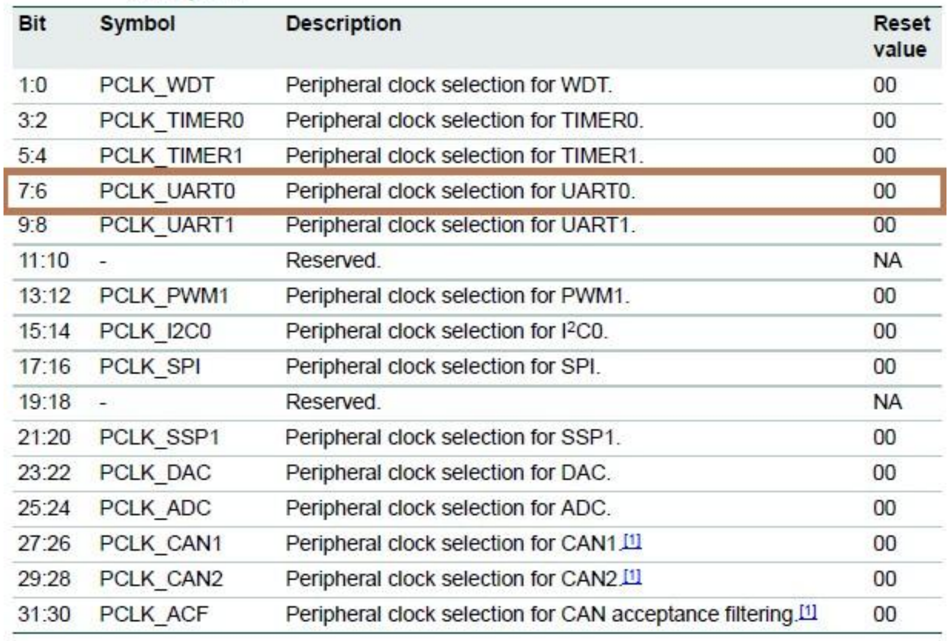
The Peripheral Clock (PCLK) for UART0 in the LPC1768 is controlled by the Peripheral Clock Selection Register 0 (PCLKSEL0). The PCLK determines the input clock frequency for the UART module, which is then used to generate the baud rate.

Table 42. Peripheral Clock Selection register bit values



PCLKSEL0 and PCLKSEL1 individual peripheral's clock select options	Function	Reset value
00	PCLK_peripheral = CCLK/4	00
01	PCLK_peripheral = CCLK	
10	PCLK_peripheral = CCLK/2	
11	PCLK_peripheral = CCLK/8, except for CAN1, CAN2, and CAN filtering when "11" selects = CCLK/6.	

Table 40. Peripheral Clock Selection register 0 (PCLKSEL0 - address 0x400F C1A8) bit description

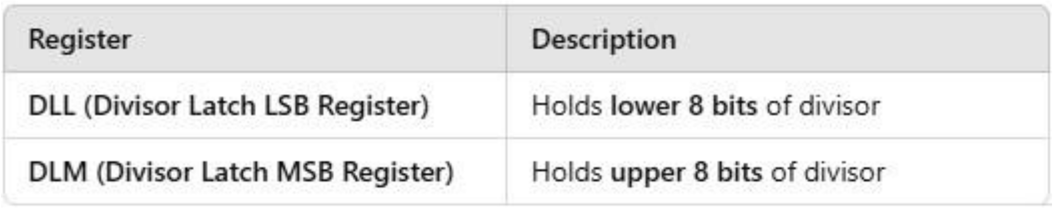


Bit	Symbol	Description	Reset value
1:0	PCLK_WDT	Peripheral clock selection for WDT.	00
3:2	PCLK_TIMER0	Peripheral clock selection for TIMER0.	00
5:4	PCLK_TIMER1	Peripheral clock selection for TIMER1.	00
7:6	PCLK_UART0	Peripheral clock selection for UART0.	00
9:8	PCLK_UART1	Peripheral clock selection for UART1.	00
11:10	-	Reserved.	NA
13:12	PCLK_PWM1	Peripheral clock selection for PWM1.	00
15:14	PCLK_I2C0	Peripheral clock selection for I^{2}C0.	00
17:16	PCLK_SPI	Peripheral clock selection for SPI.	00
19:18	-	Reserved.	NA
21:20	PCLK_SSP1	Peripheral clock selection for SSP1.	00
23:22	PCLK_DAC	Peripheral clock selection for DAC.	00
25:24	PCLK_ADC	Peripheral clock selection for ADC.	00
27:26	PCLK_CAN1	Peripheral clock selection for CAN1.[1]	00
29:28	PCLK_CAN2	Peripheral clock selection for CAN2.[1]	00
31:30	PCLK_ACF	Peripheral clock selection for CAN acceptance filtering.[1]	00

```
// Ensure correct PCLK selection (PCLK = CCLK/4 by default)
LPC_SC->PCLKSEL0 &= ~((1 << 6) | (1 << 7));
```

2.11.8 DLL AND DLM REGISTERS OF UART0 IN LPC1768

The Divisor Latch Registers (DLL and DLM) are used to configure the baud rate for UART0 in the LPC1768. These registers store the divisor value that determines the baud rate.



