**Introduction to UART (PL011)**

The **UART** (Universal Asynchronous Receiver/Transmitter) is a **communication peripheral** used in **ARM-based System-on-Chip (SoC)** designs. It is developed and licensed by **ARM** and follows the **AMBA (Advanced Microcontroller Bus Architecture)** specification, making it easy to integrate into ARM-based systems.

This UART connects to the **APB (Advanced Peripheral Bus)**, which is part of the AMBA system. It also includes support for **IrDA SIR (Infrared Data Association Serial Infrared)** protocol – allowing infrared data communication.

**✅ Key Features of the UART**

* **AMBA-compliant**: Works well with other components in the AMBA-based system (Rev 2.0 and above).
* **Supports UART and IrDA SIR modes**: Can send and receive data either through normal serial wires or through infrared.
* **FIFO Buffers**:
  + **32-byte transmit buffer (32x8)**
  + **32-byte receive buffer (32x12)**
  + These reduce the number of times the CPU is interrupted.
  + FIFO can also be disabled for single-byte data.
* **Programmable Baud Rate Generator**:
  + Allows setting the speed of data transmission.
  + Divides the clock with fine granularity to support various data rates.
  + Supports fractional values, so any clock (e.g., 3.6864 MHz) can be used.
* **Standard Serial Communication Bits**:
  + Adds start bit, stop bit, and parity bit automatically during data transmission and removes them during reception.
* **Interrupt Management**:
  + You can individually enable/disable interrupts for different events like:
    - Transmit FIFO
    - Receive FIFO
    - Receive timeout
    - Modem status
    - Error conditions
* **DMA (Direct Memory Access)** support for efficient data transfer.
* **Special Features**:
  + Detects false start bits (incorrect data starts)
  + Can generate and detect line breaks (used for special control)
  + Supports **hardware modem control**: CTS, RTS, DSR, DCD, DTR, and RI.
  + **Hardware flow control** can be programmed (automatically manages data flow to prevent overflow)

**🔧 Programmable Settings**

You can configure:

* **Baud rate**: Choose how fast data is transmitted.
* **Number of data bits**: 5, 6, 7, or 8 bits
* **Number of stop bits**: 1 or 2
* **Parity mode**: Even, odd, stick (fixed), or none
* **FIFO depth**: Use full 32-depth or disable it for 1-byte operation
* **Trigger levels** for FIFO: Choose when an interrupt is generated (like at 4 bytes full)
* **Clock frequency settings**: Supports low-power IrDA mode
* **Hardware flow control**: Optional control using modem signals

**🔄 Difference from Older UART (16C650)**

This UART is **not backward-compatible** with the older PL010 or 16C650 UARTs. Here are the differences:

* Different **FIFO trigger levels**
* Different **register map** and bit functions
* **No support for 1.5 stop bits** (only 1 or 2 are allowed)
* **No support for independent receive clock**
* **Modem delta status** signals are not available

**Summary**

The ARM PL011 UART is a powerful, programmable serial communication module that supports both traditional UART and infrared data transfer. It offers flexible settings, FIFO buffers, interrupt controls, and DMA support, making it suitable for a wide range of embedded applications. It's also more advanced than older UART designs like the 16C650, though not backward compatible.

## ****UART Product Revisions (Version Changes)****

The UART module has gone through several **revisions** (updates), and each revision may include **functional changes** or **improvements**. These changes are reflected in a special register called the UARTPeriphID2 register, specifically in **bits [7:4]**, which identify the **revision number**.

### 📋 Summary of Revisions:

| **Revision Range** | **What's Changed?** |
| --- | --- |
| **r1p0 → r1p1** | - The UARTPeriphID2 register bits [7:4] now return **0x1** |
| **r1p1 → r1p3** | - The UARTPeriphID2 register bits [7:4] now return **0x2** |
| **r1p3 → r1p4** | - No change in the revision field; still **0x2** |
| **r1p4 → r1p5** | - **Transmit and receive FIFOs** are increased to **32-depth** (more buffer size) - UARTPeriphID2 register bits [7:4] now return **0x3** |

### 📌 Note:

For more technical details or bugs in any version, you should refer to the **engineering errata** provided along with the product documentation.

## 📘 ****Chapter 2: Functional Overview of UART****

This chapter explains the **main working blocks** and **capabilities** of the UART (Universal Asynchronous Receiver/Transmitter) peripheral in an ARM-based SoC.

### 📑 **Sections in Chapter 2**

1. **Overview**
2. **Functional Description**
3. **IrDA SIR ENDEC Functional Description**
4. **UART Operation**
5. **UART Modem Operation**
6. **UART Hardware Flow Control**
7. **UART DMA Interface**
8. **Interrupts**

## 🔍 ****Overview****

### ✅ **Main Functions**

The UART performs:

* **Serial-to-parallel conversion**: Converts data coming from outside (peripheral) into a format the CPU can process.
* **Parallel-to-serial conversion**: Converts data from the CPU into serial format to send out.

### ✅ **Data Access**

* CPU **reads/writes data and control/status** via the **APB (Advanced Peripheral Bus)**.
* Data paths (transmit and receive) use **independent FIFO buffers**:
  + **32-byte Transmit FIFO**
  + **32-byte Receive FIFO**

These buffers help reduce the number of CPU accesses and avoid data loss.

### ⏱️ **Baud Rate and Clock**

* The UART includes a **programmable baud rate generator**.
* It derives a **shared clock** for transmitting and receiving from the **UARTCLK** reference input.
* It supports similar functions to the **industry-standard 16C650 UART**.

