ATM SYSTEM DESIGN

WITH DATABASE INTEGRATION

Table of Contents

OBJECTIVE: 4
PROJECT OVERVIEW: 4
BLOCK DIAGRAM: 4
HARDWARE REQUIREMENTS: 5
SOFTWARE REQUIREMENTS: 5
MICROCONTROLLER SIDE IMPLEMENTATION 5
PC-SIDE DATABASE IMPLEMENTATION 6
RFID Data Format Clarification 8
LPC2148 9
Memory Map of LPC2148 15
LPC2148 (ARM-7 Based) µC: Feature Set 19
GPIO Registers in LPC2148 22
LCD 26
KEYPAD MATRIX 56
BUZZER 64
Electromagnetic Buzzer 67
RIFD READER AND CARD 69
RFID System Components 77
125kHz Embedded RFID Reader Module (YET-125K) 80
EM4095 82
UART (Universal Asynchronous Receiver Transmitter) 86
How UART Works 87
UART – LPC2148 94
MAX232 (for UART level shifting) 105
DB-9 Cable (db9 rs232 9pin serial cable male to female) 112
What is RS-232? 114
Software 125

BANK DATABASED 139
ATM CONNECTED 158
MICROCONTROLLER SIDE 173
ATM Communication Protocol: MCU ↔ PC Commands 213
ATM Project's Working Flow 215
Advantages 217
Applications 218
Outputs: 219
LPC2148 Interfacing Connections Table 220
OUTPUT PHOTOS 221

ATM System Design with Database Integration

OBJECTIVE:

The main aim of this project is to develop a **secure ATM system** using **RFID authentication** and a **PIN-based interface**, with backend **banking database** integration implemented in C using data structures

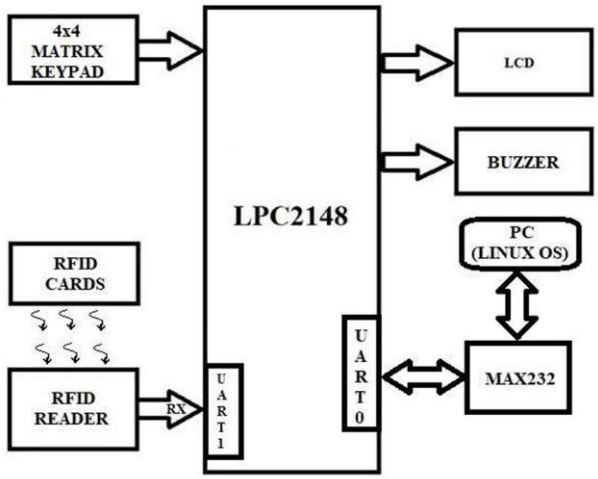
PROJECT OVERVIEW:

The system consists of two main components:

1. **Front-End (Microcontroller (LPC2148) side):**
   Acts like a real ATM interface:
   - RFID card reader to identify user
   - Keypad for PIN entry and transaction inputs
   - LCD display for menu and status messages
   - Communicates with backend PC via UART
   - Handles the ATM interface (LCD, keypad, RFID reader)

2. **Back-End (PC (Linux) side Application in C):**
   Simulates a banking system:
   - Stores user and transaction data using `struct` and file I/O
   - Receives commands via UART from MCU
   - Sends results and balance info back to MCU
   - Simulates a banking database using data structures and file handling

BLOCK DIAGRAM:



HARDWARE REQUIREMENTS:

1. **LPC2148 Microcontroller**: The brain of your ATM system
2. **RFID Reader and Cards**: For user authentication
3. **16x2 LCD Display**: For user interface
4. **4x4 Matrix Keypad**: For PIN and amount input
5. **MAX232**: For UART level shifting
6. **Buzzer**: For alerts/notifications
7. **USB-to-UART Converter/DB-9 Cable**: For communication between MCU and PC

SOFTWARE REQUIREMENTS:

1. **Embedded C**: For microcontroller programming
2. **Keil uVision IDE**: For development environment
3. **Flash Magic**: For flashing the program to MCU
4. **GCC Compiler**: For PC-side program compilation

MICROCONTROLLER SIDE IMPLEMENTATION

1. Hardware Interface Modules

Copy these essential modules to your MCU project folder:

- lcd.c, lcd.h (LCD control)
- delay.c, delay.h (timing functions)
- uart.c, uart.h (UART communication)
- keypad.c, keypad.h (keypad input)

2. Testing Individual Modules

Before integration, test each module separately:

⇒ **LCD Test**: Display characters and strings
⇒ **Keypad Test**: Read and display input values
⇒ **UART Test**: Send/receive test strings using UART0 and UART1
⇒ **RFID Test**: Read card data via UART1 and display on LCD

3. Main Program Flow (main.c)

1. **Initialization**:
   - Initialize all hardware modules (LCD, UART, keypad, RFID)
   - Display project title briefly on LCD
2. **RFID Authentication**:
   - Continuously wait for RFID card
   - When card is detected, read the 10-byte frame (starts with 0x02, ends with 0x03)

    - Extract the 8-byte card number (middle bytes)
    - Send to PC in format: #CARD:12345678$
3. **PIN Verification**:
    - If PC responds with @OK#VALID_CARDS, display "Enter PIN"
    - Read 4-digit PIN from keypad
    - Send to PC in format: #CARD:12345678#PIN:4321$
    - Wait for validation response
4. **Main Menu**:
    - On successful login, display menu options:
        1. BALANCE
        2. DEPOSIT
        3. WITHDRAW
        4. PIN CHANGE
        5. MINI STATEMENT
        6. EXIT
    - Implement 30-second timeout using timer interrupts
5. **Transaction Handling**:
    - For each menu option, send appropriate request to PC and handle response

PC-SIDE DATABASE IMPLEMENTATION

Main Program Flow

1. **Initialization**:
    - Load user data from users.txt into an array/list
    - Load transaction history from transactions.txt into a linked list
    - Initialize UART communication with MCU
2. **RFID Validation**:
    - Wait for #CARD:12345678$
    - Search user list for matching RFID
    - Respond with @OK#VALID_CARDS or @ERR#INVALID_CARDS
3. **PIN Validation**:
    - Wait for #CARD:12345678#PIN:4321$
    - Verify PIN matches stored PIN for that RFID
    - Respond with @OK#LOGIN_SUCCESS or @ERR#INVALID_PINS
4. **Transaction Handling**:
    - **Balance Enquiry** (#TXN:BALANCE#REQS):
        - Fetch and return balance: @OK#BALANCE:XXXX.XX$
    - **Deposit** (#TXN:DEPOSIT#AMOUNT:1000$):
        - Add amount to balance
        - Log transaction

        - Return new balance
    - **Withdrawal** (#TXN:WITHDRAW#AMOUNT:500$):
        - Check sufficient balance
        - Deduct amount if available
        - Log transaction
        - Return new balance or error
    - **PIN Change** (#PINCHANGE#NEWPINS):
        - Update PIN in user structure
        - Save to file
    - **Mini Statement** (#MINISTMT#REQS):
        - Find last 3 transactions for user
        - Format as: @OK#MINISTMT:TXN1|TXN2|TXN3$
5. **File Handling**:
    - After each transaction that modifies data, update the text files
    - Implement proper file locking to prevent corruption

MCU-to-PC and PC-to-MCU COMMUNICATION COMMANDS

Transaction Type	MCU → PC (Request Format)	PC → MCU (Response Format)	Description
RFID Validation	#CARD:12345678$	@OK#VALID_CARDS	MCU sends RFID card number; PC validates.
		@ERR#INVALID_CARDS	Invalid RFID card.
PIN Verification	#CARD:12345678#PIN:4321$	@OK#LOGIN_SUCCESS	Correct PIN.
		@ERR#INVALID_PINS	Wrong PIN.
Balance Enquiry	#TXN:BALANCE#REQS	@OK#BALANCE:XXXX.XX$	Returns current balance.
		@ERR#BALANCE_FAILED	Failed to fetch balance.
Deposit	#TXN:DEPOSIT#AMOUNT:1000$	@OK#BALANCE:XXXX.XX$	Updates balance after deposit.
		@ERR#DEPOSIT_FAILED	Deposit failed.
Withdrawal	#TXN:WITHDRAW#AMOUNT:500$	@OK#BALANCE:XXXX.XX$	Updates balance after withdrawal.
		@ERR#INSUFFICIENT_BALANCE$	Insufficient funds.
PIN Change	#PINCHANGE#NEWPIN:5678$	@OK#PINCHANGED$	PIN updated successfully.
		@ERR#PINCHANGE_FAILED$	PIN change failed.
Mini Statement	#MINISTMT#REQS	@OK#MINISTMT:TXN1|TXN2|TXN3$	Last 3 transactions (e.g., Deposit-

			100@10:00|Withdraw-50@12:00).
		@ERR#MINISTMT_FAILED$	Failed to fetch transactions.
Timeout/Error	(N/A)	@ERR#TIMEOUT$	User inactive for 30 seconds.
Session End	(User selects EXIT)	@OK#SESSION_ENDED$	Terminates session.

Key Notes:

1. **Message Structure**:
   - **MCU → PC**: Starts with #, ends with $ (e.g., #CARD:12345678$).
   - **PC → MCU**: Starts with @OK (success) or @ERR (error), ends with $.
2. **Example Workflow**:
   - **RFID Scan**:
     MCU → PC: #CARD:12345678$
     PC → MCU: @OK#VALID_CARDS
   - **PIN Entry**:
     MCU → PC: #CARD:12345678#PIN:4321$
     PC → MCU: @OK#LOGIN_SUCCESS
   - **Balance Check**:
     MCU → PC: #TXN:BALANCE#REQS
     PC → MCU: @OK#BALANCE:1500.00$
3. **Error Handling**:
   - If RFID is invalid: @ERR#INVALID_CARDS
   - If PIN is wrong: @ERR#INVALID_PINS
   - If withdrawal exceeds balance: @ERR#INSUFFICIENT_BALANCE$

RFID Data Format Clarification

0x02 + 8 digit + 0x03

This is a **common RFID data packet format** used by 125kHz RFID readers like **YET-125K**, which use ASCII over UART.

Byte	Purpose
0x02	Start of Text (STX)
XXXXXXXX	8-digit ASCII Tag ID
0x03	End of Text (ETX)

📄 **Example Output:**
If the RFID tag sends:

0x02 31 32 33 34 35 36 37 38 0x03

It corresponds to:

0x02 + "12345678" + 0x03

LPC2148

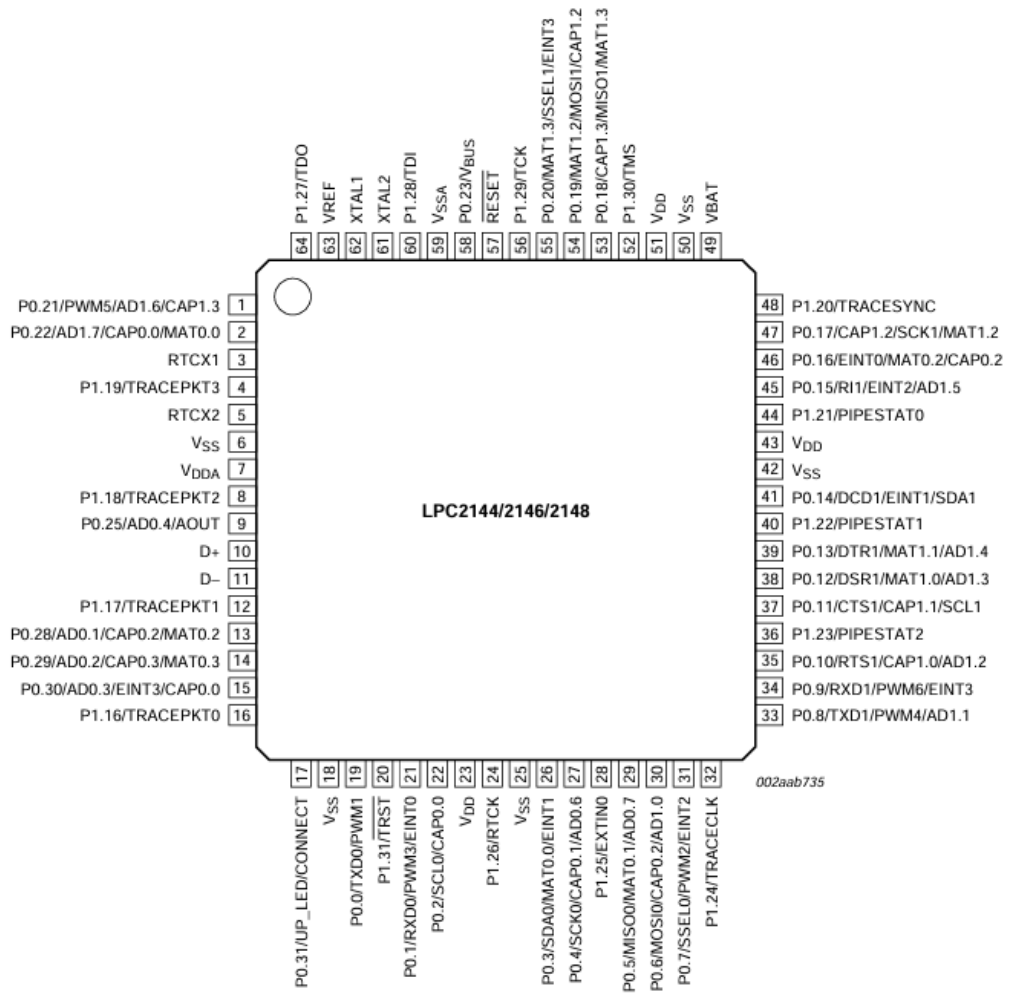
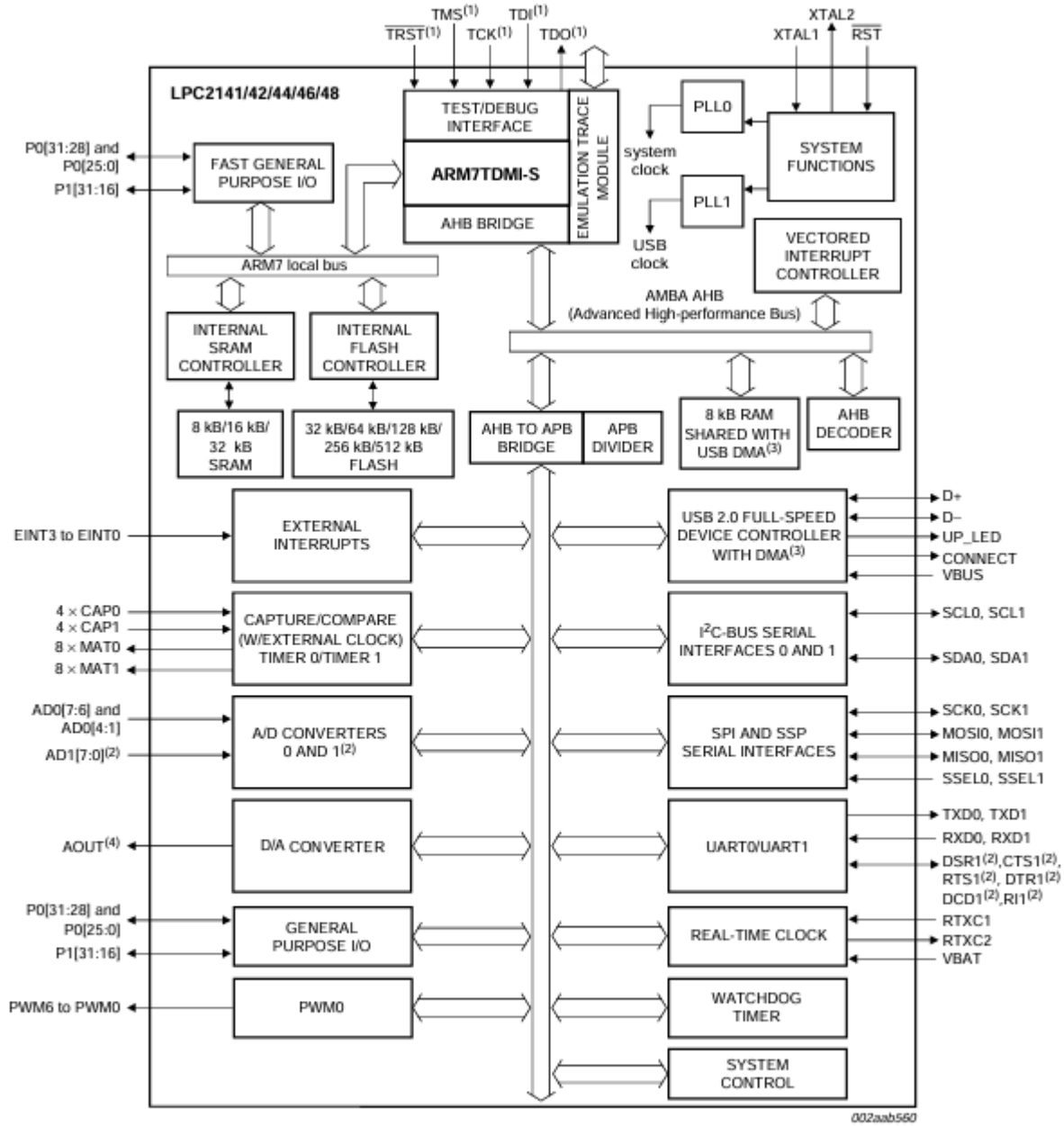


Fig 4. LPC2144/46/48 pinning



(1) Pins shared with GPIO.
(2) LPC2144/46/48 only.
(3) USB DMA controller with 8 kB of RAM accessible as general purpose RAM and/or DMA is available in LPC2146/48 only.
(4) LPC2142/44/46/48 only.