Register	Description
DLL (Divisor Latch LSB Register)	Holds lower 8 bits of divisor
DLM (Divisor Latch MSB Register)	Holds upper 8 bits of divisor

Divisor Latch Access Bit (DLAB)

- To access **DLL and DLM**, you must set **DLAB = 1** in the **LCR (bit 7)**.
- After setting the baud rate, **clear DLAB (set it back to 0)**.

Baud Rate Calculation

$$\text{Baud Rate} = \frac{\text{PCLK}}{16 \text{ x } (\text{DLL} + (\text{DLM x } 256))}$$

System Core Clock CCLK = 100 MHZ PCLK = CCLK / 4 = 25 MHZ

For baud rate of 9600 bps,

$$9600 = \frac{25 \text{ x } 10_{6}}{16 \text{ x } (\text{DLL} + (\text{DLM x } 256))}$$

DLM = 0 DLL = 162 (0xA2)

```
LPC_UART0->DLL = 0xA2;  // Lower byte of divisor LPC_UART0-
>DLM = 0x00;  // Upper byte of divisor
```

2.11.9 EXAMPLE CODE 01 – SQUARE WAVE GENERATION

Write a C program that interfaces with the LPC17xx microcontroller's UART0 to receive character over serial communication. The program should continuously listen for incoming characters, and when it receives the character 'T', it should respond by sending the string "VVCE MYSURU" over UART.

```
main.c
```

```
#include <LPC17xx.h>
#include "extern.h"

int main(void)
{
    SystemInit();

    UARTInitialize();

    while (1)
    {
        char receivedChar;

         receivedChar = UARTReceiveCharacter();

         if (receivedChar == 'T')
        {
            UARTSendString("VVCE MYSURU\r\n");
        }
    }
}
```

uart.c

```
void UARTInitialize(void)
{
    // Configure P0.2 as TXD0 and P0.3 as RXD0
    LPC_PINCON->PINSEL0 &= ~((3 << 4) | (3 << 6)); // Clear bits before setting
LPC_PINCON->PINSEL0 |= (1 << 4) | (1 << 6);   // Set bits for TXD0 and RXD0
    // Enable FIFO, reset RX & TX FIFO
    LPC_UART0->FCR = (1 << 0) | (1 << 1) | (1 << 2);
    // 8-bit character length, 1 stop bit, no parity, enable
DLAB     LPC_UART0->LCR = (1 << 0) | (1 << 1) | (1 << 7);
    // Ensure correct PCLK selection (PCLK = CCLK/4 by default)
    LPC_SC->PCLKSEL0 &= ~((1 << 6) | (1 << 7));

    // Calculate divisor for baud rate 9600 bps
    // Assuming CCLK = 100MHz, PCLK = 25MHz)
    // Divisor = PCLK / (16 * Baudrate) = 25M / (16 * 9600) = 162 = 0xA2
    LPC_UART0->DLL = 0xA2;  // Lower byte of divisor     LPC_UART0-
>DLM = 0x00;  // Upper byte of divisor

    LPC_UART0->LCR &= ~(1 << 7);  // Clear DLAB }
```

```
 void UARTSendCharacter(char
ch)
{     while ((LPC_UART0->LSR & (1 << 5)) == 0);  // Wait until THR is empty
    LPC_UART0->THR = ch;
}  void UARTSendString(char
*str)
{     if (str == '\0') return;  // Null check for
safety

    while (*str)
    {
        UARTSendCharacter(*str++);
    }
}
 char
UARTReceiveCharacter(void)
{
    // Wait until RBR contains valid data
while ((LPC_UART0->LSR & (1 << 0)) == 0);
return LPC_UART0->RBR; }
```

extern.c

```
void UARTInitialize(void); void
UARTSendCharacter(char); void
UARTSendString(char*); char
UARTReceiveCharacter(void);
```

extern.h

```
#ifndef __EXTERN_H #define
__EXTERN_H


extern void UARTInitialize(void); extern
void UARTSendCharacter(char); extern
void UARTSendString(char*); extern char
UARTReceiveCharacter(void);


#endif
```

Further exploration:

Configure for different baud rates.

Modify the code to transmit error messages if an invalid character is received.

Observe the ASCII value of characters in the DSO.

Can you rewrite using interrupts?