#### 📈 **Maximum Supported Baud Rates**:

| **Mode** | **Max Baud Rate** |
| --- | --- |
| UART mode | 921,600 bps |
| IrDA mode | 460,800 bps |
| Low-power IrDA mode | 115,200 bps |

**Registers to configure baud rate and format:**

* UARTLCR\_H – Line control (data length, stop bits, parity)
* UARTIBRD – Integer baud rate divisor
* UARTFBRD – Fractional baud rate divisor

## 🔔 ****Interrupts****

UART can generate **individually maskable interrupts** for:

* Transmit
* Receive (including timeout)
* Modem status
* Error conditions

You can also enable a **single combined interrupt**: asserted if **any** unmasked condition occurs.

## 📦 ****DMA Support****

UART can raise **DMA request signals** to allow direct data transfer between memory and UART — bypassing the CPU — for efficient high-speed communication.

## ⚠️ ****Error Handling****

* **Framing, parity, or break errors**:
  + Corresponding **error bits are set**
  + **Data is still stored** in the FIFO with error status
* **Overrun condition**:
  + Happens when FIFO is full and new data arrives
  + **Overrun bit is set**
  + **New data is discarded** (FIFO content is protected)

You can also configure FIFO to **1-byte depth**, making it behave like a **simple double-buffered UART**.

## 📞 ****Modem Control Signals****

UART supports standard modem signals:

* **Input signals**:
  + CTS (Clear To Send)
  + DCD (Data Carrier Detect)
  + DSR (Data Set Ready)
  + RI (Ring Indicator)
* **Output signals**:
  + RTS (Request To Send)
  + DTR (Data Terminal Ready)

## 🔄 ****Hardware Flow Control****

* Uses **nUARTCTS (input)** and **nUARTRTS (output)** signals
* Automatically manages **flow of data** based on line status
* Prevents overflow or data loss during high-speed transfers

## 🌐 ****IrDA SIR Block (Infrared Communication)****

* UART includes an **IrDA SIR protocol ENDEC** (encoder/decoder)
* Instead of standard serial pins (UARTTXD, UARTRXD), it uses:
  + SIROUT (output)
  + SIRIN (input)
* Works with an **infrared transceiver**

### ⚙️ Behavior When IrDA is Enabled:

* UARTTXD line stays HIGH (inactive)
* UARTRXD and modem status signals are **ignored**
* IrDA works in **half-duplex mode**:
  + Can either **transmit or receive**, but **not both at the same time**
* IrDA requires a **minimum 10ms delay** between transmission and reception.

## ✅ Summary

The UART module offers:

* Reliable data transfer using **FIFOs and baud rate generator**
* Configurable **data format and flow control**
* Support for **interrupts and DMA**
* Modem interface signals for serial communication standards
* **Infrared (IrDA) communication** option

## ****2.2 UART Functional Description****

**🔹 Top-Level View**

The UART block can be divided into **5 functional parts**:

1. **APB Interface & Register Block**
2. **Baud Rate Generator**
3. **Transmit Path**
4. **Receive Path**
5. **FIFO & Interrupt/DMA Handling**

**1️⃣ APB Interface and Register Block**

* Signals: PCLK, PSEL, PENABLE, PWRITE, PADDR, PWDATA, PRDATA
* This is how the **processor (CPU / APB bus master)** configures and communicates with UART.
* Functions:
  + Configure **baud rate divisor**, control registers, interrupt enable, FIFO control, etc.
  + Write to **TX FIFO** (data to send).
  + Read from **RX FIFO** (received data).
  + Expose **status flags** (empty/full/error).

**2️⃣ Baud Rate Generator**

* Takes **UARTCLK** (reference clock).
* Uses **baud rate divisor** to generate a Baud16 clock → 16× oversampling for accurate RX sampling.
* Ensures correct **transmit timing** and **receive sampling**.

**3️⃣ Transmit Path**

* **32×8 Transmit FIFO**
  + Stores outgoing characters (up to 32 bytes).
  + CPU or DMA writes into it via APB.
* **Transmitter**
  + Fetches data from FIFO.
  + Adds **start bit, data bits, parity (if enabled), stop bit(s)** → forming the UART frame.
  + Sends serialized data via UARTTXD.
  + If **IrDA SIR mode** → passes through SIR encoder → nSIROUT.

**4️⃣ Receive Path**

* **UART input pin:** UARTRXD (or SIRIN if using IrDA).
* **Receiver**
  + Samples input at 16× baud rate (Baud16).
  + Detects **start bit**, shifts in data bits, checks parity & stop bits.
  + Pushes valid data into **32×12 Receive FIFO** (stores data + error flags).
* CPU/DMA reads from RX FIFO via APB.

**5️⃣ FIFO Status & Interrupt Generation**

* Monitors:
  + TX FIFO status (empty, almost full, etc.).
  + RX FIFO status (full, threshold reached, etc.).
  + Error conditions (parity, framing, overrun).
* Generates interrupts:
  + UARTTXINTR, UARTRXINTR, UARTMSINTR, UARTRTINTR, etc.
* Works with **DMA interface** for high-speed data transfer:
  + UARTRXDMASREQ / UARTTXDMASREQ for DMA requests.
  + UARTRXDMAACK / UARTTXDMAACK for acknowledgements.

**🔹 Reset Signals**

* PRESETn (global reset) + nUARTRST (block reset).
* Reset clears FIFOs and control registers.

**🔹 Data Flow Summary**

1. **CPU writes a byte → APB bus → TX FIFO → Transmitter → UARTTXD pin.**
2. **Remote data arrives on RXD → Receiver → RX FIFO → CPU/DMA reads via APB.**
3. **Baud rate generator** ensures correct timing.
4. **FIFO + Interrupt/DMA** make data movement efficient.