Fig 1. Block diagram

ARM LPC2148 microcontroller architecture, which is part of the **LPC2100 series** from NXP Semiconductors. This microcontroller is based on the **ARM7TDMI-S core** and is designed for embedded applications. Below is a detailed explanation of the architecture and its components:

1. ARM7TDMI-S Core

- **Core**: The LPC2148 is built around the **ARM7TDMI-S** core, which is a 32-bit RISC processor.
- **Features**:
  - Supports both **ARM** (32-bit) and **Thumb** (16-bit) instruction sets.
  - Includes a **3-stage pipeline** for efficient instruction execution.
  - Provides **low power consumption** and high performance.

---

2. Memory System

1. **Internal SRAM**:
   - The microcontroller includes **32 KB of internal SRAM** for fast data storage and execution.
2. **Internal Flash Memory**:
   - The LPC2148 has **512 KB of internal flash memory** for storing program code.
   - The flash memory is managed by an **internal flash controller**.
3. **External Memory Interface**:
   - The microcontroller supports external memory through the **AMBA (Advanced Microcontroller Bus Architecture)** interface.

---

3. AMBA Bus System

- **AHB (Advanced High-performance Bus)**:
  - Used for high-speed communication between the ARM core, memory, and high-performance peripherals.
  - Supports **burst transfers** and **pipelined operations**.
- **APB (Advanced Peripheral Bus)**:
  - Used for low-speed communication with peripherals.
  - Connected to the AHB via an **AHB-to-APB bridge**.

---

4. Peripherals

The LPC2148 microcontroller includes a wide range of peripherals, making it suitable for various embedded applications:

a. General-Purpose I/O (GPIO)

- **Fast GPIO pins** for interfacing with external devices.
- Each pin can be configured as input or output.

b. Serial Interfaces

1. **UART (Universal Asynchronous Receiver-Transmitter)**:
   - Supports full-duplex serial communication.
   - Includes features like hardware flow control (RTS, CTS) and modem control (DCD, DTR, DSR, RI).
2. **SPI (Serial Peripheral Interface)**:

    - Used for high-speed synchronous communication with peripherals.
    - Supports master and slave modes.
3. **I2C (Inter-Integrated Circuit)**:
    - Used for low-speed communication with peripherals.
    - Supports multi-master and slave modes.

c. Timers

1. **Capture/Compare Timer**:
    - Used for timing, event capture, and PWM (Pulse Width Modulation) generation.
2. **Watchdog Timer**:
    - Used to reset the system in case of software failures.

d. Analog-to-Digital Converter (ADC)

- **10-bit ADC** with multiple input channels.
- Used for converting analog signals (e.g., from sensors) to digital values.

e. Digital-to-Analog Converter (DAC)

- **10-bit DAC** for converting digital values to analog signals.

f. Real-Time Clock (RTC)

- Provides timekeeping functionality.
- Can be used for timestamping and scheduling tasks.

g. PWM (Pulse Width Modulation)

- Supports **PWM output** for controlling devices like motors and LEDs.

h. External Interrupts

- Supports external interrupts for real-time event handling.

---

5. System Control

1. **Reset Controller**:
    - Manages system reset operations.
2. **Power Control**:
    - Includes power-saving modes to reduce power consumption.
3. **Clock Generation**:
    - Includes an **on-chip oscillator** and **PLL (Phase-Locked Loop)** for clock generation.

---

6. Debug and Test Interface

- **JTAG (Joint Test Action Group)**:
    - Used for debugging and testing the microcontroller.
    - Includes pins for **TMS, TDI, TDO, TCK, TRST, and RTCK**.

---

7. Pin Configuration

- The LPC2148 microcontroller has a **64-pin LQFP package**.
- Pins are multiplexed to support multiple functions (e.g., GPIO, UART, SPI).

8. Key Features of LPC2148

- **ARM7TDMI-S Core**: 32-bit RISC processor with Thumb support.
- **Memory**: 512 KB flash, 32 KB SRAM.
- **Peripherals**: GPIO, UART, SPI, I2C, Timers, ADC, DAC, RTC, PWM.
- **Bus System**: AHB for high-speed communication, APB for peripherals.
- **Power Management**: Low-power modes for energy efficiency.
- **Debug Interface**: JTAG for debugging and testing.

Summary of LPC2148 Architecture

Component	Description
ARM7TDMI-S Core	32-bit RISC processor with Thumb support.
Memory	512 KB flash, 32 KB SRAM.
AHB Bus	High-speed bus for core, memory, and high-performance peripherals.
APB Bus	Low-speed bus for peripherals.
GPIO	Fast general-purpose I/O pins.
Serial Interfaces	UART, SPI, I2C for communication.
Timers	Capture/Compare Timer, Watchdog Timer.
ADC	10-bit analog-to-digital converter.
DAC	10-bit digital-to-analog converter.
RTC	Real-time clock for timekeeping.
PWM	Pulse Width Modulation for controlling devices.
Debug Interface	JTAG for debugging and testing.

Memory Map of LPC2148

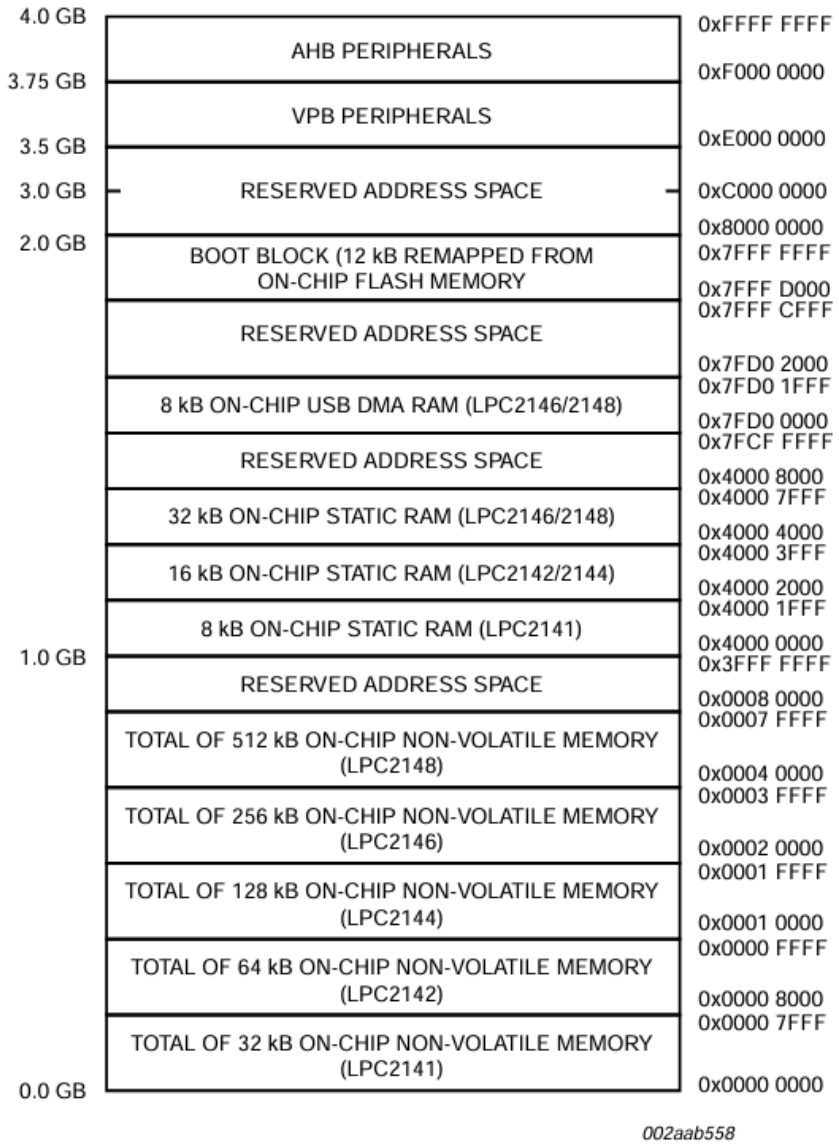


Fig 5. LPC2141/42/44/46/48 memory map

memory map for the **LPC2141/42/44/46/48 microcontrollers**, which are part of the **LPC2100 series** from NXP Semiconductors. The memory map shows how the **4 GB address space** is divided between different memory regions and peripherals. Below is a detailed explanation of the memory map and its components:

---

1. Overview of the Memory Map

- The ARM7-based LPC214x microcontrollers use a **32-bit address bus**, allowing access to a **4 GB address space** (0x00000000 to 0xFFFFFFFF).
- The memory map is divided into regions for **on-chip memory**, **peripherals**, and **reserved spaces**.

---

2. Memory Regions

a. On-Chip Non-Volatile Memory (Flash)

- **LPC2141**:
  - **32 KB** of on-chip flash memory.
  - Address range: **0x00000000 to 0x00007FFF**.
- **LPC2142**:
  - **64 KB** of on-chip flash memory.
  - Address range: **0x00000000 to 0x0000FFFF**.
- **LPC2144**:
  - **128 KB** of on-chip flash memory.
  - Address range: **0x00000000 to 0x0001FFFF**.
- **LPC2146**:
  - **256 KB** of on-chip flash memory.
  - Address range: **0x00000000 to 0x0003FFFF**.
- **LPC2148**:
  - **512 KB** of on-chip flash memory.
  - Address range: **0x00000000 to 0x0007FFFF**.
- **Purpose**: Stores the program code and constants.

b. On-Chip Static RAM (SRAM)

- **LPC2141**:
  - **8 KB** of on-chip SRAM.
  - Address range: **0x40000000 to 0x40001FFF**.
- **LPC2142/2144**:
  - **16 KB** of on-chip SRAM.
  - Address range: **0x40000000 to 0x40003FFF**.
- **LPC2146/2148**:
  - **32 KB** of on-chip SRAM.
  - Address range: **0x40000000 to 0x40007FFF**.
- **Purpose**: Used for data storage and stack during program execution.

c. USB DMA RAM

- **LPC2146/2148**:
  - **8 KB** of on-chip USB DMA RAM.
  - Address range: **0x7FD00000 to 0x7FD01FFF**.

- **Purpose**: Used for USB data transfers.

d. Boot Block

- **Boot Block**: A special region of flash memory that contains the bootloader code.
- Address range: **0x7FFFD000 to 0x7FFFFFFF** (remapped from on-chip flash memory).
- **Purpose**: Used for system initialization and booting.

---

3. Peripheral Regions

a. AHB (Advanced High-performance Bus) Peripherals

- **AHB Peripherals**: High-speed peripherals connected to the AHB bus.
- Address range: **0xE0000000 to 0xFFFFFFFF**.
- **Purpose**: Includes high-performance peripherals like the **DMA controller** and **external memory interface**.

b. VPB (VLSI Peripheral Bus) Peripherals

- **VPB Peripherals**: Low-speed peripherals connected to the VPB bus (similar to APB in newer ARM architectures).
- Address range: **0xE0000000 to 0xEFFFFFFF**.
- **Purpose**: Includes peripherals like **UART**, **SPI**, **I2C**, **Timers**, and **GPIO**.

---

4. Reserved Address Space

- **Reserved Address Space**: Regions of the memory map that are not used or are reserved for future use.
- Address ranges:
  - **0x40008000 to 0x7FFFCFFF**: Reserved.
  - **0x80000000 to 0xDFFFFFFF**: Reserved.
- **Purpose**: These regions are not mapped to any physical memory or peripherals and should not be accessed.

---

5. Memory Map Summary

Address Range	Region	Description
0x00000000–0x00007FFF	On-Chip Flash Memory (LPC2141)	32 KB flash memory for program code.
0x00000000–0x0000FFFF	On-Chip Flash Memory (LPC2142)	64 KB flash memory for program code.
0x00000000–0x0001FFFF	On-Chip Flash Memory (LPC2144)	128 KB flash memory for program code.
0x00000000–0x0003FFFF	On-Chip Flash Memory (LPC2146)	256 KB flash memory for program code.
0x00000000–0x0007FFFF	On-Chip Flash Memory (LPC2148)	512 KB flash memory for program code.

0x40000000–0x40001FFF	On-Chip SRAM (LPC2141)	8 KB SRAM for data storage and stack.
0x40000000–0x40003FFF	On-Chip SRAM (LPC2142/2144)	16 KB SRAM for data storage and stack.
0x40000000–0x40007FFF	On-Chip SRAM (LPC2146/2148)	32 KB SRAM for data storage and stack.
0x7FD00000–0x7FD01FFF	USB DMA RAM (LPC2146/2148)	8 KB USB DMA RAM for USB data transfers.
0x7FFFD000–0x7FFFFFFF	Boot Block	Remapped bootloader code for system initialization.
0xE0000000–0xEFFFFFFF	VPB Peripherals	Low-speed peripherals (UART, SPI, I2C, Timers, GPIO).
0xF0000000–0xFFFFFFFF	AHB Peripherals	High-speed peripherals (DMA, external memory interface).
Reserved Regions	Reserved Address Space	Unused or reserved for future use.

---

6. Key Features of the Memory Map

1. **Unified Address Space**:
   - The 4 GB address space is shared between memory and peripherals, simplifying access.
2. **On-Chip Memory**:
   - Flash memory for program storage and SRAM for data storage.
3. **Peripheral Access**:
   - Peripherals are memory-mapped, allowing them to be accessed using load/store instructions.
4. **Boot Block**:
   - A dedicated region for the bootloader, ensuring reliable system initialization.
5. **USB DMA RAM**:
   - Dedicated RAM for USB data transfers, improving USB performance.

---

7. Example Use Cases

1. **Program Execution**:
   - The program code is stored in the **on-chip flash memory** and executed by the ARM7 core.
2. **Data Storage**:
   - Variables and stack are stored in the **on-chip SRAM**.
3. **Peripheral Control**:
   - Peripherals like UART, SPI, and Timers are accessed via their memory-mapped addresses in the **VPB** and **AHB** regions.

4. **System Initialization**:
    - The **Boot Block** contains the bootloader code, which initializes the system and loads the main program.
5. **USB Data Transfer**:
    - The **USB DMA RAM** is used for efficient USB data transfers.

LPC2148 (ARM-7 Based) µC: Feature Set

The **LPC2148 microcontroller**, based on the **ARM7TDMI-S core**, is a powerful and versatile 32-bit RISC microcontroller designed for embedded applications. Below is a detailed explanation of its **feature set**:

---

1. ARM7TDMI-S Core

- **High-Performance 32-bit RISC Processor**:
    - Executes both **ARM** (32-bit) and **Thumb** (16-bit) instruction sets.
    - Includes a **3-stage pipeline** for efficient instruction execution.
    - Provides **low power consumption** and high performance.

---

2. On-Chip Memory

1. **512 KB On-Chip Flash ROM**:
    - Address range: **0x00000000 to 0x0007FFFF**.
    - Supports **In-System Programming (ISP)** and **In-Application Programming (IAP)** for easy firmware updates.
    - Used for storing program code and constants.
2. **32 KB On-Chip SRAM**:
    - Address range: **0x40000000 to 0x40007FFF**.
    - Used for data storage and stack during program execution.
3. **8 KB USB DMA RAM**:
    - Address range: **0x7FD00000 to 0x7FD01FFF**.
    - Dedicated RAM for USB data transfers, improving USB performance.

---

3. General-Purpose I/O (GPIO)

- **Up to 45 GPIO Pins**:
    - Can be configured as input or output.
    - Used for interfacing with external devices (e.g., LEDs, buttons, sensors).

---

4. Clock Generation

1. **CPU Clock up to 60 MHz**:
    - Provides high-speed processing for time-critical applications.
2. **On-Chip Crystal Oscillator**:

- Supports external crystals for clock generation.

3. **On-Chip PLL (Phase-Locked Loop)**:
    - Multiplies the external clock frequency to achieve higher CPU speeds.

---

5. Timers

- **Two 32-bit Timers (Timer0 & Timer1)**:
    - Used for timing, event capture, and PWM (Pulse Width Modulation) generation.
    - Can be used for tasks like measuring time intervals, generating delays, and controlling motors.

---

6. Serial Interfaces

1. **Two UARTs (UART0 & UART1)**:
    - Support full-duplex serial communication.
    - Include features like hardware flow control (RTS, CTS) and modem control (DCD, DTR, DSR, RI).
2. **I2C Serial Interface**:
    - Used for low-speed communication with peripherals.
    - Supports multi-master and slave modes.
3. **Two SPI Serial Interfaces (SPI0 & SPI1)**:
    - Used for high-speed synchronous communication with peripherals.
    - Supports master and slave modes.

---

7. Real-Time Clock (RTC)

- **RTC**:
    - Provides timekeeping functionality.
    - Can be used for timestamping and scheduling tasks.

---

8. Watchdog Timer

- **Watchdog Timer**:
    - Used to reset the system in case of software failures.
    - Ensures system reliability by detecting and recovering from software lockups.

---

9. Analog-to-Digital Converter (ADC)

- **8-Channel 10-bit ADC**:
    - Converts analog signals (e.g., from sensors) to digital values.
    - Used for applications like sensor interfacing and data acquisition.

---

10. Digital-to-Analog Converter (DAC)

- **10-bit DAC**:
    - Converts digital values to analog signals.

- Used for applications like audio output and motor control.

---

11. Pulse Width Modulation (PWM)

- **PWM Unit with up to 6 PWM Outputs**:
  - Used for controlling devices like motors, LEDs, and servos.
  - Provides precise control over power delivery.

---

12. USB Interface

- **USB 2.0 Full-Speed Device Controller**:
  - Supports USB communication for data transfer and device control.
  - Includes **8 KB of dedicated USB DMA RAM** for efficient data transfers.

---

13. Interrupt Controller

- **Vectored Interrupt Controller (VIC)**:
  - Supports up to **32 interrupts**.
  - Provides fast and efficient interrupt handling for real-time applications.

Summary of LPC2148 Features

Feature	Description
ARM7TDMI-S Core	32-bit RISC processor with Thumb support.
On-Chip Flash ROM	512 KB for program storage with ISP and IAP support.
On-Chip SRAM	32 KB for data storage and stack.
USB DMA RAM	8 KB dedicated RAM for USB data transfers.
GPIO Pins	Up to 45 general-purpose I/O pins.
CPU Clock	Up to 60 MHz with on-chip oscillator and PLL.
Timers	Two 32-bit timers for timing and PWM generation.
UARTs	Two UARTs for serial communication.
I2C Interface	One I2C interface for low-speed communication.
SPI Interfaces	Two SPI interfaces for high-speed communication.
Real-Time Clock (RTC)	Provides timekeeping functionality.
Watchdog Timer	Ensures system reliability by detecting software lockups.
ADC	8-channel 10-bit ADC for analog signal conversion.
DAC	10-bit DAC for digital-to-analog conversion.
PWM	Up to 6 PWM outputs for precise control of devices.
USB Interface	USB 2.0 Full-Speed Device Controller with dedicated DMA RAM.
Interrupt Controller	Vectored Interrupt Controller with up to 32 interrupts.

Feature	LPC2129	LPC2148
Core	ARM7TDMI-S	ARM7TDMI-S

Flash Memory	256 KB	512 KB
SRAM	16 KB	32 KB
USB DMA RAM	Not available	8 KB
GPIO Pins	Up to 46	Up to 45
UARTs	2 (UART0 & UART1)	2 (UART0 & UART1)
SPI Interfaces	2 (SPI0 & SPI1)	2 (SPI0 & SPI1)
I2C Interface	1	1
Timers	2 (Timer0 & Timer1)	2 (Timer0 & Timer1)
ADC Channels	4-channel 10-bit ADC	8-channel 10-bit ADC
DAC	Not available	10-bit DAC
PWM Outputs	Up to 6	Up to 6
CAN Channels	2 (CAN1 & CAN2)	Not available
USB Interface	Not available	USB 2.0 Full-Speed Device Controller
CPU Clock	Up to 60 MHz	Up to 60 MHz
On-Chip Oscillator	Yes	Yes
On-Chip PLL	Yes	Yes
Vectored Interrupt Controller	Supports up to 32 interrupts	Supports up to 32 interrupts
Watchdog Timer	Yes	Yes
Real-Time Clock (RTC)	Yes	Yes

GPIO Registers in LPC2148

LPC2148 has **two GPIO ports**:

1. **PORT 0 (P0.x)** → 32 pins available.
2. **PORT 1 (P1.x)** → Limited to 16 pins (P1.16 to P1.31).

Each GPIO port has **four key registers** to control digital input/output.

Register	Address (Port 0)	Address (Port 1)	Description
IODIR	0xE0028008	0xE0028018	Configures pin as Input (0) or Output (1)
IOSET	0xE0028004	0xE0028014	Sets specified output pins HIGH (1)
IOCLR	0xE002800C	0xE002801C	Clears (sets LOW) specified output pins
IOPIN	0xE0028000	0xE0028010	Reads current pin states (input/output)

Basic GPIO Pin Operations

- **Set pin as input**: IODIRx &= ~(1 << n);
- **Set pin as output**: IODIRx |= (1 << n);
- **Set pin HIGH**: IOSETx = (1 << n);
- **Set pin LOW**: IOCLRx = (1 << n);
- **Read pin value**: state = (IOPINx & (1 << n)) ? 1 : 0;

IODIRx: To configure the direction of a grid pin as input or output

To configure a particular gpio pin as output:

```
IODIRx |= 1 << BITPOS;
```

To configure a few particular gpio pins as output:

```
IODIRx |= (1 << BITPOSa) | (1 << BITPOSb) | ... | (1 << BITPOSn);
```

To configure contiguous pins as gpio output:

```
IODIRx |= 0xFF << BITSTARTPOS;
```

To configure 4 contiguous pins as gpio output:

```
IODIRx |= 0x0F << BITSTARTPOS;
```

To configure a particular gpio pin as input:

```
IODIRx &= ~(1 << BITPOS);
```

Note: The default value in IODIRx is all 0's(32bits) hence all pins are default input

IOSETx: To exclusively write logic '1' onto a gpio pin without affecting other pins in case some/all pins are configured as gpio pins

To set a gpio pin:

```
IOSETx = 1 << BITPOS;
```

To set a few gpio pins:

```
IOSETx = (1 << BITPOSa) | (1 << BITPOSb) | ... | (1 << BITPOSn);
```

To set 8 contiguous gpio pins:

```
IOSETx = 0xFF << BITSTARTPOS;
```

To set 4 contiguous gpio pins:

```
IOSETx = 0x0F << BITSTARTPOS;
```

IOCLRx: To exclusively write logic '0' onto a gpio pin without affecting other pins

To clear a particular gpio pin:

```
IOCLRx = 1 << BITPOS;
```

To clear a few particular gpio pins:

```
IOCLRx = (1 << BITPOSa) | (1 << BITPOSb) | ... | (1 << BITPOSn);
```

To clear 8 contiguous pins:

```
IOCLRx = 0xFF << BITSTARTPOS;
```

To clear 4 contiguous pins:

```
IOCLRx = 0x0F << BITSTARTPOS;
```

IOPINx: To write logic i/o and read the status of pins

To write logic to a particular gpio pin without affecting other pins:

```
IOPINx |= 1 << BITPOS;
```

To write logic to a few gpio pins without affecting other pins:

```
IOPINx |= (1 << BITPOSa) | (1 << BITPOSb) | ... | (1 << BITPOSn);
```

To write any byte onto 8 contiguous pins without affecting other pins:

```
IOPINx |= (IOPINx & ~(0xFF << BITSTARTPOS)) | (BYTE << BITSTARTPOS);
```

To clear a gpio pin without affecting other pins:

```
IOPINx &= ~(1 << BITPOS);
```

To clear a few gpio pins without affecting other pins:

```
IOPINx &= ~(1 << BITPOSa | 1 << BITPOSb | ... | 1 << BITPOSn);
```

To clear 8 contiguous pins without affecting other pins:

```
IOPINx &= ~(0xFF << BITSTARTPOS);
```

To clear 4 contiguous pins without affecting other pins:

```
IOPINx &= ~(0x0F << BITSTARTPOS);
```

Reading the status of pins

To read the status of a gpio pin:

```
status = (IOPINx >> BITPOS) & 1;
```

To read the status of 4 contiguous pins:

```
status = (IOPINx >> BITSTARTPOS) & 0x0F;
```

To read the status of 8 contiguous pins:

```
status = (IOPINx >> BITSTARTPOS) & 0xFF;
```

LCD

An **Alphanumeric LCD** or **Character LCD** is a type of liquid crystal display designed to display alphanumeric characters (letters, numbers, and basic symbols). These LCDs are commonly used in various electronic devices for displaying text-based information. Here's a summary of key points about Alphanumeric/Character LCDs:

Key Features:

1. **Based on HD44780 Controller**:
   - The HD44780 is a widely used controller for alphanumeric LCDs, providing the necessary functionality to display characters and control the display.
   - It supports various LCD configurations, such as 16x1, 16x2, 20x2, 20x4, etc., where the first number represents the number of characters per line, and the second number represents the number of lines.
2. **Memory Organization**:
   - **CGROM (Character Generator ROM)**:
     - Stores pre-defined character patterns (240 characters in total).
     - Includes 208 characters for 5x8 dot format and 32 characters for 5x10 dot format.
   - **DDRAM (Display Data RAM)**:
     - Stores the characters to be displayed on the screen (up to 80 characters).
   - **CGRAM (Character Generator RAM)**:
     - Allows users to create custom characters (up to 8 custom 5x8 dot characters or 4 custom 5x10 dot characters).
3. **Display Capabilities**:
   - Displays ASCII characters.
   - Supports both standard and user-defined characters.
4. **Control Functions**:
   - **Cursor Control**:
     - Move cursor left/right, enable/disable cursor, and control cursor blinking.
   - **Display Control**:

        - Turn the display on/off, shift the display, and control the direction of cursor movement.
    - **Interface Settings**:
        - Configure data interface length (4-bit or 8-bit), number of display lines (1 or 2), and character font (5x7 or 5x10 dots).
5. **Busy Flag**:
    - A flag (BF) indicates when the LCD is busy processing a command, preventing further commands until it is ready.

Common Applications:

- **Coin Boxes**: Displaying transaction information.
- **Water Vending Machines**: Showing options and payment details.
- **Hotel Billing Machines**: Displaying billing information.
- **Industrial Equipment**: Providing status or diagnostic information.
- **Consumer Electronics**: Used in devices like calculators, microwaves, and printers.

Advantages:

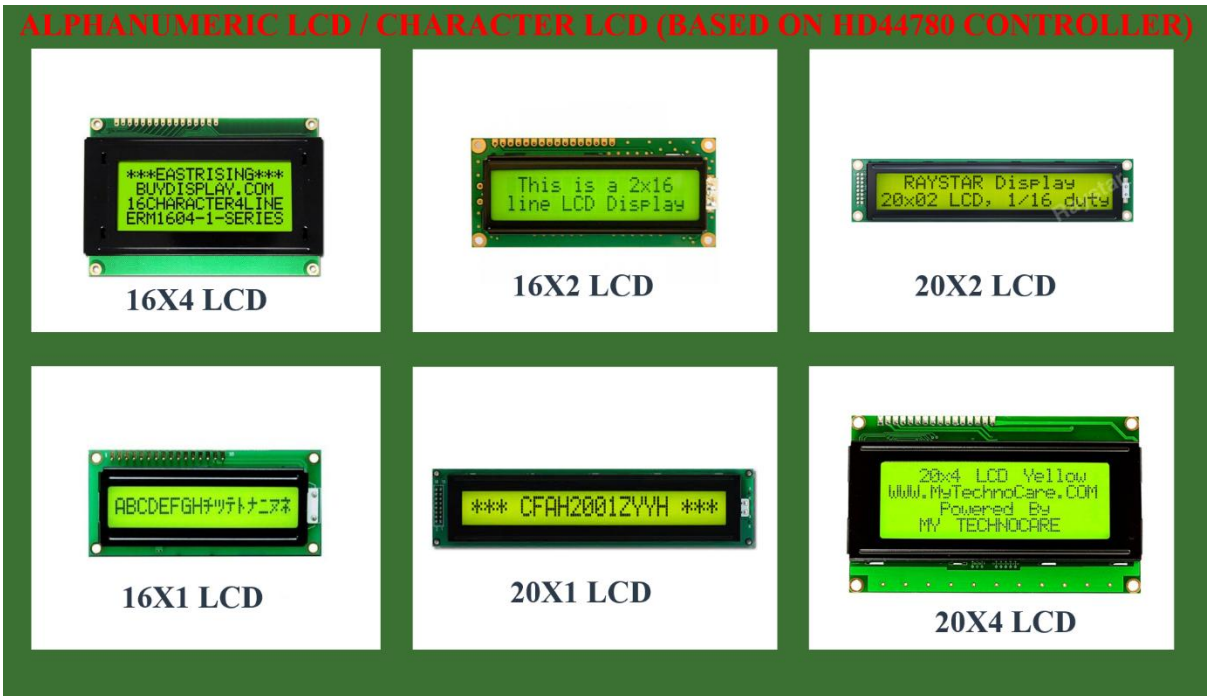
- Simple to interface with microcontrollers.
- Low power consumption.
- Cost-effective for displaying text-based information.
- Customizable with user-defined characters.

Limitations:

- Limited to displaying alphanumeric characters and basic symbols.
- Limited resolution (typically 5x7 or 5x10 dots per character).
- Not suitable for displaying complex graphics or images.

Types of Alphanumeric LCDs (Based on HD44780 Controller)

Alphanumeric LCDs are categorized based on their display format, commonly defined by **columns (characters) x rows (lines)**.



---

1. 16x1 LCD

- **Display Format:** 16 characters in a single row.
- **Usage:** Suitable for compact devices where minimal information is displayed at a time (e.g., small appliances, measurement tools).

---

2. 16x2 LCD

- **Display Format:** 16 characters per row × 2 rows.
- **Usage:** Commonly used in embedded systems, Arduino projects, and digital clocks due to its balance between size and information display.

---

3. 16x4 LCD

- **Display Format:** 16 characters per row × 4 rows.
- **Usage:** Suitable for more extensive information displays like industrial equipment or electronic devices requiring more content visibility.

---

4. 20x1 LCD

- **Display Format:** 20 characters in a single row.
- **Usage:** Often seen in compact systems that require slightly more information than a standard 16x1 LCD.

---

5. 20x2 LCD

- **Display Format:** 20 characters per row × 2 rows.
- **Usage:** Frequently used in communication systems, vending machines, and control panels for displaying concise yet effective information.

---

6. 20x4 LCD

- **Display Format:** 20 characters per row × 4 rows.
- **Usage:** Ideal for systems requiring multiple lines of detailed information such as cash counters, ATMs, and medical equipment displays.

Common Features Across All Types:

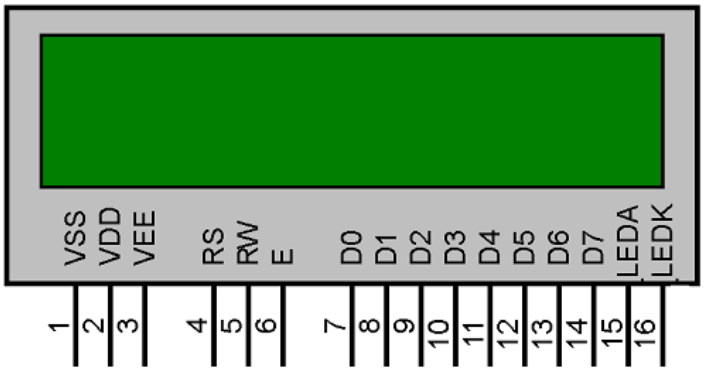
☑ Uses **HD44780 Controller** for easy interfacing.

☑ Supports **4-bit** or **8-bit** data communication modes.

☑ Displays ASCII characters and allows custom character creation.

☑ Suitable for microcontroller interfacing like **8051**, **AVR**, **ARM**, and **Arduino**.

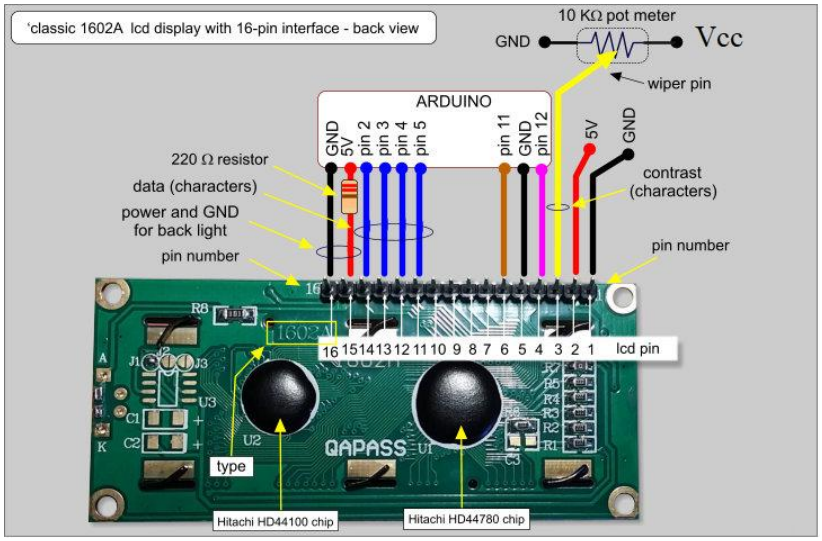
16x2 LCD Module Pinouts:



Sr. No	Pin No.	Pin Name	Pin Type	Pin Description	Pin Connection
1	Pin 1	**VSS (Ground)**	Source Pin	Ground pin of the LCD.	Connected to the ground of the MCU (Microcontroller Unit) or power source.
2	Pin 2	**VCC**	Source Pin	Supply voltage pin for the LCD (typically +5V).	Connected to the +5V supply pin of the power source.
3	Pin 3	**V0/VEE**	Control Pin	Adjusts the contrast of the LCD display.	Connected to a variable potentiometer (POT) that can source 0-5V.