✅ In short:

* The **APB interface** is the control brain.
* The **baud rate generator** sets timing.
* **TX FIFO → Transmitter → UARTTXD** handles sending.
* **RXD → Receiver → RX FIFO** handles receiving.
* **FIFO status + Interrupt/DMA** provide efficient, error-checked communication.

.

### 🧱 **Main Blocks Described in This Section**

Each sub-block plays a distinct role in the UART’s operation. Here’s a concise explanation:

### **2.2.1 AMBA APB Interface**

* Interfaces UART with the **CPU via APB bus**.
* Handles **read/write decoding** for:
  + **Status/control registers**
  + **Transmit FIFO**
  + **Receive FIFO**

### **2.2.2 Register Block**

* Stores:
  + Data **written** via APB
  + Data **read** by the CPU
* Acts as the access point for **UART configuration and control**.

### **2.2.3 Baud Rate Generator**

* Generates internal **timing signals** for:
  + **UART mode**: Baud16 (16× baud rate)
  + **Low-power IrDA mode**: IrLPBaud16
* These are derived from the **UARTCLK (reference clock)**.
* Ensures proper timing for:
  + **Transmit and receive logic**
  + **IrDA pulse width generation**

### **2.2.4 Transmit FIFO**

* **8-bit wide**, **32-entry** buffer
* Temporarily holds data written by CPU before sending
* Can be **disabled** to behave like a **1-byte register**

### **2.2.5 Receive FIFO**

* **12-bit wide**, **32-entry** buffer
* Holds received data **plus error status bits**
* Can also be **disabled** to act like a **1-byte register**

### **2.2.6 Transmit Logic**

* Reads from the **Transmit FIFO**
* Converts **parallel data to serial**
* Sends frame:
  + Start bit
  + Data bits (LSB first)
  + Parity bit (if configured)
  + Stop bits

### **2.2.7 Receive Logic**

* Detects **start bit**
* Converts **serial to parallel**
* Performs **error detection**:
  + Parity error
  + Framing error
  + Overrun
  + Break condition
* Writes data and error info to **Receive FIFO**

### **2.2.8 Interrupt Generation Logic**

* Generates **individual maskable interrupts** for:
  + Transmit
  + Receive
  + Modem status
  + Error conditions
* Also generates a **combined interrupt output** (logical OR of all interrupt sources)
* Two usage modes:
  + **Modular drivers**: Use **combined interrupt**
  + **Global interrupt controller**: Use **individual interrupt lines**

### **2.2.9 DMA Interface**

* UART can raise request data transfers through a **Direct Memory Access (DMA)** controller.
* Reduces CPU load during large or fast data transfers.
* See **Section 2.19** for details.

### **2.2.10 Synchronization Registers and Logic**

* The uart supports both asynchronous and synchronous operation clock **pclk,uartclk**.
* Synchronization registers and handshaking logic have been implemented and are **active at all times**.
* Synchronization of control sigal is performed on both directions of data flow.that is **pclk to uartclk** domain and **uartclk to pclk** domain.

### **2.2.11 Test Registers and Logic**

* Used only for **integration and functional testing**
* Provides access to **all UART inputs/outputs** for verification
* **Do NOT use** during normal operation

## 📊 ****Block Diagram (Figure 2-1 Summary)****

The UART block diagram includes:

* **FIFO Buffers**: 32×8 Transmit and 32×12 Receive
* **Transmit and Receive logic**
* **Baud Rate Generator**
* **APB Interface & Control Logic**
* **DMA Interface**
* **Interrupt Generator**
* **Modem I/O (CTS, RTS, DSR, etc.)**
* **IrDA Interface (SIRIN, SIROUT)**

📌 Inputs like PCLK, PWRITE, PADDR, etc., are from the **APB interface**, while outputs like UARTTXD, UARTRXD are connected to serial lines or transceivers.

## 🔦 ****2.3 IrDA SIR ENDEC Functional Description****

The **IrDA SIR ENDEC (Encoder/Decoder)** block enables **infrared (IrDA)** communication using **Serial Infrared (SIR)** format.

It consists of:

* ✅ **SIR Transmit Encoder**
* ✅ **SIR Receive Decoder**

These connect to the **PrimeCell UART core** and interface with infrared transceivers.

**🔹 What is IrDA SIR ENDEC?**

* **IrDA SIR = Infrared Data Association, Serial Infrared**
* ENDEC = **Encoder + Decoder** block
* Purpose: Convert between **standard UART signals (electrical)** and **infrared optical pulses** used in IrDA communication.

So this block allows the same UART core to support **both normal UART mode and IrDA infrared mode**.

**🔹 Key Signals**

* **TXD** → Normal UART transmit signal (from UART core).
* **UARTTXD** → Final electrical TX pin.
* **nSIROUT** → Infrared (optical) TX output (active low).
* **RXD** → Normal UART receive input (to UART core).
* **UARTRXD** → Electrical RX pin.
* **SIRIN** → Infrared (optical) RX input.
* **SIREN** → Control signal (enables IrDA mode).