4	Pin 4	**RS**	Control Pin	Register Select: Toggles between Command and Data registers.	Connected to an MCU pin. 0 = Command Mode, 1 = Data Mode.
5	Pin 5	**RW**	Control Pin	Read/Write: Toggles between Read and Write operations.	Connected to an MCU pin. 0 = Write Operation, 1 = Read Operation.
6	Pin 6	**E**	Control Pin	Enable: Must be held high to perform Read/Write operations.	Connected to the MCU and always held high during operations.
7	Pin 7-14	**D0-D7**	Data/Command Pin	Data pins used to send commands or data to the LCD.	In **4-Wire Mode**: Only pins D4-D7 are connected to the MCU.
					In **8-Wire Mode**: All 8 pins (D0-D7) are connected to the MCU.
8	Pin 15	**LED+ (LEDA)**	LED Pin	Positive pin for the backlight LED (illuminates the LCD).	Connected to +5V.
9	Pin 16	**LED- (LEDK)**	LED Pin	Negative pin for the backlight LED (connected to ground).	Connected to ground.

Key Functions of the Pins:



1. **VSS (Pin 1)**:
    - This is the **ground pin** of the LCD. It must be connected to the ground of the power supply or microcontroller.
2. **VCC (Pin 2)**:
    - This is the **power supply pin** for the LCD. It typically requires +5V to operate.
3. **V0/VEE (Pin 3)**:
    - This pin controls the **contrast** of the LCD display. It is usually connected to a potentiometer to adjust the voltage between 0V and 5V, which changes the contrast.
4. **RS (Pin 4)**:
    - The **Register Select** pin determines whether the data being sent to the LCD is a **command** (e.g., clear screen, move cursor) or **data** (e.g., characters to display).
    - **0 = Command Mode**, **1 = Data Mode**.
5. **RW (Pin 5)**:
    - The **Read/Write** pin determines whether the LCD is in **read mode** (reading data) or **write mode** (sending data).
    - **0 = Write Mode**, **1 = Read Mode**.
6. **E (Pin 6)**:
    - The **Enable** pin is used to enable the LCD for reading or writing. It must be held high during operations.
7. **D0-D7 (Pins 7-14)**:
    - These are the **data pins** used to send commands or data to the LCD.
    - In **4-Wire Mode**, only pins D4-D7 are used, while in **8-Wire Mode**, all 8 pins (D0-D7) are used.
8. **LED+ (Pin 15)** and **LED- (Pin 16)**:
    - These pins control the **backlight LED** of the LCD.
    - **LED+** is connected to +5V, and **LED-** is connected to ground to illuminate the display.

Modes of Operation:

- **4-Wire Mode**: Only 4 data pins (D4-D7) are used to communicate with the LCD. This mode saves microcontroller pins but is slower.
- **8-Wire Mode**: All 8 data pins (D0-D7) are used for faster communication with the LCD.

Applications:

- **16x2 LCDs** are widely used in projects like:
    - Temperature displays.
    - Digital clocks.
    - Home automation systems.
    - Industrial control panels.

- Embedded systems for displaying status information.

HD44780 Memory Organization Overview

1. ROM – CGROM (Character Generator ROM)

- **Purpose:** Stores a **fixed set of predefined characters** (such as letters, numbers, symbols).
- **Size: 9920 bits** total.
- **Character Capacity:**
  - **208 characters** for **5x8 dot matrix**.
  - **32 characters** for **5x10 dot matrix**.
- **Fixed Characters:** Includes English letters (A-Z), numbers (0-9), special symbols, and commonly used characters.

☑ **CGROM data is read-only** and cannot be modified.

---

2. RAM – DDRAM (Display Data RAM)

- **Purpose:** Stores the **actual characters** displayed on the LCD.
- **Size: 80 bytes** total.
- **Character Capacity:**
  - Each byte corresponds to **one character**.
  - Supports a maximum of **80 characters** on the display.
- **Addressing:**
  - For **16x2 LCD**:
    - First row starts at **0x00** to **0x0F**.
    - Second row starts at **0x40** to **0x4F**.

☑ Data written to DDRAM is what appears on the screen.

---

3. RAM – CGRAM (Character Generator RAM)

- **Purpose:** Used to create **custom characters** defined by the user.
- **Size: 64 bytes** total.
- **Character Capacity:**
  - **8 custom characters** for **5x8 dot matrix** (each character takes **8 bytes**).
  - **4 custom characters** for **5x10 dot matrix** (each character takes **16 bytes**).

☑ CGRAM allows developers to create custom symbols like icons, logos, or special patterns.

Memory Allocation Summary Table

Memory Type	Purpose	Size	Character Capacity	Customizable
CGROM	Predefined characters	**9920 bits**	**240 characters** (208 in 5x8, 32 in 5x10)	✕ (Read-only)

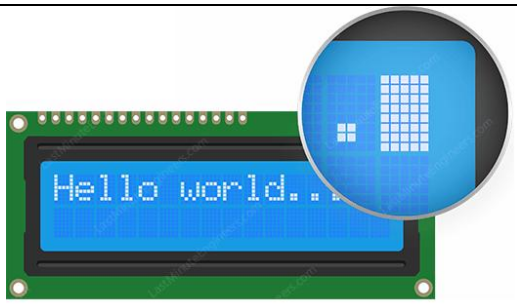
DDRAM	Display data storage	**80 bytes**	**Up to 80 characters**	☑ (Write data to display)
CGRAM	Custom character storage	**64 bytes**	**8 characters (5x8)** or **4 characters (5x10)**	☑ (User-defined patterns)

Example Usage Scenarios

☑ Use **CGROM** for displaying standard ASCII characters like text, numbers, and common symbols.

☑ Use **DDRAM** to send text content to be displayed on the screen.

☑ Use **CGRAM** to create special icons like arrows, battery symbols, or smiley faces.



1. Display Overview

- The text displayed on the LCD reads **"Hello world..."** — this is a common test message used to confirm successful LCD interfacing.
- The display appears to be backlit with a **blue screen** and **white characters**, which is typical for HD44780-based LCDs.

---

2. Pixel Structure (Zoomed-in Section)

The magnified section reveals the **dot matrix pattern** used to form characters.

- Each character in the LCD is formed using a grid of **5x8 dots** (5 columns × 8 rows) or **5x10 dots** in some configurations.
- The zoomed-in pixel structure shows the exact arrangement of dots that shapes each character.

---

3. How Characters Are Formed

- Each dot is an individual pixel controlled by the **DDRAM (Display Data RAM)** or **CGRAM (Character Generator RAM)**.
- Characters are mapped as a combination of **ON** (illuminated pixel) and **OFF** (non-illuminated pixel) states.
- For example:
    - The letter **'H'** is formed by illuminating the appropriate pixels in the 5x8 matrix.

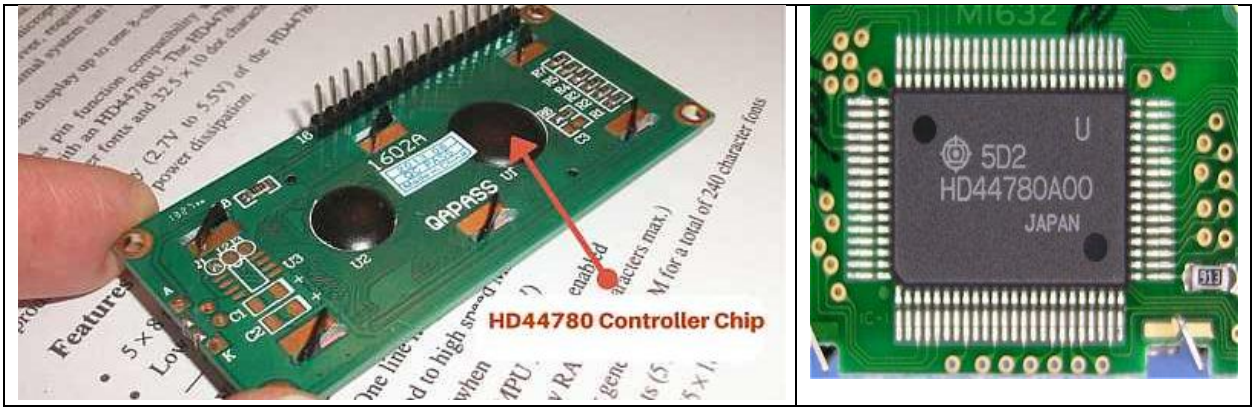
- The **block-like icon** shown in the zoomed view is likely a **user-defined character** stored in **CGRAM**.

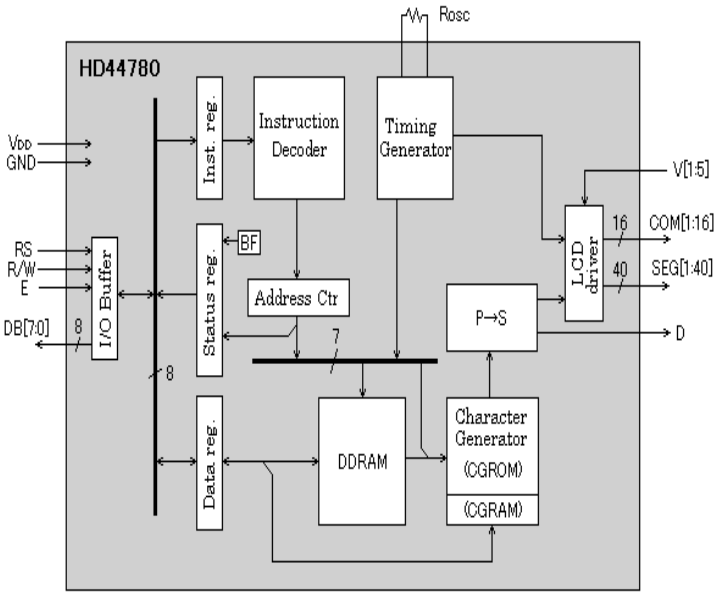
---

4. Important Points

- ☑ **DDRAM** stores ASCII codes for standard text characters.
- ☑ **CGRAM** allows you to create custom icons or symbols by manually defining pixel patterns.
- ☑ Each column in the grid corresponds to one data bit when sending information to the LCD.

HD44780 LCD Controller Architecture Explanation





The **HD44780** is a commonly used LCD controller that manages alphanumeric displays like **16x2**, **20x4**, etc. This architecture is designed to control the LCD's internal functions, character display, and custom symbols. Let's break down the key components and their roles.

Key Components and Explanation

1. Power Supply (VDD & GND)

- **VDD**: Provides the required **+5V** power to the controller.
- **GND**: Ground connection for the circuit.

---

2. Control Pins (RS, R/W, E)

These pins control data flow and command execution:

- **RS (Register Select):**
  - **0** → Command register (for sending commands like clear display, cursor control).
  - **1** → Data register (for sending actual data/characters).
- **R/W (Read/Write):**
  - **0** → Write data/command to LCD.
  - **1** → Read status (e.g., busy flag).
- **E (Enable):** Acts as a trigger. Data is processed on the **falling edge** of this signal.

---

3. Data Bus (DB[7:0])

- **DB0 - DB7:** These **8-bit data lines** transmit data or commands to the LCD.
- The controller can operate in either:
  - **8-bit mode** (all 8 data lines used).
  - **4-bit mode** (only DB4 to DB7 used, reducing the number of I/O pins needed).

---

4. I/O Buffer

- Temporarily holds data to manage synchronization between external signals and internal data processing.
- Acts as a bridge between the data bus and internal registers.

---

5. Instruction Register (Inst. reg.)

- Stores command instructions like:
  - Clear Display
  - Cursor Control
  - Display ON/OFF
  - Shift Operations
- Data written to this register is decoded and executed by the **Instruction Decoder**.

---

6. Timing Generator

- Manages internal timing sequences for LCD operations.
- Ensures proper signal timing for stable data transfer.

---

7. Status Register

- Contains the **Busy Flag (BF)**:

    - **BF = 1** → LCD is busy and cannot accept new instructions.
    - **BF = 0** → LCD is ready for new commands/data.
- Monitors display conditions and error states.

---

8. Address Counter (Address Ctr)

- Holds the address for:
    - **DDRAM** (Display Data RAM)
    - **CGRAM** (Character Generator RAM)
- Ensures the correct data is fetched or written at the required memory address.

---

9. DDRAM (Display Data RAM)

- Stores character data that appears on the LCD screen.
- Can hold up to **80 characters** in total.

---

10. CGROM (Character Generator ROM)

- Stores **fixed character patterns** for common symbols, numbers, and letters.
- Provides **208 characters** for **5x8 dots** or **32 characters** for **5x10 dots**.

---

11. CGRAM (Character Generator RAM)

- Used to define **custom characters**.
- Supports **8 custom 5x8 characters** or **4 custom 5x10 characters**.

---

12. Parallel to Serial Converter (P→S)

- Converts parallel data from DDRAM/CGROM/CGRAM into a serialized format suitable for driving the LCD segments.

---

13. LCD Driver

- Controls the physical pixels on the display.
- Manages:
    - **COM[1:16]**: Controls common lines for the LCD rows.
    - **SEG[1:40]**: Controls segment lines for the LCD columns.

Working Flow

1. **Data Input:**
    - Data or commands are sent via **DB[7:0]**.
2. **Instruction Processing:**
    - The **Instruction Decoder** interprets commands and activates the necessary functions.
3. **Memory Management:**
    - The **Address Counter** tracks DDRAM or CGRAM data locations.

4. **Pixel Control:**
   - The **LCD Driver** activates the necessary pixels based on data retrieved from DDRAM/CGROM/CGRAM.

Summary Table

Component	Function
I/O Buffer	Temporary data storage for synchronization
Instruction Register	Stores incoming instructions
Timing Generator	Generates timing signals for internal operations
Status Register	Tracks LCD status (Busy Flag)
Address Counter	Manages address locations in DDRAM/CGRAM
DDRAM	Stores displayed characters
CGROM	Predefined character library
CGRAM	Custom character storage
P→S Converter	Converts parallel data to serial format for pixel control
LCD Driver	Drives the LCD's rows (COM) and columns (SEG)

Key Points for Developers

- ☑ Use **RS, RW, and E** pins effectively to control data flow.
- ☑ Monitor the **Busy Flag (BF)** before sending new instructions.
- ☑ For custom characters, define patterns in **CGRAM** and load them to DDRAM.
- ☑ Choose between **4-bit mode** or **8-bit mode** based on your microcontroller's I/O limitations.

HD44780 Character Generator ROM (CGROM) Table

This table represents the **Character Generator ROM (CGROM)** of the **HD44780 LCD Controller**. The CGROM stores predefined character patterns that are displayed when corresponding codes are sent to the LCD. Let's break down the table and understand how to access specific characters.

Understanding the Table Structure

The table is divided into two parts:

- **Upper 4 bits** → Column values (0000 to 1111)
- **Lower 4 bits** → Row values (xxxx0000 to xxxx1111)

Character Code Format

- Each character is accessed using an **8-bit code**.
- The **upper 4 bits** define the **column**.
- The **lower 4 bits** define the **row**.

Example of Character Encoding

Suppose you want to display the letter **'A'**:

- Upper 4 bits → 0100
- Lower 4 bits → 0001
- **Character Code = 0x41 (in Hex) or 65 (in Decimal)**

Table Breakdown

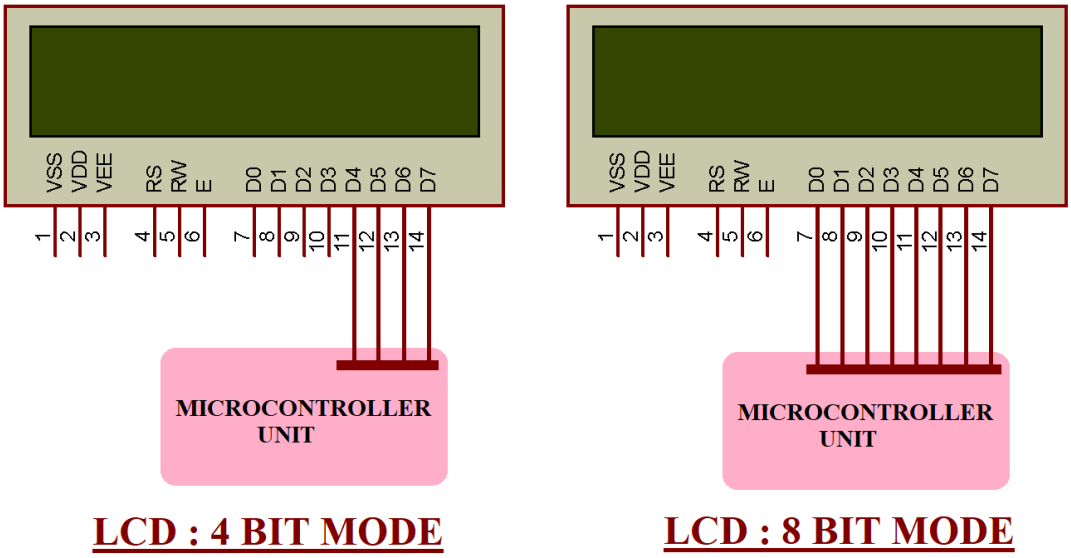
Lower 4 Bits	Upper 4 Bits (0000-1111)	Character Example
xxxx0000	0 → @ , P , , ~	Special characters, capital letters
xxxx0001	1 → ! , 1 , A , a	Numbers, uppercase, lowercase
xxxx0010	2 → " , 2 , B , b	Special symbols, characters
xxxx0011	3 → # , 3 , C , c	Special symbols, characters
xxxx0100	4 → $, 4 , D , d	Currency, numbers, letters
xxxx0101	5 → % , 5 , E , e	Special characters, letters
xxxx0110	6 → & , 6 , F , f	Special symbols, letters
xxxx0111	7 → ' , 7 , G , g	Apostrophe, numbers, letters
xxxx1000	8 → (, 8 , H , h	Parenthesis, letters
xxxx1001	9 →) , 9 , I , i	Parenthesis, numbers, letters
xxxx1010	A → * , : , J , j	Special symbols, letters
xxxx1011	B → + , ; , K , k	Arithmetic operators, letters
xxxx1100	C → , , < , L , l	Symbols, comparison operators

xxxx1101	D → - , = , M , m	Arithmetic symbols, letters
xxxx1110	E → . , > , N , n	Period, greater than, letters
xxxx1111	F → / , ? , O , o	Slash, question mark, letters

Key Points

- ☑ Each character is stored as a **5x8 pixel** or **5x10 pixel** dot matrix.
- ☑ Characters are stored in **CGROM**, and user-defined characters are stored in **CGRAM**.
- ☑ Custom characters are defined using **8-byte data sets** (for 5x8 mode).
- ☑ For special characters (e.g., smiley faces, arrows), you can define them in **CGRAM**.

LCD Interfacing in 4-bit Mode vs 8-bit Mode (Detailed Explanation)



LCD : 4 BIT MODE **LCD : 8 BIT MODE**

The diagram shows two different configurations for interfacing an **HD44780 LCD** with a microcontroller:

1. **4-bit Mode**
2. **8-bit Mode**

Both configurations have distinct wiring methods, each with its advantages and disadvantages.

1. 4-bit Mode (Left Side of the Diagram)

In **4-bit mode**, only **4 data pins (D4 to D7)** are connected to the microcontroller, while the remaining data pins (D0 to D3) are left unconnected.

Connections in 4-bit Mode

- **VSS (Pin 1)** → GND
- **VDD (Pin 2)** → VCC (+5V)
- **VEE (Pin 3)** → Potentiometer Wiper (For Contrast Control)
- **RS (Pin 4)** → Connected to MCU control pin (for Register Select)
- **RW (Pin 5)** → GND (For Write Mode Only)
- **E (Pin 6)** → Connected to MCU control pin (For Enable Signal)
- **D4 to D7 (Pins 11 to 14)** → Connected to MCU data pins
- **D0 to D3 (Pins 7 to 10)** → **Not connected** (ignored in 4-bit mode)

How 4-bit Mode Works

- Since only 4 data lines are used, **each byte of data is divided into two 4-bit nibbles** and sent sequentially.
- The LCD receives the **higher nibble** (upper 4 bits) first, followed by the **lower nibble** (lower 4 bits).
- Control signals like **RS**, **RW**, and **E** are still used in the same way as in 8-bit mode.

Advantages of 4-bit Mode

☑ Uses fewer MCU I/O pins (ideal for pin-limited MCUs like Arduino Nano or ESP8266).
☑ Easier wiring with reduced complexity.
☑ Requires only **6 data lines total** (4 data + RS + E).

Disadvantages of 4-bit Mode

❗ Data transmission is slower since each byte requires **two cycles**.
❗ More programming effort due to nibble-splitting logic.

---

2. 8-bit Mode (Right Side of the Diagram)

In **8-bit mode**, **all 8 data pins (D0 to D7)** are connected to the microcontroller.

Connections in 8-bit Mode

- **VSS (Pin 1)** → GND
- **VDD (Pin 2)** → VCC (+5V)
- **VEE (Pin 3)** → Potentiometer Wiper (For Contrast Control)
- **RS (Pin 4)** → Connected to MCU control pin (for Register Select)
- **RW (Pin 5)** → GND (For Write Mode Only)
- **E (Pin 6)** → Connected to MCU control pin (For Enable Signal)
- **D0 to D7 (Pins 7 to 14)** → Connected to MCU data pins

How 8-bit Mode Works

- In 8-bit mode, the LCD receives a **full 8-bit data** (one complete byte) in a single cycle.

- Control signals like **RS**, **RW**, and **E** operate similarly to 4-bit mode.

Advantages of 8-bit Mode

☑ Faster data transmission since each byte is sent in **one cycle**.

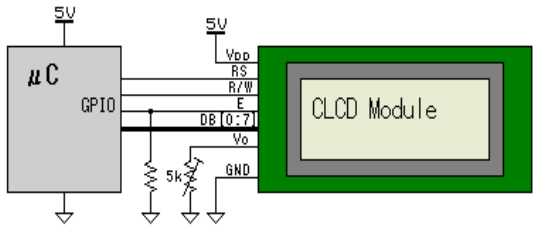
☑ Simpler coding since there's no need to split data into nibbles.

Disadvantages of 8-bit Mode

❗ Requires **more I/O pins** (total 10 lines including control pins).

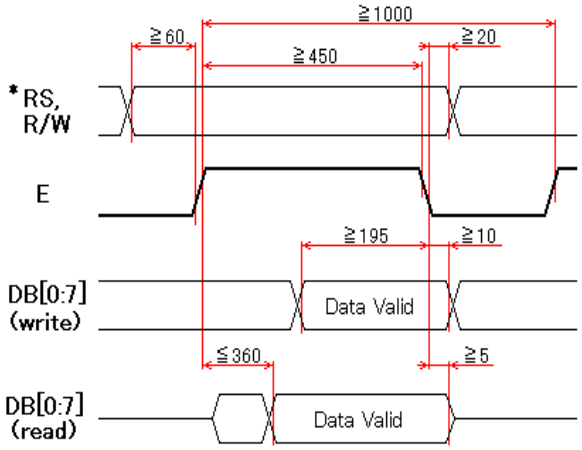
❗ More wiring complexity.

(a) Typical Connection between uC and LCD



Pin#	Name	Function	
1	GND	Power Supply	
2	VDD		
3	Vo	Contrast	
4	RS	Register Select	
5	R/W	Read/Write	
6	E	Enable	
7	DB0	Data Bus (8-bit mode)	N.C. (4-bit mode)
8	DB1		
9	DB2		
10	DB3		
11	DB4		Data Bus (4-bit mode)
12	DB5		
13	DB6		
14	DB7		

(b) Timing Diagram (1MHz)



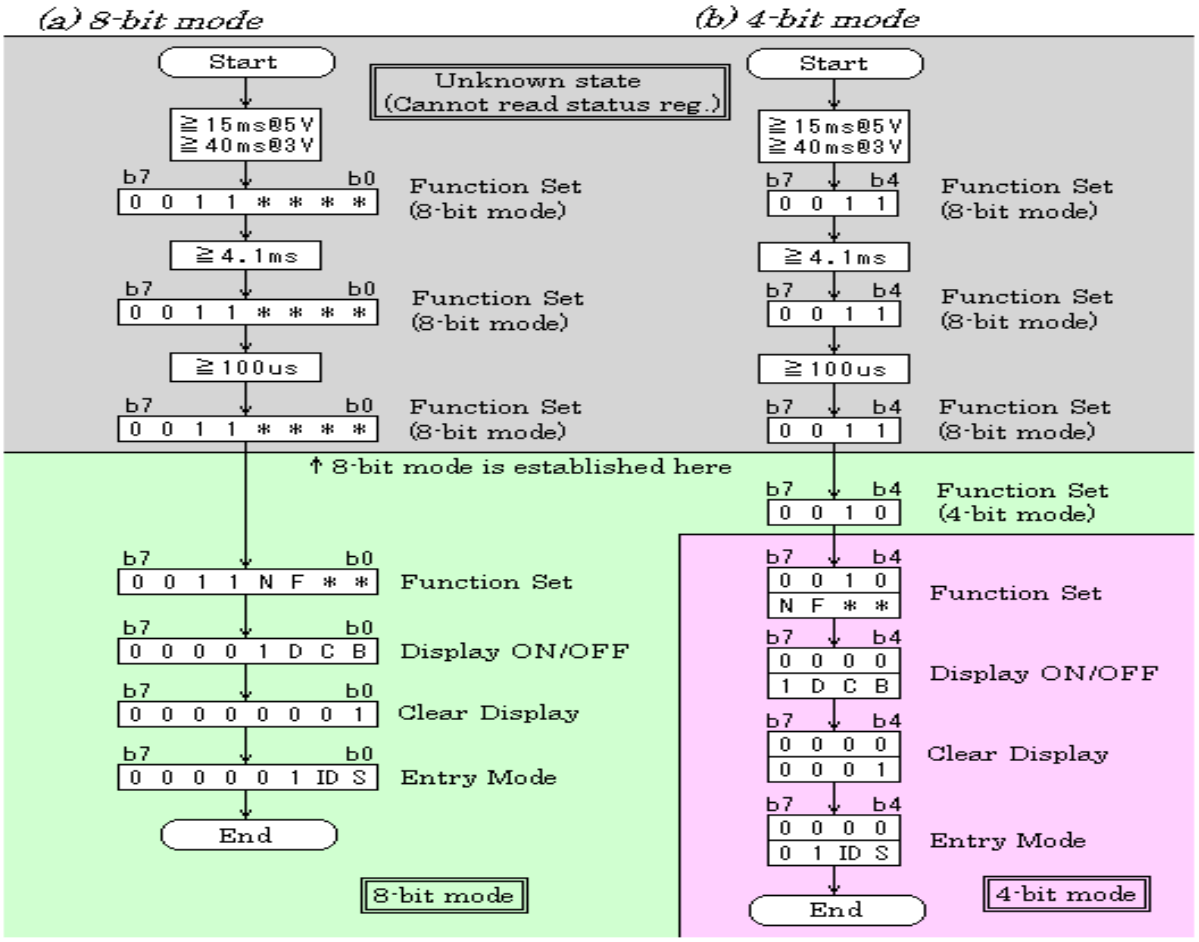
All units are nanosecond

***Truth Table**

RS	R/W	Function
L	L	Write Command Register
H	L	Write Data Register
L	H	Read Status Register
H	H	Read Data Register

This diagram illustrates the following concepts:

1. **Typical Connection between Microcontroller (μC) and LCD (Left Side)**
2. **Timing Diagram for Data Transfer (Right Side)**
3. **Pin Description Table**
4. **Truth Table for Control Signals**



---

1. Typical Connection between Microcontroller (µC) and LCD

The left portion of the diagram shows the physical wiring between the microcontroller and an LCD module.

Pin Descriptions and Connections

Pin No.	Name	Function
1	**GND**	Ground (0V)
2	**VDD**	Power Supply (+5V)
3	**Vo**	Contrast Control (Connected to 5kΩ potentiometer)
4	**RS**	Register Select (Command/Data selection)
5	**R/W**	Read/Write Control (0 = Write, 1 = Read)
6	**E**	Enable Pin (Triggers data read/write)
7-14	**DB0 - DB7**	Data Bus (8-bit mode) / Only D4-D7 used in 4-bit mode

Explanation of Key Connections

- **Power Supply:**
  - **VDD** → +5V Power
  - **GND** → Ground
- **Contrast Control:**

- **Vo** is connected to a **5kΩ potentiometer** to adjust LCD contrast.

- **Control Pins:**
  - **RS** (Register Select):
    - **0** → Command Register (Control Instructions like Clear Display).
    - **1** → Data Register (Display Characters).
  - **R/W** (Read/Write):
    - **0** → Write Operation.
    - **1** → Read Operation.
  - **E** (Enable Pin):
    - Acts as a **trigger** for data transmission. A **high-to-low** pulse triggers data processing.
- **Data Pins:**
  - **DB0 - DB7** → Used in **8-bit mode**.
  - **DB4 - DB7** → Used in **4-bit mode**.

---

2. Timing Diagram (Right Side)

The timing diagram shows the critical timing requirements for the LCD to process data correctly.

Key Timing Parameters

- **RS, R/W Setup Time:** ≥ 60 ns (before Enable signal goes HIGH).
- **Enable Pulse Width (High):** ≥ 450 ns (for stable data transfer).
- **Data Hold Time (after Enable LOW):** ≥ 10 ns.
- **Enable Pulse Width (Low):** ≥ 360 ns (ensures data is latched properly).

Write Operation (DB[0:7] Write)

- Data is placed on the data bus.
- **RS** and **R/W** are set as required.
- The **Enable (E)** pin goes HIGH, then LOW to latch the data.

Read Operation (DB[0:7] Read)

- Data is placed on the bus by the LCD.
- The microcontroller reads data while Enable remains HIGH.
- Data becomes valid after a defined delay.

---

3. Truth Table (Control Logic)

The truth table specifies the conditions for different LCD operations.

RS	R/W	Function
0	**0**	Write Command (e.g., Clear Display)
0	**1**	Read Status Register (e.g., Busy Flag)
1	**0**	Write Data (Display Characters)
1	**1**	Read Data (Retrieve Displayed Data)

Explanation

- **Command Mode:**
  - RS = **0**, R/W = **0** → Write commands like cursor position or clear screen.
- **Data Mode:**
  - RS = **1**, R/W = **0** → Write data to display characters.
- **Read Operations:**
  - RS = **0**, R/W = **1** → Read status register (e.g., busy flag).
  - RS = **1**, R/W = **1** → Read data from DDRAM or CGRAM.

---

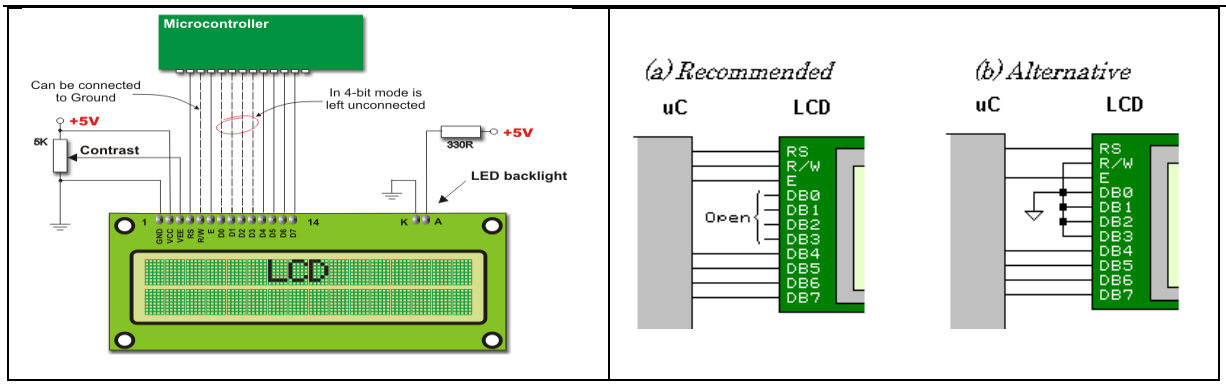
4. Key Points to Remember

☑ **Timing precision** is critical — improper delays will cause data corruption.

☑ Always configure the potentiometer properly for optimal **contrast control**.

☑ Use the **Enable (E)** pin as a trigger mechanism for both **write** and **read** operations.

☑ Ensure your **RS** and **R/W** settings are correct based on whether you're sending commands or data.



1. First Diagram - LCD Interfacing with Microcontroller

This diagram shows a detailed wiring setup for connecting an LCD with a microcontroller.

Key Components:

- **Microcontroller:** Generates signals and controls the LCD.
- **LCD Module:** Displays characters based on instructions from the microcontroller.
- **5K Potentiometer:** Used for adjusting LCD contrast.
- **330Ω Resistor:** Limits the current for the LED backlight.
- **Power Supply (5V):** Powers both the LCD and microcontroller.

Connections in Detail:

- **Power Supply Connections:**
  - **Pin 1 (GND)** → Ground
  - **Pin 2 (VCC)** → +5V

- **Contrast Control (Pin 3 - VEE):** Connected to a **5K potentiometer** with one end at +5V and the other at GND. The wiper pin adjusts the display contrast.
- **Control Pins:**
  - **RS (Pin 4):** Register Select (Command/Data)
  - **R/W (Pin 5):** Read/Write control
  - **E (Pin 6):** Enable signal to trigger data transfer
- **Data Pins (D0 to D7):**
  - In **8-bit mode**, all 8 data pins (D0 to D7) are connected.
  - In **4-bit mode**, only **D4 to D7** are connected, and **D0 to D3** are left unconnected.

LED Backlight Control:

- **Pin 15 (A - Anode)** → +5V through **330Ω resistor** for current limitation.
- **Pin 16 (K - Cathode)** → Connected to **GND**.

---

2. Second Diagram - Recommended vs Alternative Wiring

This diagram highlights two different wiring methods for unused data pins in **4-bit mode**:

(a) Recommended Wiring (Left Side)

- **D0 to D3** pins are **left unconnected** (open).
- This method is preferred because the internal pull-down resistors in the LCD keep these unused data lines stable.

(b) Alternative Wiring (Right Side)

- Unused data pins **D0 to D3** are connected to **GND** via resistors.
- This method is also acceptable to prevent floating inputs, but the **recommended method (left)** is simpler and widely used.