**🔹 Functional Blocks**

1. **PrimeCell UART core**
   * The normal UART transmitter and receiver logic (data framing, start/stop bits, FIFO handling).
   * Provides TXD (to be sent) and consumes RXD (received).
2. **SIR Transmit Encoder**
   * Takes TXD from the UART core.
   * Converts logic level UART bits into **short pulses** suitable for IrDA.
     + In SIR, a logical "0" is sent as an infrared pulse, while a logical "1" is idle (no light).
   * Output → nSIROUT (goes to infrared diode driver).
3. **SIR Receive Decoder**
   * Takes input from SIRIN (infrared photodiode).
   * Converts received light pulses back into standard UART logic bits.
   * Output → RXD → UART core.
4. **Multiplexer (RX Path)**
   * Selects between normal UART input (UARTRXD) and IrDA input (SIRIN → decoder → RXD).
   * Controlled by **SIREN** (IrDA enable).
5. **OR Gate (TX Path)**
   * Combines standard UART TXD with IrDA output selection.
   * If IrDA disabled, UARTTXD = TXD (normal UART).
   * If IrDA enabled, UARTTXD = encoded output from SIR encoder.

**🔹 Modes of Operation**

* **Normal UART mode (SIREN = 0)**
  + TX path: TXD → UARTTXD
  + RX path: UARTRXD → RXD
  + IrDA encoder/decoder bypassed.
* **IrDA SIR mode (SIREN = 1)**
  + TX path: TXD → SIR Encoder → nSIROUT (infrared output)
  + RX path: SIRIN → SIR Decoder → RXD
  + UART core still sees normal UART-style frames, but transmitted/received via infrared pulses.

**✅ Summary**

The **IrDA SIR ENDEC** is a **bridge between UART and Infrared communication**:

* **SIR Transmit Encoder**: Converts UART TX bits into IR pulses.
* **SIR Receive Decoder**: Converts IR pulses back into UART RX bits.
* **SIREN control**: Chooses between **normal UART mode** and **IrDA mode**.

# 🔹 **2.3.1 IrDA SIR Transmit Encoder**

This block takes the **UART TX bitstream (NRZ format)** and converts it into the **IrDA SIR pulse format**.

### **Modulation Scheme**

* UART output is **NRZ (Non Return to Zero)**:
  + "1" → line stays HIGH
  + "0" → line stays LOW
* IrDA requires **RZI (Return to Zero, Inverted)**:
  + Logic **0** → transmitted as a **short infrared light pulse**
  + Logic **1** → no pulse (idle / marking state)

So, the encoder **inverts + pulses** the UART data into IR-compatible format.

### **Pulse Width**

* **Normal IrDA mode**:
  + Pulse width = **3/16 of a bit period**
  + Implemented using the **×16 internal clock (Baud16)**
  + Example: At 9600 baud, bit period = 104.2 µs → pulse = ~19.5 µs
* **Low-Power IrDA mode**:
  + Fixed pulse width = **3/16 of a 115200 bps bit period (~1/38400 s ≈ 26 µs)**
  + Independent of actual baud rate.
  + Implemented using **IrLPBaud16 = 1.8432 MHz** (derived from UARTCLK and divisor in **UARTILPR register**).

### **Polarity**

* Encoder output is normally **LOW** (marking state = no light).
* To represent a **logic 0**, encoder outputs a **HIGH pulse**, which triggers the IR LED to emit a light pulse.

### **Jitter**

* When fractional baud rate divisors are used, **Baud16 edges are irregular**, causing pulse timing variation (**jitter**).
* Worst case = **3 UARTCLK cycles**.
* Still safe within IrDA spec if:
  + UARTCLK > 3.6864 MHz
  + Baud rate ≤ 115200 bps
  + Jitter < 9% (IrDA allows up to 13%).

# 🔹 **2.3.2 IrDA SIR Receive Decoder**

This block takes the **infrared pulses from the photodiode (SIRIN)** and converts them back into a **UART-compatible NRZ bitstream**.

### **Operation**

* Input is normally **HIGH** (marking state = idle, no light).
* A **LOW pulse** = start of a data bit → indicates **logic 0**.
* Demodulates the incoming **RZI pulses** back into **NRZ bits** for the UART core.

### **Glitch Filtering**

To avoid false triggering from noise or short IR spikes:

* In **normal IrDA mode**: ignores any SIRIN pulse **shorter than 3/16 of Baud16**.
* In **low-power IrDA mode**: ignores pulses **shorter than 3/16 of IrLPBaud16**.

This ensures only valid IR pulses are treated as data bits.

# ✅ **Summary**

* **Transmit Encoder:**
  + Converts UART NRZ to IrDA RZI pulses.
  + Logic 0 = IR pulse (width = 3/16 bit).
  + Supports both **normal mode** (baud-dependent pulse) and **low-power mode** (fixed width).
* **Receive Decoder:**
  + Converts IR pulses back to NRZ bitstream.
  + Ignores short glitches (< 3/16 bit).
  + Ensures reliable detection of start/data bits.

⚡ In short:

* **TX Encoder:** UART → IR pulses (0 = pulse, 1 = idle).
* **RX Decoder:** IR pulses → UART (filter out noise).

# 🔹 **2.4 Operation (Overview)**

This section explains how the UART + IrDA SIR ENDEC actually works.  
The operation covers:

1. **Interface reset** → how the UART is properly reset.
2. **Clock signals** → what clock frequencies are needed to generate accurate baud rates.  
   (Other subsections like UART operation, IrDA mode, character frame, etc. are described later in the manual.)

# **2.4.1 Interface Reset**

The UART and the IrDA SIR ENDEC (the infrared encoder/decoder) are reset by two signals:

* **PRESETn** = **global reset signal** (applies to whole chip/subsystem).
* **nUARTRST** = **block-specific reset signal** (only for UART).

### How reset must be applied:

* The **external reset controller** uses PRESETn to control nUARTRST.
* **Assertion (active reset)**: nUARTRST must be asserted **asynchronously** (immediately, not waiting for a clock edge).
* **Deassertion (release from reset)**: must happen **synchronously with UARTCLK** (to avoid glitches).