---

3. Key Differences:

Feature	**4-Bit Mode**	**8-Bit Mode**
Data Lines Used	Only **D4 to D7** connected	All **D0 to D7** connected
Pins Required	Fewer GPIO pins needed	More GPIO pins required
Data Transmission	Data sent in **nibbles** (half-byte)	Data sent in **one-byte**
Speed	**Slower** due to data splitting	**Faster** due to full-byte data transfer

---

HD44780 Instruction Set Explanation

The HD44780 instruction set is shown below:

R/S	R/W	D7	D6	D5	D4	D3	D2	D1	D0	Instruction/Description
4	5	14	13	12	11	10	9	8	7	Pins
0	0	0	0	0	0	0	0	0	1	Clear Display
0	0	0	0	0	0	0	0	1	*	Return Cursor and LCD to Home Position
0	0	0	0	0	0	0	1	ID	S	Set Cursor Move Direction
0	0	0	0	0	0	1	D	C	B	Enable Display/Cursor
0	0	0	0	0	1	SC	RL	*	*	Move Cursor/Shift Display
0	0	0	0	1	DL	N	F	*	*	Set Interface Length
0	0	0	1	A	A	A	A	A	A	Move Cursor into CGRAM
0	0	1	A	A	A	A	A	A	A	Move Cursor to Display
0	1	BF	*	*	*	*	*	*	*	Poll the "Busy Flag"
1	0	D	D	D	D	D	D	D	D	Write a Character to the Display at the Current Cursor Position
1	1	D	D	D	D	D	D	D	D	Read the Character on the Display at the Current Cursor Position

The bit descriptions for the different commands are:

"*" - Not Used/Ignored. This bit can be either "1" or "0"

The **HD44780 LCD controller** follows a specific instruction set for performing various tasks like clearing the display, setting the cursor position, and displaying characters. Let's break down the table for better understanding.

---

1. Instruction Table Overview

- **R/S (Register Select):**
  - **0** → Instruction Register (Commands)
  - **1** → Data Register (Character Data)
- **R/W (Read/Write):**
  - **0** → Write Operation
  - **1** → Read Operation
- **D0 to D7 (Data Pins):** Used to send command or data values to the LCD.

---

2. Command Details

Instruction	Description	Bit Details
Clear Display	Clears the display and sets the cursor to the home position (top-left corner).	0000 0001
Return Home	Sets the cursor to the home position without clearing the display content.	0000 001*

Set Cursor Move Direction	Sets cursor movement direction (increment or decrement) and enables display shift.	0000 01ID
Enable Display/Cursor	Controls display ON/OFF, cursor visibility, and cursor blinking.	0000 1DCB
Move Cursor/Shift Display	Shifts the entire display or moves the cursor without modifying text.	0001 SC RL
Set Interface Length	Sets data interface (4-bit or 8-bit mode), display line, and font type.	001 DL N F
Move Cursor into CGRAM	Allows writing custom characters to the LCD.	01AAAAAA
Move Cursor to Display	Sets the cursor's position on the display.	1AAAAAAA
Poll the Busy Flag	Checks if the LCD is ready for the next instruction.	BF******
Write Character	Writes a character at the current cursor position.	1DDDDDDD
Read Character	Reads the character at the current cursor position.	11DDDDDD

---

3. Key Bit Definitions

Bit	**Description**
ID (Increment/Decrement)	1 → Increment cursor position (right shift) 0 → Decrement cursor position (left shift)
S (Shift Display)	1 → Shift display when writing data 0 → No display shift
D (Display Control)	1 → Display ON 0 → Display OFF
C (Cursor Control)	1 → Cursor ON 0 → Cursor OFF
B (Blink Control)	1 → Cursor Blinking 0 → Cursor Steady
SC (Shift Control)	1 → Shift display 0 → Move cursor only
RL (Right/Left)	1 → Shift display or cursor to the **right** 0 → Shift display or cursor to the **left**
DL (Data Length)	1 → **8-bit mode** 0 → **4-bit mode**
N (Number of Lines)	1 → **2 lines** 0 → **1 line**
F (Font Type)	1 → 5x10 dots 0 → 5x8 dots
A (Address Bits)	Used for cursor positioning in **CGRAM** or **DDRAM**

BF (Busy Flag)	1 → LCD is busy (wait before sending next command) 0 → LCD is ready

---

4. Important Instructions for Practical Use

LCD Initialization (4-bit Mode Example)

1. **Function Set**: 0010 1000 (4-bit mode, 2-line display, 5x8 font)
2. **Display ON/OFF Control**: 0000 1100 (Display ON, Cursor OFF, Blink OFF)
3. **Entry Mode Set**: 0000 0110 (Increment cursor, no shift)
4. **Clear Display**: 0000 0001 (Clears display and sets cursor to the start)

---

5. Key Tips for Coding

☑ Always **check the busy flag** before sending the next command.

☑ For **4-bit mode**, send the command in **two nibbles** (upper 4 bits first, then lower 4 bits).

☑ Use the **clear display** command at the beginning to ensure the display is clean.

Detailed Explanation of HD44780 LCD Commands

Let's break down each instruction with clarity and practical examples to help you implement them effectively.

```
Set Cursor Move Direction:
        ID - Increment the Cursor after Each Byte Written to Display if Set
        S - Shift Display when Byte Written to Display
Enable Display/Cursor
        D - Turn Display On(1)/Off(0)
        C - Turn Cursor On(1)/Off(0)
        B - Cursor Blink On(1)/Off(0)
Move Cursor/Shift Display
        SC - Display Shift On(1)/Off(0)
        RL - Direction of Shift Right(1)/Left(0)
Set Interface Length
        DL - Set Data Interface Length 8(1)/4(0)
        N - Number of Display Lines 1(0)/2(1)
        F - Character Font 5x10(1)/5x7(0)
Poll the "Busy Flag"
        BF - This bit is set while the LCD is processing
Move Cursor to CGRAM/Display
        A - Address
Read/Write ASCII to the Display
        D - Data
```

1. Set Cursor Move Direction

- **ID (Increment Direction):**
  - **1** → Cursor moves **right** after each character is written (default).
  - **0** → Cursor moves **left** after each character is written.
- **S (Shift Display):**

- **1** → Display shifts when a new character is written.
- **0** → Display remains stationary.

Example Command:

- 0b00000110 → Increment cursor position without shifting display.
- 0b00000111 → Increment cursor position **and** shift display.

---

2. Enable Display/Cursor

- **D (Display Control):**
  - **1** → Display **ON**.
  - **0** → Display **OFF**.
- **C (Cursor Control):**
  - **1** → Cursor **ON**.
  - **0** → Cursor **OFF**.
- **B (Blink Control):**
  - **1** → Cursor **blinks**.
  - **0** → Cursor does **not blink**.

Example Commands:

- 0b00001100 → Display ON, Cursor OFF, Blink OFF.
- 0b00001111 → Display ON, Cursor ON, Blink ON.
- 0b00001000 → Display OFF.

---

3. Move Cursor/Shift Display

- **SC (Shift Control):**
  - **1** → Shift the **display**.
  - **0** → Shift **cursor only**.
- **RL (Right/Left):**
  - **1** → Shift **right**.
  - **0** → Shift **left**.

Example Commands:

- 0b00011000 → Shift cursor **left**.
- 0b00011100 → Shift cursor **right**.
- 0b00010100 → Shift display **left**.
- 0b00010110 → Shift display **right**.

---

4. Set Interface Length

- **DL (Data Length):**
  - **1** → Use **8-bit** data interface.
  - **0** → Use **4-bit** data interface.
- **N (Number of Display Lines):**
  - **1** → Use **2 lines** (commonly used).

    - **0** → Use **1 line**.
- **F (Font Style):**
    - **1** → **5x10** dot character font.
    - **0** → **5x8** dot character font (default).

Example Commands:

- 0b00111000 → **8-bit mode**, 2 lines, 5x8 font.
- 0b00101000 → **4-bit mode**, 2 lines, 5x8 font.

---

5. Poll the "Busy Flag"

- **BF (Busy Flag):**
    - **1** → LCD is **busy** (processing command).
    - **0** → LCD is **ready** for next instruction.

Use this command to ensure the LCD has finished executing a previous command before sending a new one.

Example Code Snippet (Busy Flag Check in C):

```
while(PIN_BUSY == 1); // Wait until LCD is not busy
```

---

6. Move Cursor to CGRAM/Display

- **A (Address Bits):** Used to define specific address locations in either:
    - **CGRAM (Character Generator RAM)** — for custom characters.
    - **DDRAM (Display Data RAM)** — for setting cursor positions.

Example Commands:

- 0b01000000 → Move cursor to the **start of CGRAM**.
- 0b10000000 → Move cursor to the **start of DDRAM** (Home Position).

---

7. Read/Write ASCII Data

- **D (Data Bits):** 8-bit data representing ASCII characters or custom patterns.

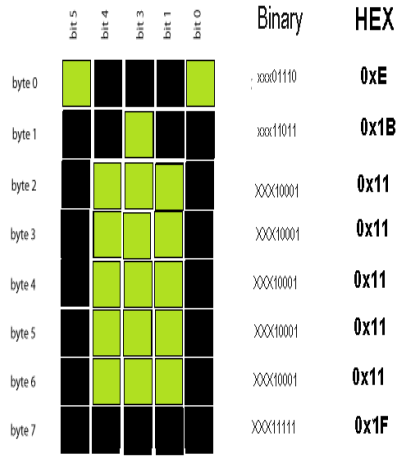
Example Commands:

- Writing 'A' → 0b01000001 (ASCII value for 'A').
- Writing '9' → 0b00111001 (ASCII value for '9').

---

8. Typical Initialization Flow (4-bit Mode Example)

1. Send 0x02 → Initialize 4-bit mode.
2. Send 0x28 → 4-bit mode, 2 lines, 5x8 font.
3. Send 0x0C → Display ON, Cursor OFF.
4. Send 0x06 → Entry mode set (cursor moves right).
5. Send 0x01 → Clear display.



Custom Character Creation on 16x2 LCD (HD44780) Using CGRAM
The image demonstrates a custom battery icon displayed on an **HD44780 16x2 LCD** using **CGRAM (Character Generator RAM)**. Let's break down the process in detail.

What is CGRAM in HD44780 LCD?
The HD44780 controller has **64 bytes of CGRAM** (Character Generator RAM) which allows you to create **8 custom characters** (each of 5x8 pixel format).
Key CGRAM Details:

- Each custom character is designed using **8 bytes** where each byte represents a row of 5 pixels.
- Only **8 locations** are available (address 0 to 7) for custom characters.

Structure of Custom Character in CGRAM
A character is defined in an **8-row x 5-column matrix**:

- Each **bit** represents an individual pixel:
  - **1** → Pixel ON
  - **0** → Pixel OFF
- Each row is controlled by writing binary data in CGRAM.

Example Custom Battery Character (from the image):

Row	Binary Code	Hex Code
Row 0	0b01110	0x0E
Row 1	0b11011	0x1B
Row 2	0b10001	0x11
Row 3	0b10001	0x11
Row 4	0b10001	0x11
Row 5	0b10001	0x11
Row 6	0b10001	0x11
Row 7	0b11111	0x1F

Steps to Create Custom Characters on HD44780 LCD

1. Initialize LCD in 4-bit/8-bit Mode

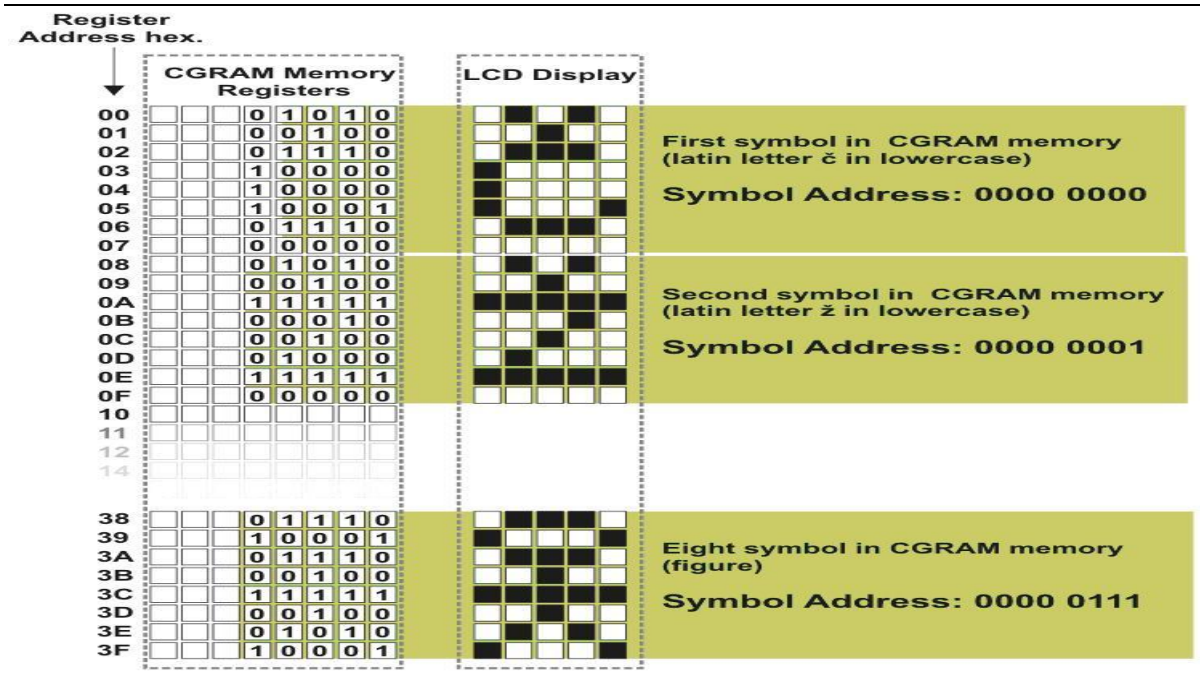
Before writing data to CGRAM, initialize the LCD as usual with proper configuration.

2. Select CGRAM Address

- Use the command **0x40** to select CGRAM.
- Each custom character starts at a specific address:
  - **Character 0** → 0x40
  - **Character 1** → 0x48
  - **Character 2** → 0x50
  - **Character 3** → 0x58
  - **Character 4** → 0x60
  - **Character 5** → 0x68
  - **Character 6** → 0x70
  - **Character 7** → 0x78

3. Write Pixel Data to CGRAM

- Send 8 rows of binary data to define the character's pattern.



Explanation of Custom Character Creation in CGRAM (HD44780 LCD)

The provided image shows how custom characters are created and stored in the **CGRAM (Character Generator RAM)** of the **HD44780 LCD Controller**.

Understanding CGRAM (Character Generator RAM)

- The **HD44780 LCD controller** allows creating **8 custom characters**.
- Each custom character is stored in **8 consecutive bytes** in CGRAM.
- Each byte represents one row in the **5x8 pixel matrix**.
- Each bit in the byte corresponds to a pixel:
  - **1** → Pixel ON
  - **0** → Pixel OFF
- Custom characters are stored in CGRAM starting at address 0x40.

CGRAM Address Mapping

- Each custom character occupies **8 bytes** in CGRAM.
- Character 0 starts at **0x40**, character 1 at **0x48**, and so on.
- The 8 available custom character addresses are:

Symbol 0 → 0000 0000 (Address 0x40)
Symbol 1 → 0000 0001 (Address 0x48)
Symbol 2 → 0000 0010 (Address 0x50)
Symbol 3 → 0000 0011 (Address 0x58)
Symbol 4 → 0000 0100 (Address 0x60)
Symbol 5 → 0000 0101 (Address 0x68)
Symbol 6 → 0000 0110 (Address 0x70)
Symbol 7 → 0000 0111 (Address 0x78)

Custom Character Examples in the Image

1. First Symbol (Latin Letter č in Lowercase) - Symbol Address: 0000 0000

CGRAM Data (8 rows of pixels):

0b00101 → 0x05
0b00101 → 0x05
0b01110 → 0x0E
0b01010 → 0x0A
0b01010 → 0x0A
0b01010 → 0x0A
0b01010 → 0x0A
0b11111 → 0x1F

- This pattern forms the symbol č.

---

2. Second Symbol (Latin Letter ž in Lowercase) - Symbol Address: 0000 0001

CGRAM Data (8 rows of pixels):

0b00101 → 0x05
0b00101 → 0x05
0b01110 → 0x0E

0b01010 → 0x0A
0b01110 → 0x0E
0b00101 → 0x05
0b00101 → 0x05
0b11111 → 0x1F

- This pattern forms the symbol ž.

---

3. Eighth Symbol (Figure/Shape) - Symbol Address: 0000 0111
CGRAM Data (8 rows of pixels):
0b01110 → 0x0E
0b01110 → 0x0E
0b00100 → 0x04
0b00100 → 0x04
0b00100 → 0x04
0b00101 → 0x05
0b00101 → 0x05
0b11111 → 0x1F

- This pattern forms a unique figure/shape.

Steps to Create Custom Characters in Code (C for LPC2148/Arduino)

1. **Initialize the LCD.**
2. **Write the custom character data to CGRAM.**
3. **Switch back to DDRAM to display characters on screen.**
4. **Use lcd_data(0) to display character at Symbol Address 0000 0000.**

Advantages of a Keypad

1. **Simple Interface**
   - Easy to connect to microcontrollers using GPIOs or keypad encoder ICs.
2. **Cost-Effective**
   - Low manufacturing and implementation cost, especially membrane keypads.
3. **Compact Design**
   - Suitable for handheld devices or systems with limited space.
4. **User-Friendly Input Method**
   - Provides a tactile or digital method for entering commands, numbers, or passwords.
5. **Customizable Layout**
   - Can be configured as 3x4, 4x4, or any matrix based on the number of keys required.
6. **Durable and Long-Lasting**
   - Especially membrane keypads — resistant to dust, moisture, and wear.

7. **Low Power Consumption**
    - Ideal for battery-operated embedded systems.
8. **Fast Input Detection**
    - With proper software scanning or interrupt handling, key detection is fast and reliable.