### Timing requirement:

* PRESETn must be held **LOW long enough** to reset the slowest block in the system.
* UART specifically requires PRESETn **LOW for at least 1 PCLK period**.

👉 After reset, UART registers will have default values (defined later in Chapter 3: Programmer’s Model).

# **2.4.2 Clock Signals**

The UART uses **UARTCLK** to generate baud rates. There are strict rules for setting this frequency.

### **Baud rate range requirement**

1. **Minimum UARTCLK** must be fast enough to generate the **maximum baud rate**:

FUARTCLK(min)≥16×baudmaxF\_{UARTCLK(min)} \geq 16 \times baud\_{max}FUARTCLK(min)​≥16×baudmax​

* + Because UART needs 16 clock ticks per bit for sampling.

1. **Maximum UARTCLK** must not exceed what’s needed for the **minimum baud rate**:

FUARTCLK(max)≤16×65535×baudminF\_{UARTCLK(max)} \leq 16 \times 65535 \times baud\_{min}FUARTCLK(max)​≤16×65535×baudmin​

* + Because the baud rate divisor register is 16-bit (max 65535).

**Example:**  
For baud range 110 bps → 460,800 bps:

* FUARTCLK(min)=16×460800=7.3728 MHzF\_{UARTCLK(min)} = 16 \times 460800 = 7.3728 \text{ MHz}FUARTCLK(min)​=16×460800=7.3728 MHz
* FUARTCLK(max)=16×65535×110=115.34 MHzF\_{UARTCLK(max)} = 16 \times 65535 \times 110 = 115.34 \text{ MHz}FUARTCLK(max)​=16×65535×110=115.34 MHz  
  So UARTCLK must lie between **7.37 MHz and 115.34 MHz**.

### **Clock accuracy**

UARTCLK frequency must be **precise enough** so baud rate errors stay within tolerance for communication.

### **PCLK vs UARTCLK constraint**

There’s a ratio limit between **PCLK (processor/system clock)** and **UARTCLK**:

FUARTCLK≤53×FPCLKF\_{UARTCLK} \leq \tfrac{5}{3} \times F\_{PCLK}FUARTCLK​≤35​×FPCLK​

👉 This ensures the processor has enough time to move received data into the FIFO before new data arrives.

**Example:**

* If UARTCLK = 14.7456 MHz, and baud = 921,600 bps,
* Then PCLK must be ≥ 8.85276 MHz.
* This guarantees FIFO writes happen in time.

# ✅ **Summary**

* **Reset**:
  + PRESETn (global reset) must be applied LOW long enough, then released in sync with UARTCLK.
  + nUARTRST is driven by PRESETn.
* **Clock signals**:
  + UARTCLK must be set within limits based on baud rate range.
  + Must be accurate enough for communication.
  + UARTCLK must not exceed **5/3 of PCLK**, to allow safe data transfers.

**🔹 UART Registers for Control & Baud Rate**

The UART is controlled by **three key registers**:

1. **UARTLCR\_H (Line Control Register):**  
   Defines transmission parameters:
   * Word length (5–8 bits per character)
   * Number of stop bits (1 or 2)
   * Parity (none, odd, even)
   * Break signal generation
   * FIFO enable/disable
2. **UARTIBRD (Integer Baud Rate Register):**  
   Holds the **integer part** of the baud rate divisor.
3. **UARTFBRD (Fractional Baud Rate Register):**  
   Holds the **fractional part** of the baud rate divisor.

**🔹 Baud Rate Divider**

* The UART clock **UARTCLK** is divided down to generate the baud rate.
* Formula:

Baud Divisor=UARTCLK16×Baud\text{Baud Divisor} = \frac{UARTCLK}{16 \times Baud}Baud Divisor=16×BaudUARTCLK​

* Example: If UARTCLK=16 MHzUARTCLK = 16\ \text{MHz}UARTCLK=16 MHz, baud = 115200:

Divisor=16,000,00016×115200=8.68Divisor = \frac{16,000,000}{16 \times 115200} = 8.68Divisor=16×11520016,000,000​=8.68

So:

* UARTIBRD = 8 (integer part)
* UARTFBRD ≈ 0.68 × 64 = 44 (fractional part)

This lets the UART support **any baud rate** with good accuracy.

**🔹 Transmission**

* Data to send is written into the **Transmit FIFO (32 bytes)**.
* UART hardware automatically sends it out, bit by bit, based on settings.
* The **BUSY signal** goes HIGH as soon as data is written and stays HIGH until:
  + FIFO is empty
  + Last character + stop bits are transmitted.

**🔹 Reception**

* Received data is stored in the **Receive FIFO (32 bytes)**.
* Each entry has **extra 4 bits** for error info.

How it works:

1. UART monitors the input line (UARTRXD).
   * Idle state = HIGH (marking).
   * Start bit = LOW (detected).
2. Timing:
   * UART uses an internal **Baud16 clock** (16× baud rate).
   * Start bit checked at **8th cycle** → ensures valid start.
   * Data bits sampled every 16 cycles.
   * Parity checked (if enabled).
   * Stop bit checked (must be HIGH, else framing error).
3. The **word + error bits** are pushed into the receive FIFO.