Applications of a Keypad

1. **Access Control Systems**
    - Used to enter passcodes or PINs for authentication (e.g., door lock systems).
2. **ATMs and Banking Systems**
    - User interface for entering card PINs, amounts, and transaction types.
3. **Home and Industrial Automation**
    - Input commands for controlling devices, fans, lights, or machines.
4. **Consumer Electronics**
    - Remote controls, microwave ovens, washing machines, calculators, etc.
5. **Point of Sale (POS) Terminals**
    - Used for inputting product codes, prices, or customer data.
6. **Embedded System Projects**
    - Password-based security, menu navigation, and system configuration.
7. **Digital Meters and Measuring Devices**
    - Input for setting thresholds, range, or mode selection.
8. **Medical Equipment**
    - Safe and clean interface for nurses and doctors to enter settings.
9. **Telecommunication Devices**
    - Phone dialing pads, intercom systems.
10. **Vending Machines**

- Item selection and payment input.

KEYPAD MATRIX

A **keypad matrix** is an input device commonly used in embedded systems for user interaction. It consists of **multiple push-button keys arranged in a row-column format** to reduce the number of GPIO pins required for interfacing.
Keypad matrices are widely used in **calculators, ATMs, security systems, vending machines, and industrial control panels**.

---

1. Structure of a Keypad Matrix
A **keypad matrix** consists of:

- **Rows (R1, R2, R3, etc.)**
- **Columns (C1, C2, C3, etc.)**
- **Keys placed at row-column intersections**

For example, in a **4×4 matrix keypad**, there are **4 rows** and **4 columns**, creating **16 keys**.

4×4 Keypad Layout Example

	C1	**C2**	**C3**	**C4**
R1	1	2	3	A
R2	4	5	6	B
R3	7	8	9	C
R4	*	0	#	D

Each button acts as a **switch**, which **connects a row and a column** when pressed.

---

2. Working Principle of a Keypad Matrix
Instead of connecting each key to a separate GPIO pin, a matrix configuration reduces the number of required pins.

- For an **M×N matrix**, only **(M + N) pins** are needed.
- A **4×4 keypad (16 keys)** needs **4 row + 4 column = 8 pins** instead of 16.

Key Detection Process

1. **Microcontroller sets all columns HIGH**.
2. **It drives one row LOW at a time** and checks which column reads LOW.
3. If a column goes LOW, the intersection of the **LOW row and LOW column** gives the key pressed.

Example: Detecting a Key Press

1. **Initial state**:
    - Rows are **HIGH**.
    - Columns are **configured as input with pull-ups**.

2. **Scanning process**:
    - Microcontroller **pulls R1 LOW**.
    - It checks for a **LOW signal in any column**.
    - If **C2 is LOW**, the pressed key is **'2' (R1, C2)**.
3. **If no column is LOW**, the microcontroller **moves to the next row** and repeats the process.

A **matrix keypad** is a type of keypad where switches (buttons) are arranged in a **matrix format** of rows and columns. It is commonly used in embedded systems to reduce the number of required I/O pins.

Benefits of a Matrix Keypad:

1. **Fewer I/O Pins**: A matrix keypad requires fewer microcontroller pins than individually connected buttons.
2. **Scalability**: Ideal when the number of required switches is large compared to the available I/O pins.
3. **Multi-functionality**: A single button can be programmed for multiple functions.

Working of a Matrix Keypad:

1. **Initialize Rows and Columns**: Configure row pins as outputs and column pins as inputs (or vice versa).
2. **Wait for Key Press**: Continuously check for a button press.
3. **Row Identification**: Determine which row the pressed key belongs to.
4. **Column Identification**: Find the corresponding column of the pressed key.
5. **Wait for Key Release**: Ensure the key is released before detecting the next press.
6. **Return Key Value**: Map the row and column to a key value using a lookup table.

Types of Keypad Matrix

Matrix keypads are categorized based on their size, technology, and construction. Here are the main types:

---

1. Based on Size (Row × Column Configuration)

Matrix keypads come in various sizes depending on the number of buttons required:

- **3×3 Keypad (9 keys)** – Used in basic applications like calculator-style input.
- **4×3 Keypad (12 keys)** – Commonly used in telephones and security systems.
- **4×4 Keypad (16 keys)** – Widely used in embedded projects like microcontroller-based password locks.
- **Custom Keypad** – Any other matrix configuration based on the application needs.

---

2. Based on Switch Technology

Matrix keypads use different types of switches to detect button presses:

- **Membrane Keypad** – Flexible, thin, and commonly used in remote controls and industrial panels.
- **Mechanical Keypad** – Uses physical push buttons, more durable and tactile (e.g., ATMs, industrial machines).
- **Capacitive Touch Keypad** – Uses touch sensors instead of physical buttons, found in modern consumer electronics.
- **Optical Keypad** – Uses infrared sensors to detect key presses, often used in specialized equipment.

---

3. Based on Connection Type

- **Wired Keypad** – Connected via direct wiring to a microcontroller or processor.
- **Wireless Keypad** – Communicates via RF, Bluetooth, or Wi-Fi (used in home automation, smart locks).

---

4. Based on Input Mechanism

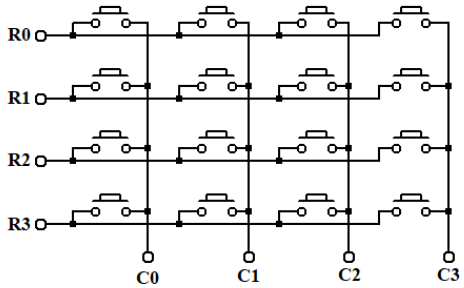
- **Single-Key Input** – Detects only one key press at a time.

- **Multi-Key Input (N-Key Rollover)** – Allows multiple key presses simultaneously, used in gaming keypads and advanced security systems.

---

5. Based on Communication Protocol

- **Parallel Interface Keypad** – Each key has a dedicated connection, requiring more pins.
- **Matrix-Scanned Keypad** – Uses fewer pins by arranging keys in a matrix (most common in embedded systems).
- **I2C/SPI Interface Keypad** – Uses serial communication to reduce pin usage, common in modern microcontrollers.



Conclusion from the Given Keypad Matrix Diagram

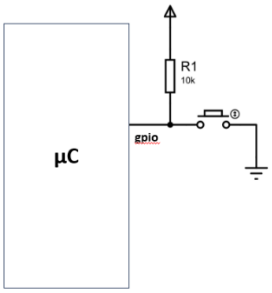
The provided diagram represents a **4×4 matrix keypad**, commonly used in embedded systems for user input. Here are some key observations and conclusions:

1. **Matrix Structure**
   - The keypad consists of **4 rows (R0 to R3)** and **4 columns (C0 to C3)**.
   - Each key is placed at the intersection of a row and a column.
2. **Working Principle**
   - All keys are normally open switches.
   - When a key is pressed, it establishes an electrical connection between a specific row and column.
   - The microcontroller detects the pressed key by scanning the rows and columns.
3. **Pin Reduction Technique**
   - Instead of using **16 separate GPIO pins**, only **8 pins (4 rows + 4 columns)** are used.
   - This reduces hardware complexity and makes the system efficient.
4. **Scanning Method**
   - **Row Scanning:** The microcontroller pulls each row LOW one by one while reading columns to detect which key is pressed.
   - **Column Scanning:** If a key is pressed, a connection is formed, and the corresponding column is detected.
5. **Applications**
   - Used in security systems (password input).

- Employed in calculators and telephone keypads.
- Common in embedded system projects for menu navigation.

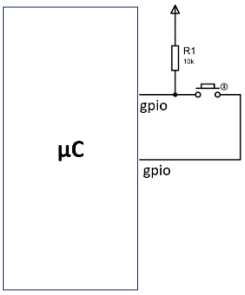
Understanding Keypad Matrix 4x4 and Normal Switch Connections

1. Normal Switch Connection



- A single push-button switch is connected to a microcontroller (µC).
- A pull-up resistor (10kΩ) is used to keep the GPIO pin at a high logic level when the button is not pressed.
- When the button is pressed, the GPIO pin is pulled to ground (LOW).
- The microcontroller reads the GPIO state to detect a button press.

2. Keypad Matrix Connection (4x4 Keypad)



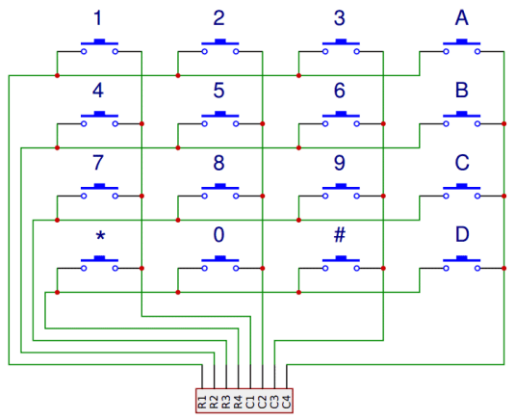
- Instead of connecting each switch separately, a **4x4 matrix keypad** groups 16 switches into **4 rows and 4 columns**.
- Each switch is placed at the intersection of a row and a column.
- The microcontroller **scans the keypad** by setting rows as outputs and columns as inputs (or vice versa).
- When a button is pressed, a connection is made between a row and a column, which the microcontroller detects.

Steps to Read a 4x4 Keypad in a Microcontroller

1. Set all rows to output (HIGH initially).
2. Set all columns as inputs with pull-up resistors enabled.
3. Drive one row LOW at a time and check which column reads LOW.

4. If a column detects LOW, that means the corresponding key is pressed.
5. Debounce the key to prevent false triggering.
6. Repeat the process for all rows.

How Does It Work?



1. Wiring the Keypad

- The **rows (R0, R1, R2, R3)** are connected to **output pins** of the microcontroller.
- The **columns (C0, C1, C2, C3)** are connected to **input pins** of the microcontroller.
- Pull-up or pull-down resistors are used on the columns to ensure a defined state (HIGH or LOW) when no button is pressed.

---

2. Scanning the Keypad

The microcontroller scans the keypad by **activating one row at a time** and **reading the columns** to detect which button is pressed. Here's how it works:

1. **Activate a Row**:
   - Set one row (e.g., R0) to **LOW (0V)** or **HIGH (e.g., 3.3V/5V)** while keeping the other rows in a **high-impedance or inactive state**.
   - For example, if R0 is activated, the microcontroller sets R0 to LOW and keeps R1, R2, and R3 HIGH or in a high-impedance state.
2. **Read the Columns**:
   - Read the state of each column (C0, C1, C2, C3).
   - If a button is pressed, the corresponding column will read **LOW** (or HIGH, depending on the circuit design).
   - For example, if the button at the intersection of R0 and C1 is pressed, C1 will read LOW when R0 is activated.
3. **Identify the Pressed Key**:
   - Based on the active row and the column that reads LOW, determine which button is pressed.
   - Use a **lookup table** to map the row and column combination to a specific key (e.g., R0 and C1 = '1').

4. **Repeat for All Rows**:
    - Repeat the process for each row (R1, R2, R3) to scan the entire keypad.

Advantages of a Keypad:

1. **Simple and Easy to Use** – Keypads provide a straightforward interface for numeric and limited alphanumeric input.
2. **Fast Data Entry** – They allow quick input compared to scrolling or touchscreen methods, especially for numeric data.
3. **Tactile Feedback** – Physical keypads provide better tactile feedback than touchscreens, making them easier to use without looking.
4. **Durability** – Mechanical and membrane keypads are built to withstand repeated usage, making them ideal for rugged environments.
5. **Low Power Consumption** – Compared to touchscreen interfaces, keypads consume significantly less power, making them ideal for battery-powered devices.
6. **Cost-Effective** – Keypads are inexpensive to manufacture and integrate into electronic devices.
7. **Reliable in Harsh Environments** – Unlike touchscreens, keypads work well even with gloves or in wet, dusty, or oily conditions.
8. **Security Features** – Keypads are widely used in security systems (such as PIN-based authentication) for access control and safe locks.
9. **No Accidental Touch** – Unlike touchscreens, keypads do not register unintended touches, reducing input errors.
10. **Works Without a Display** – Keypads can function independently of a display, making them useful for blind or low-vision users.

Applications of Keypads

Keypads are widely used in various industries and electronic systems due to their efficiency, durability, and ease of use. Some common applications include:

1. Consumer Electronics

- **Mobile Phones & Smartphones** – Older mobile phones used physical keypads for dialing and messaging.
- **Calculators** – Numeric keypads are essential for inputting mathematical calculations.
- **Remote Controls** – Used for operating TVs, ACs, and other appliances.

2. Security & Access Control

- **ATM Machines** – Used for PIN entry and transaction selection.
- **Door Lock Systems** – Digital locks use keypads for password-based access.
- **Alarm Systems** – Security keypads enable users to arm or disarm alarm systems.
- **Vaults & Safes** – PIN-based safes use keypads for authentication.

3. Industrial & Embedded Systems

- **Factory Automation** – Machines and industrial control panels use keypads for operation.

- **Medical Devices** – Equipment like glucose meters and blood pressure monitors use keypads for user input.
- **Automotive Systems** – Some vehicles use keypads for unlocking doors or entering security codes.

4. Banking & Financial Systems

- **Point of Sale (POS) Terminals** – Used for entering payment details during transactions.
- **Bank Teller Machines** – Employees use keypads for processing transactions.

5. Gaming & Entertainment

- **Arcade Machines** – Used for input in arcade and gaming consoles.
- **Musical Instruments** – Digital keyboards and synthesizers use keypads for sound input.

6. Telecommunications

- **Landline Phones** – Traditional telephones use numeric keypads for dialing.
- **Intercom Systems** – Used for communication and security verification.

7. Transportation & Navigation

- **Parking Ticket Machines** – Used for entering vehicle numbers and payment details.
- **Public Transport Systems** – Ticket vending machines and kiosks use keypads for user input.

Difference Between Switches and Keypads

Feature	Switches	Keypads
Definition	A switch is a device used to turn a circuit ON or OFF.	A keypad is an input device consisting of multiple keys (switches) arranged in a grid pattern.
Structure	Consists of a single button or toggle.	Composed of multiple buttons arranged in rows and columns.
Functionality	Primarily used for simple ON/OFF control.	Used for inputting numerical or alphanumeric data.
Examples	Toggle switch, push button, rocker switch, DIP switch.	Mobile phone keypad, ATM keypad, calculator keypad.
Output	Sends a single ON/OFF signal.	Sends multiple signals based on which key is pressed.
Application	Used in electrical circuits, light switches, and machinery.	Used in security systems, electronic devices, and embedded systems.
Complexity	Simple mechanism.	More complex as it requires scanning techniques to detect multiple key presses.
Wiring	Direct connection in a circuit.	Uses a matrix arrangement (row and column scanning) for detecting key presses.

BUZZER

What is a Buzzer?
A **buzzer** is an audio signaling device commonly used in embedded systems to provide sound-based notifications, alerts, or alarms.

Types of Buzzers
1. Piezoelectric Buzzer

- **Working Principle**: Utilizes the **piezoelectric effect** — a piezoelectric material deforms when voltage is applied, causing vibrations and generating sound.
- **Voltage**: Operates typically between **3V to 24V**.
- **Current**: Low current consumption.
- **Sound Output**: Moderate; suitable for alerts.
- **Applications**: Embedded systems, watches, alarms.
- **Advantages**: Low power, simple control with GPIO.

2. Electromagnetic Buzzer

- **Working Principle**: A magnetic coil attracts a diaphragm. An alternating current causes the diaphragm to vibrate and generate sound.
- **Voltage**: Typically 1.5V to 12V.
- **Current**: Moderate.
- **Sound Output**: Louder than piezo buzzers.
- **Applications**: Toys, household alarms, automotive.
- **Advantages**: Simple and low-cost.

3. Mechanical Buzzer

- **Working Principle**: A mechanical buzzer uses a physical mechanism (like a bell or spring) to generate sound, powered by AC or DC.
- **Voltage**: Typically 6V to 24V AC or DC.
- **Sound Output**: Loud.

- **Applications**: Industrial alarms, old telephones.
- **Advantages**: No need for complex circuitry.
- **Disadvantages**: Bulky, more power-hungry.

4. Electromechanical Buzzer

- **Working Principle**: Combines **electrical** activation with **mechanical** sound production, often with a relay-style mechanism inside.
- **Voltage**: 6V to 24V.
- **Sound Output**: Very loud and harsh.
- **Applications**: Fire alarms, emergency systems.
- **Advantages**: High decibel output.
- **Disadvantages**: Larger size, high power consumption.

5. Magnetic Buzzer

- **Working Principle**: Similar to electromagnetic buzzers, but specifically uses a **magnet and coil system** to produce audible tone when driven by AC or square wave DC.
- **Control**: Requires oscillating signal (can be PWM).
- **Voltage**: Often 3V to 5V.
- **Applications**: Microcontroller systems, small alert systems.
- **Advantages**: Loud sound in compact size.
- **Note**: Often confused with electromagnetic buzzers — they are related, but "magnetic" buzzers usually require external oscillation (not self-driven).

Comparison Table

Type	Driving Method	Power	Loudness	Requires Oscillator	Common Use
Piezoelectric	DC or PWM	Low	Moderate	Optional	Microcontrollers, alarms
Electromagnetic	DC	Moderate	High	No	Toys, alarms
Mechanical	AC/DC (Direct)	High	High	No	Old systems, bells
Electromechanical	DC with Relay	High	Very High	No	Fire/industrial alarms
Magnetic	Square Wave/PWM	Low	High	Yes (external)	Microcontrollers, smart devices

---

How a Buzzer Works

- A buzzer converts **electrical energy** into **sound energy**.
- In a microcontroller setup, the buzzer is turned ON/OFF using a **digital output pin** (HIGH or LOW).