**🔹 Error Handling (stored with each character in FIFO)**

* **Bit 11 – Overrun:** FIFO full, new data lost.
* **Bit 10 – Break error:** Line stayed LOW for entire frame.
* **Bit 9 – Parity error:** Received parity mismatch.
* **Bit 8 – Framing error:** Stop bit missing (not HIGH).
* **Bits 7:0 – Received data.**

**🔹 FIFO vs. No FIFO Mode**

* Normally: 32-byte FIFOs handle data.
* If FIFOs are disabled → UART acts like **1-byte buffer**.
  + Easier for legacy software.
  + Overrun happens if you don’t read the byte before the next arrives.

**🔹 Loopback Test Mode**

* Setting the **Loop Back Enable (LBE)** bit makes TX feed directly back to RX.
* Useful for **self-testing** without external connections.

✅ **In short:**  
UART operation covers **baud rate generation**, **transmission via FIFO**, **reception with error detection**, **FIFO or 1-byte buffer modes**, and **loopback testing**.

**🔹 1. Purpose of the SIR ENDEC**

* Converts between:
  + **UART stream** (standard asynchronous bits: start, data, parity, stop).
  + **IrDA SIR (Serial Infrared)** pulses for half-duplex optical communication.
* It **doesn’t do analog work** (like LED drive or photodiode amplification).
  + Instead:
    - It outputs a **digital encoded pulse** (nSIROUT) for the IR transmitter driver.
    - It accepts a **digital decoded signal** (SIRIN) from the IR receiver.

**🔹 2. Normal IrDA Mode**

* **Encoding on transmit**:
  + Logic **0** → a **HIGH pulse** that is **3/16 of a bit time** long, output on nSIROUT.
  + Logic **1** → a **continuous LOW** on nSIROUT.
  + (That HIGH drives the IR LED, producing a short optical flash for each “0”.)
* **Decoding on receive**:
  + Incoming light → turns IR phototransistor ON → pulls output LOW → drives SIRIN low.
  + These transitions are decoded back into UART bits.

So basically:

* **UART 0 → IR pulse**
* **UART 1 → No pulse**

**🔹 3. Low-Power IrDA Mode**

* Normally, IrDA uses **3/16 of bit time** pulses.
* In **low-power mode**, the pulse width is **fixed** instead of proportional to baud:
  + Width = 3 × IrLPBaud16 cycles = about **1.63 µs** (if IrLPBaud16 ≈ 1.8432 MHz).
  + Controlled by **SIRLP bit in UARTCR** register.

This makes power consumption more predictable across baud rates.

**🔹 4. Baud Clock for IrDA Low-Power**

* A special clock **IrLPBaud16** is generated from UARTCLK by a divider stored in **UARTILPR**.
* Formula:

Low-power divisor=FUARTCLKFIrLPBaud16\text{Low-power divisor} = \frac{F\_{UARTCLK}}{F\_{IrLPBaud16}}Low-power divisor=FIrLPBaud16​FUARTCLK​​

* With constraint:

1.42 MHz<FIrLPBaud16<2.12 MHz1.42 \, \text{MHz} < F\_{IrLPBaud16} < 2.12 \, \text{MHz}1.42MHz<FIrLPBaud16​<2.12MHz

(Nominal target: 1.8432 MHz).

**🔹 5. Half-Duplex Requirement**

* IrDA SIR is **half-duplex**: can’t TX and RX simultaneously.
* After a transmission, software must wait at least **10 ms** before receiving.
* This accounts for **receiver saturation/latency** (IR photodiode “blinds” itself from local LED).
* The UART **does not handle this delay automatically** → you must implement it in software.

**🔹 6. Loopback Testing**

* **System test mode** lets you connect TX to RX internally.
* Enable both:
  + LBE (Loop Back Enable) in **UARTCR**.
  + SIRTEST in **UARTTCR** (Test Control Register).
* Data on nSIROUT is fed back into SIRIN.
* Useful for self-testing without actual IR hardware.

**🔹 7. Summary of Key Points**

* **Normal IrDA** → 3/16 bit-time pulses for “0”.
* **Low-power IrDA** → fixed ~1.6 µs pulse width regardless of baud.
* **Half-duplex** → must wait 10 ms between TX and RX (software enforced).
* **Divider register (UARTILPR)** generates IrDA low-power clock.
* **Loopback test** → use LBE + SIRTEST.

**UART modem operation**,

specifically how UART can act as either:

* **DTE (Data Terminal Equipment)** – e.g., a computer, terminal, or microcontroller.
* **DCE (Data Communication Equipment)** – e.g., a modem, network device, or communication interface.

UART supports **handshaking** using modem control signals (the extra pins besides TX/RX).

### 📌 Key Idea

* In **DTE mode**, UART behaves like a computer/terminal.
* In **DCE mode**, UART behaves like a modem.
* The signals flip roles depending on which mode is selected.

### 📊 Table 2-2: Function of the signals

| **Signal (Pin)** | **DTE Function** | **DCE Function** |
| --- | --- | --- |
| **nUARTCTS** (Clear to Send) | **Clear to send** (from DCE → DTE) → tells terminal it can transmit. | **Request to send** (from DCE → peer) → modem requests permission. |
| **nUARTDSR** (Data Set Ready) | **Data set ready** (DCE ready for comms). | **Data terminal ready** (DTE is ready). |
| **nUARTDCD** (Data Carrier Detect) | **Data carrier detect** (DCE senses valid carrier from phone line). | Same function but used differently for modem role. |
| **nUARTRI** (Ring Indicator) | **Ring indicator** (telephone line is ringing). | Same but indicates signal to DCE. |
| **nUARTRTS** (Request to Send) | **Request to send** (DTE → DCE). | **Clear to send** (DCE → peer). |
| **nUARTDTR** (Data Terminal Ready) | **Data terminal ready** (DTE wants to communicate). | **Data set ready** (DCE ready to accept). |
| **nUARTOUT1** | **Data carrier detect** (user defined). | Same but used as a modem signal. |
| **nUARTOUT2** | **Ring indicator** (user defined). | Same but used as a modem signal. |

### 🔄 How it works in practice

1. **DTE ↔ DCE handshaking**
   * Example: PC (DTE) sends **RTS** → modem (DCE) responds with **CTS**.
   * This ensures data is only sent when the channel is ready.
2. **Connection status**
   * **DSR/DTR** handshake tells each side that the other is ready.
   * **DCD** shows if a carrier (communication line) is active.
3. **Incoming call / notification**
   * **RI (Ring Indicator)** alerts the DTE when a call is incoming.

✅ **In short:**

* UART can act as either a **terminal (DTE)** or a **modem (DCE)**.
* The **same physical pins** are used, but their **meaning flips** depending on mode.
* These signals handle **flow control, readiness, and line status** for reliable communication.

**2.6: UART Hardware Flow Control**.

**🔹 What is Hardware Flow Control?**

When two UART devices talk, they need a way to avoid:

* **Overflow** → receiver’s buffer gets full, but sender keeps transmitting.
* **Underrun** → sender waits unnecessarily when receiver is ready.

This is solved with **RTS (Request to Send)** and **CTS (Clear to Send)** signals.

**🔹 How it works (based on your diagram)**

* **UART1 RTS → UART2 CTS**
* **UART2 RTS → UART1 CTS**

This cross-wiring means:

* Each UART tells the other when it’s ready (RTS).
* Each UART listens to CTS before sending.

**🔹 RTS Flow Control (Receive Side)**

* **nUARTRTS = Request To Send (output pin)**
* Controlled by **receive FIFO watermark** (a threshold you configure).
* **When enabled**:
  + RTS is asserted (LOW, because of the n in the name) until the RX FIFO gets too full.
  + Once FIFO reaches the **watermark level**, RTS is deasserted → telling the sender: “Stop, I can’t take more right now.”
  + RTS is reasserted once FIFO empties below the watermark.

👉 Basically: **RTS tells the peer when to pause sending.**

**🔹 CTS Flow Control (Transmit Side)**

* **nUARTCTS = Clear To Send (input pin)**
* Checked by the **transmitter** before sending each byte.
* **When enabled**:
  + If CTS is asserted → transmitter sends.
  + If CTS is deasserted → transmitter stops (but finishes the current byte before halting).

👉 Basically: **CTS tells the local UART when it can transmit.**

**🔹 Control via Registers**

In **UARTCR register**:

* **CTSEn = 1, RTSEn = 1** → both RTS & CTS enabled.
* **CTSEn = 1, RTSEn = 0** → only CTS enabled.
* **CTSEn = 0, RTSEn = 1** → only RTS enabled.
* **CTSEn = 0, RTSEn = 0** → no hardware flow control.

⚠️ Note: If RTS is controlled by hardware flow control, software cannot directly toggle it.

**✅ In short**

* **RTS** = “I’m ready, you may send me data.”
* **CTS** = “You are allowed to send me data now.”
* They work together to **prevent FIFO overflow/underrun** and make UART more reliable at high speeds.

**2.7: UART DMA Interface**.

# 🔹 What is UART DMA Interface?

Normally, the CPU has to **read/write every byte** from/to the UART FIFO.  
That’s slow if you’re handling large data streams.

**DMA (Direct Memory Access)** lets the UART talk directly to memory, without the CPU moving every byte.  
The CPU just sets up the DMA, then DMA + UART handle the rest.

# 🔹 UART DMA Signals

There are **two sets of signals**:

1. **For Receive (RX)** → moving data from UART to memory
2. **For Transmit (TX)** → moving data from memory to UART

### 1️⃣ Receive Side

* **UARTRXDMASREQ** = Single character DMA request  
  → Asserted when there’s **at least 1 character** in the RX FIFO.
* **UARTRXDMABREQ** = Burst DMA request  
  → Asserted when RX FIFO has **more than the watermark level**.  
  (Watermark is programmable using UARTIFLS register).
* **UARTRXDMACLR** = Request clear (from DMA controller)  
  → Clears the DMA request after transfer is completed.

### 2️⃣ Transmit Side

* **UARTTXDMASREQ** = Single character DMA request  
  → Asserted when there’s **at least 1 empty slot** in TX FIFO.
* **UARTTXDMABREQ** = Burst DMA request  
  → Asserted when TX FIFO has **fewer characters than watermark level**.
* **UARTTXDMACLR** = Request clear  
  → Clears the TX DMA request after transfer is completed.

# 🔹 Single vs Burst Transfer

* **Single transfer** → moves **1 character** at a time.
* **Burst transfer** → moves **multiple characters** (depending on watermark setting).
* Both can be asserted **at the same time**:
  + If FIFO > watermark → burst request **+** single request active.
  + If FIFO < watermark → only single request active.

👉 Example:

* You need to receive 19 characters.
* Watermark = 4.
* DMA does: 4 bursts × 4 chars (16 total) + 3 single transfers = 19 chars complete.

# 🔹 FIFO Disabled Case

If **FIFO is disabled** → UART works in **character mode** (1 char buffer).

* Only **single transfer requests** (RXDMASREQ / TXDMASREQ) are possible.
* No burst transfers.

# 🔹 Watermark Levels & Burst Trigger Points

The watermark defines **when burst requests fire**.

From **Table 2-4**:

| **Watermark** | **TX (empty slots)** | **RX (filled slots)** |
| --- | --- | --- |
| 1/8 | 28 | 4 |
| 1/4 | 24 | 8 |
| 1/2 | 16 | 16 |
| 3/4 | 8 | 24 |
| 7/8 | 4 | 28 |

👉 Example:

* RX watermark = 1/2 → burst DMA request when FIFO has ≥16 chars.
* TX watermark = 1/4 → burst DMA request when FIFO has ≤24 empty slots.

# 🔹 DMA Error Handling

* **DMAONERR bit (UARTDMACR)**:
  + If set → UART disables RX DMA requests when an **error interrupt (UARTEINTR)** occurs.
  + Requests resume only when the error is cleared.
  + TX DMA is **not affected**.

# 🔹 Timing Diagram (Figure 2-7)

* Shows how **requests (SREQ/BREQ)** stay active until DMA asserts **DMACLR**.
* Everything is synchronized to **PCLK**.
* Ensures smooth handshake between UART and DMA controller.

# ✅ In short

* UART DMA reduces CPU load by automating data movement.
* **Single request** = 1 character, **Burst request** = multiple (based on watermark).
* RX/TX each have **SREQ, BREQ, and CLRs**.
* Watermark thresholds decide when burst requests happen.
* Errors can disable RX DMA until cleared.
* If FIFO disabled → only single transfers are possible.

2.8 UART **Interrupts**

(2.8 UART **Interrupts**) explains how the UART generates **different types of interrupts**, how they are masked/cleared, and how you can handle them in software or via an interrupt controller.

Let’s break it into **concepts → types → handling**:

# 🔹 1. Overview of UART Interrupts

* The UART has **11 different interrupt sources**, but instead of giving you 11 separate outputs, they are grouped into **5 functional interrupt outputs**:
  1. **UARTRXINTR** → Receive interrupt (RX data ready).
  2. **UARTTXINTR** → Transmit interrupt (TX FIFO needs data).
  3. **UARTRTINTR** → Receive timeout interrupt (RX idle).
  4. **UARTMSINTR** → Modem status interrupt (RI, CTS, DCD, DSR changes).
  5. **UARTEINTR** → Error interrupt (framing, parity, break, overrun).
* Finally, there is a **combined interrupt output (UARTINTR)** which is just an **OR** of all 5 outputs.  
  👉 This is useful if you want only **one system interrupt line** instead of multiple.
* Each interrupt can be **enabled/disabled individually** using the **Interrupt Mask Set/Clear Register (UARTIMSC)**.

# 🔹 2. Types of Interrupts

### 2.1 **UARTMSINTR (Modem status interrupt)**

* Triggered when **any modem status pin changes**:
  + nUARTCTS → Clear to Send
  + nUARTDCD → Data Carrier Detect
  + nUARTDSR → Data Set Ready
  + nUARTRI → Ring Indicator
* Cleared by writing 1 to the corresponding bit in **UARTICR** (Interrupt Clear Register).

### 2.2 **UARTRXINTR (Receive interrupt)**

* Triggered when **data is available in RX FIFO**:
  + If **FIFO enabled**: Raised when RX FIFO reaches the **trigger level** (programmable).
  + If **FIFO disabled** (only 1-byte buffer): Raised when data arrives and fills it.
* Cleared by:
  + Reading data from FIFO until below trigger level, or
  + Writing 1 to clear in **UARTICR**.

### 2.3 **UARTTXINTR (Transmit interrupt)**

* Triggered when **TX FIFO needs more data**:
  + If **FIFO enabled**: Raised when TX FIFO ≤ trigger level.
  + If **FIFO disabled**: Raised when the single holding register is empty.
* Cleared by:
  + Writing more data into FIFO, or
  + Writing 1 to clear in **UARTICR**.

⚠️ **Important Note**: The interrupt is based on a **transition through the level**, not just the level itself.  
👉 If UART is enabled with empty FIFO, **no interrupt is generated immediately** — it only triggers **after at least one character has been sent and the FIFO becomes empty**.

### 2.4 **UARTRTINTR (Receive timeout interrupt)**

* Triggered when:
  + RX FIFO is **not empty**, and
  + No new data arrives for **32 bit periods** (idle condition).
* Cleared when:
  + FIFO becomes empty (all data read), or
  + Writing 1 to **UARTICR**.

👉 Useful for protocols where RX may pause (like UART streaming without fixed packet size).

### 2.5 **UARTEINTR (Error interrupt)**

* Triggered when any **RX error** occurs:
  + **Overrun error** (FIFO full, new char lost).
  + **Break** (RX held low too long).
  + **Parity error**.
  + **Framing error** (bad stop bit).
* Cleared by writing to bits 7–10 in **UARTICR**.

### 2.6 **UARTINTR (Combined interrupt)**

* Simply an **OR of all 5 interrupts**.
* Lets you connect UART to a single system interrupt line and then check **UARTMIS / UARTRIS** to see which sub-interrupt caused it.

# 🔹 3. How interrupts are handled in software

1. **Check source**:
   * Use **UARTRIS (Raw Interrupt Status)** or **UARTMIS (Masked Interrupt Status)** to see the cause.
2. **Handle it**:
   * If RX → read data from FIFO.
   * If TX → write more data.
   * If error → check status & clear.
   * If modem change → handle signal change.
3. **Clear it**:
   * Write 1 to the relevant bit(s) in **UARTICR**.

# ✅ In short

* UART generates **5 categories of interrupts** (RX, TX, Timeout, Modem, Error).
* They can be enabled/disabled individually.
* There’s also a **global combined interrupt (UARTINTR)**.
* Each interrupt is cleared by **servicing the condition** or writing to **UARTICR**.