- It can also be controlled with a **PWM signal** to vary the tone (frequency control).

✅ **Advantages of a Buzzer**

1. **Simply Compatible**
   - Easy to interface with microcontrollers using GPIO or PWM signals.
2. **Good Frequency Response**
   - Capable of producing sound at various frequencies, useful for tones and alarms.
3. **Compact Size**
   - Small physical footprint makes it suitable for space-constrained embedded designs.
4. **Low Energy Consumption**
   - Especially in the case of **piezo buzzers**, the power required is minimal.
5. **Wide Voltage Range**
   - Can operate over a range (e.g., 3V to 24V), depending on the type.
6. **High Sound Pressure**
   - Effective for attracting attention in alarms, even in noisy environments.

❌ **Disadvantages of a Buzzer**

1. **Harder to Control (in some cases)**
   - Generating complex tones requires PWM or frequency control, which can complicate coding.
2. **Produces Annoying Sound**
   - The constant or repetitive sound can be irritating in certain environments.
3. **Repair Requires Training**
   - In case of malfunction in devices using mechanical or electromechanical buzzers, proper knowledge is needed to troubleshoot.

🧩 **Applications of Buzzers**

1. **Communication Devices**
   - Used for call notifications, alerts in intercoms or phones.
2. **Automotive Electronics**
   - Seatbelt reminder, door open alert, reverse gear buzzer.
3. **Alarm Circuits**
   - Fire alarms, theft alarms, gas leak detectors.
4. **Portable Devices**
   - Beepers in pagers, handheld testing tools.
5. **Security Systems**
   - Intrusion detection alerts in access control systems.
6. **Timers**
   - Indicate the end of a countdown (kitchen timers, lab instruments).

7. **Household Appliances**
    - Microwave, washing machine, fridge door open alarms.
8. **Electronic Metronomes**
    - Used in musical devices for tempo indication.
9. **Sporting Events**
    - Timer buzzers, score signal alerts.
10. **Annunciator Panels**
    - Used in control panels for industrial alarms and alerts.
11. **Game Shows**
    - Buzzer rounds for fastest response indication.

Electromagnetic Buzzer

Definition

An **electromagnetic buzzer** is an audio signaling device that uses the principle of **electromagnetic induction** to produce sound. It is widely used in electronic circuits where a simple, audible alert or notification is required.

Working Principle

1. It consists of a **coil**, **magnet**, and **vibrating diaphragm**.
2. When DC power is applied:
    1. Current flows through the coil.
    2. The coil becomes an electromagnet and attracts a metal diaphragm.
    3. The diaphragm moves, breaking the circuit (if it's internally oscillated).
    4. This causes vibrations, producing sound waves.
3. **Frequency** of vibration = frequency of current applied (often generated internally in self-driven buzzers).

Types

1. **Self-Drive (Internally Oscillated)**
    - Contains an internal oscillator.

    - Directly connected to a DC power supply (e.g., 5V).
    - No PWM or signal needed.
2. **External-Drive (Without Oscillator)**
    - Needs an external square wave or PWM signal.
    - Controlled by microcontroller pins or timer circuits.

Construction

- Coil wound around a soft iron core.
- Metal diaphragm (vibrating plate).
- Magnet (permanent or temporary).
- Plastic enclosure for acoustic amplification.

Typical Ratings

Parameter	Value Range
Voltage	1.5V to 12V DC
Current	10mA to 50mA
Frequency	2 kHz to 4 kHz
Sound Pressure	70–90 dB at 10cm
Drive Type	DC or PWM

Advantages

- Simple to use.
- Loud output at low voltage.
- Inexpensive.
- Easy to interface with logic circuits or MCUs.

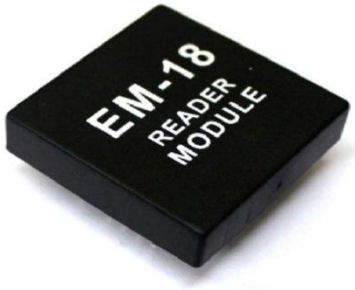
Disadvantages

- Limited frequency range.
- Can be annoying due to harsh tone.
- Not suitable for producing musical tones.

Applications

- Reverse gear alarms in vehicles.
- Timers and kitchen appliances.
- Fire and smoke alarms.
- Toy sound effects.
- Door buzzers.
- Notification alerts in embedded systems.

RIFD READER AND CARD



What is RFID?
RFID stands for **Radio Frequency Identification**. It's a contactless communication technology that uses **radio waves** to read and capture information stored on a tag/card.

Basic Components in RFID System

Component	Description
RFID Tag/Card	Stores a **unique ID**. Can be passive (no battery), active (with battery), or semi-passive.
RFID Reader	Emits radio frequency to power the tag and **reads the unique ID** from the tag.
Microcontroller	Interfaces with the RFID reader to process ID data (e.g., LPC2148 via UART).

RFID Tags/Cards Classification
RFID tags or cards are generally classified based on three main factors:

1. **Power Source**
2. **Operating Frequency**
3. **Data Capability & Security**

1. Based on Power Source

Type	Description	Examples
Passive	No internal battery. Powered by the RFID reader's RF field.	EM4100, MIFARE 1K
Active	Has an internal battery to power its own transmission. Longer range.	Used in tolls, logistics

Semi-Passive	Battery powers internal circuitry but not transmission. Hybrid performance.	Temperature loggers

2. Based on Operating Frequency

Frequency	Range	Examples	Applications
Low Frequency (LF)	125–134 kHz	EM4100, TK4100	Access control, livestock tracking
High Frequency (HF)	13.56 MHz	MIFARE 1K, DESFire	Library systems, smart cards, NFC
Ultra-High Frequency (UHF)	860–960 MHz	EPC Gen2, Alien Higgs	Inventory, logistics, long-range tracking

3. Based on Data Capability & Security

Type	Description	Card Examples
Read-Only	Pre-programmed UID. Cannot be changed.	EM4100, TK4100
Read/Write	Data can be written and read multiple times.	MIFARE 1K, ICODE SLI
Secure Cards	Features encryption, authentication, multiple sectors, memory protection.	MIFARE DESFire, HID iCLASS

Working Principle

1. **RFID reader emits RF signal** (usually 125kHz or 13.56 MHz).
2. The **RFID tag/card gets powered** by this field (if passive).
3. The card **sends back its unique ID**.
4. The reader **receives the ID via UART** or other serial protocols.
5. The microcontroller **compares it to stored IDs** for authentication or further processing.

---

1. What is RFID?

Radio Frequency Identification (RFID) is a contactless automatic identification technology that uses **radio waves** to transfer data between a **reader** and a **tag/card** attached to an object for the purpose of identifying and tracking it.

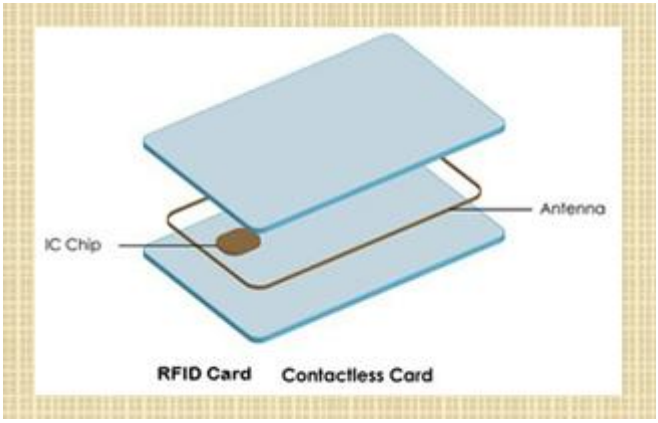
It's widely used in **access control systems**, **inventory tracking**, **supply chain**, **automated billing**, **toll collection**, **ATM authentication**, and more.

✅ **2. Basic Components of an RFID System**

Component	Role
Tag/Card	Stores unique data (ID, credentials). Mounted on objects or users.
Reader	Sends RF signals to power passive tags, receives response data.

Antenna	Transmits signals between tag and reader.
Host System	Runs the software that processes and interprets tag data.
API Software	Interfaces reader data with application-level logic.

✅ 3. RFID Tags: Types & Variations



🎯 A. Based on Power Source

Type	Description	Range	Cost	Example Uses
Passive	No battery. Uses energy from reader signal.	Short (cm–1m)	Low	Access control, library

Active	Has internal battery. Actively transmits data.	Long (30–100m)	High	Vehicle tracking, logistics
Semi-Passive	Battery powers internal circuitry only (not transmitter).	Medium	Moderate	Temperature sensors

---

🎯 B. Based on Memory & Functionality

Tag Type	Description
Read-Only	Data written at manufacture time. Only UID can be read.
WORM	Write Once Read Many. Data can be written once by user, then locked.
Read/Write	Data can be repeatedly written and read. Can store user info, logs, etc.

Memory Size:

- Ranges from **16 bits to several kilobytes** (e.g., 512 bytes, 1KB, 4KB).
- Some tags support **partitioning and locking** of memory blocks.

🎯 C. Based on Frequency

Frequency Band	Range	Common Tags	Examples
LF (125–134 kHz)	4–10 cm	EM4100, TK4100	Pet ID, Access control, EM-18 module
HF (13.56 MHz)	up to 1 meter	MIFARE 1K, DESFire	Smart cards, metro, ATM access, libraries
UHF (860–960 MHz)	2–7 meters	EPC Gen2	Retail, supply chain, pallet tracking
Microwave (2.45 GHz)	1–2 meters	ISO 18000-4	Electronic Article Surveillance (EAS)

ℹ **EM-18** uses 125kHz and works with passive, read-only cards like **EM4100**.
RC522 uses 13.56 MHz and works with **MIFARE 1K** (read/write).

✅ 4. Working Principle of RFID System

📡 Sequence of Communication:

1. **Reader** sends a carrier RF signal through antenna.
2. Passive **tag receives energy** from the signal.
3. Tag **modulates the signal** using load modulation or backscatter.
4. The **reader receives the modulated signal**.
5. Reader **decodes the tag data** and passes it to the **host computer/application**.

➕ Example:

- Tag ID: 0A0035BC29
- Reader receives serial UART data: 0x02 30 41 30 30 33 35 42 43 32 39 0x03

✅ 5. Tag Technologies & Memory Types

Memory Tech	Description
EEPROM	Electrically erasable, used in MIFARE, general tags
Antifuse	Permanent one-time programmable memory
FeRAM	Fast, energy-efficient read/write memory

Other features:

- **Anti-collision protocols** (read multiple tags simultaneously)
- **Sector protection** (lock data by block or segment)
- **UID + additional user memory** in secure tags

✅ 6. Popular RFID Tags Overview

🔹 **EM4100 / TK4100**

- 125 kHz LF
- 64-bit UID (10 hex characters)
- Read-only
- Cheap and simple
- Works with EM-18 module

🔹 **MIFARE 1K**

- 13.56 MHz HF
- 1 KB EEPROM in 16 sectors
- Supports password-based authentication
- Works with RC522, PN532
- Reusable and secure

🔹 **MIFARE DESFire**

- High-security version with AES/3DES encryption
- Used in transport ticketing and secure entry

🔹 **UHF EPC Gen2**

- 860–960 MHz
- High-speed, longer range (up to 5–10 meters)
- Used in logistics, warehouse, retail

✅ 7. RFID Frequency Comparison Table

Frequency	Range	Speed	Memory	Use Case
125 kHz (LF)	<10 cm	Low	Small	Access control, animal ID
13.56 MHz	~1 meter	Moderate	1KB+	Smart cards, payment, libraries
433 MHz	1–100 m	Moderate	Varies	Asset tracking, military pallets
860–930 MHz	2–7 meters	High	Varies	Retail supply chain, inventory
2.45 GHz	~2 meters	High	Varies	Item-level security, RFID tunnels

✅ 8. Reader Types

Reader Type	Description	Example Use
Fixed Reader	Installed at gates, doors, POS	Warehouse, ATMs, building entry
Handheld	Portable, battery-powered	Inventory check, field asset scan
Embedded	Used in embedded systems via UART/SPI	EM-18 with LPC2148 or Arduino

✅ 9. Regulatory & Frequency Bands

Band	Region	Regulation
125 kHz	Global (mostly)	Unlicensed, ISM band
13.56 MHz	Global	ISM band, regulated
860–960 MHz	Region-specific	US: 902–928 MHz, EU: 865–868 MHz
2.45 GHz	Global ISM band	Shared with Wi-Fi, Bluetooth

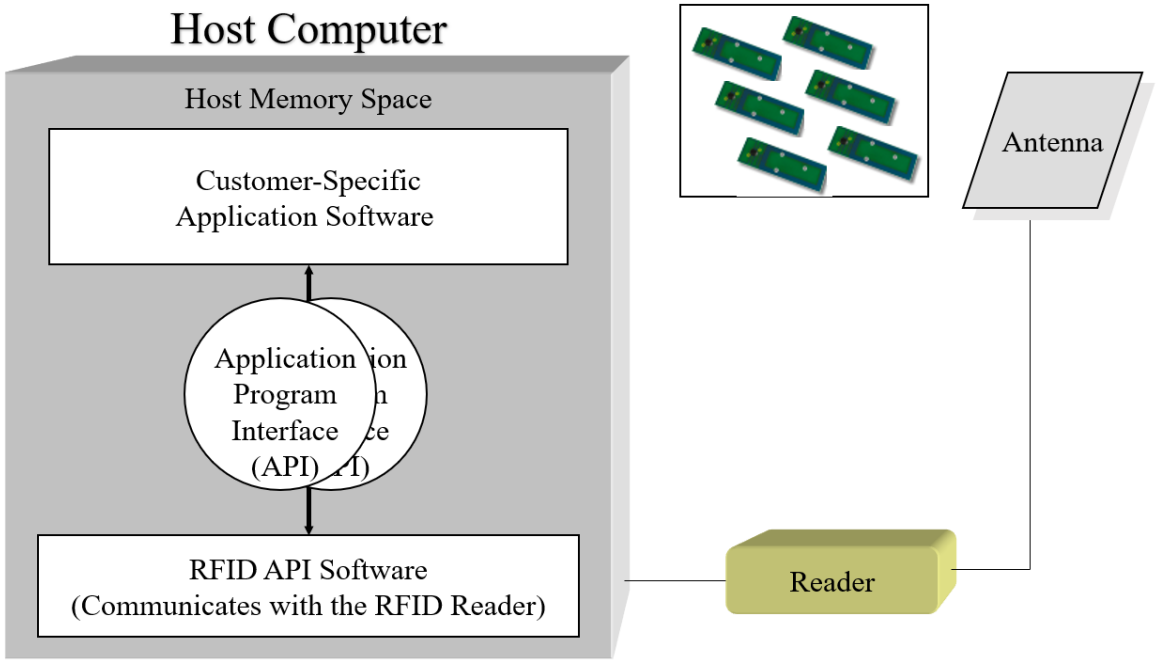
✅ 10. Use Case Matrix

Use Case	Tag Type	Frequency	Recommended Tag
ATM Authentication	Passive R/W	13.56 MHz	MIFARE 1K with RC522
Access Control	Passive RO	125 kHz	EM4100 with EM-18 reader
Library Card	Passive R/W	13.56 MHz	MIFARE 1K
Warehouse Tracking	Passive R/W	860–960 MHz	EPC Gen2
Pet/Animal ID	Passive RO	125 kHz	EM4305, FDX-B

✅ Summary – Ask Yourself 5 Key Questions

1. **How far?** → Decide range (cm, meters)
2. **How fast?** → Data rate needed?
3. **How many?** → One or many tags at a time?
4. **How much?** → Cost constraints?
5. **Attached to what?** → Material: metal, water, plastic?

Components of an RFID System (RFID System Architecture Explanation)



■ 1. Host Computer

The **central processing unit** of the system. It runs the main software and coordinates RFID operations.

📦 **Inside the Host Computer:**

- **Host Memory Space**: Stores temporary and permanent data.
- **Customer-Specific Application Software**: This is the custom software developed for a specific RFID application (e.g., inventory tracking, access control, ATM authentication).
- **API (Application Program Interface)**: Acts as a bridge between the custom application and the RFID API. It enables smooth communication and abstraction.
- **RFID API Software**: Communicates directly with the RFID reader hardware. It receives raw data from the reader and passes it to the application software via API.

📡 2. Antenna

The **antenna** emits **radio frequency (RF) signals** to power RFID tags (in the case of passive tags) and **receive modulated signals** reflected back from the tags.

- It is a critical component that defines the **read range**, **zone**, and **signal quality**.
- Can be **internal (in small modules)** or **external (industrial systems)**.
- Can be directional or omnidirectional depending on the use case.

📖 3. RFID Tags (shown above the antenna)

These are the **green circuit boards** in the image. RFID tags are attached to objects or carried by people.

- Tags contain **unique identifiers** (UID) or user data.
- May be **read-only**, **write-once**, or **read/write**.
- Communicates via **inductive coupling (LF/HF)** or **electromagnetic wave backscatter (UHF)**.

🟧 4. Reader

The **reader** (interrogator) is responsible for:

- Generating RF signals (via the antenna)
- Powering passive tags
- Receiving the **modulated response** from tags
- **Decoding** and **transmitting** the data to the host system

Most readers interface with the host computer via:

- UART/RS-232
- USB
- Ethernet
- SPI/I2C (for embedded systems)

🔁 Data Flow Overview

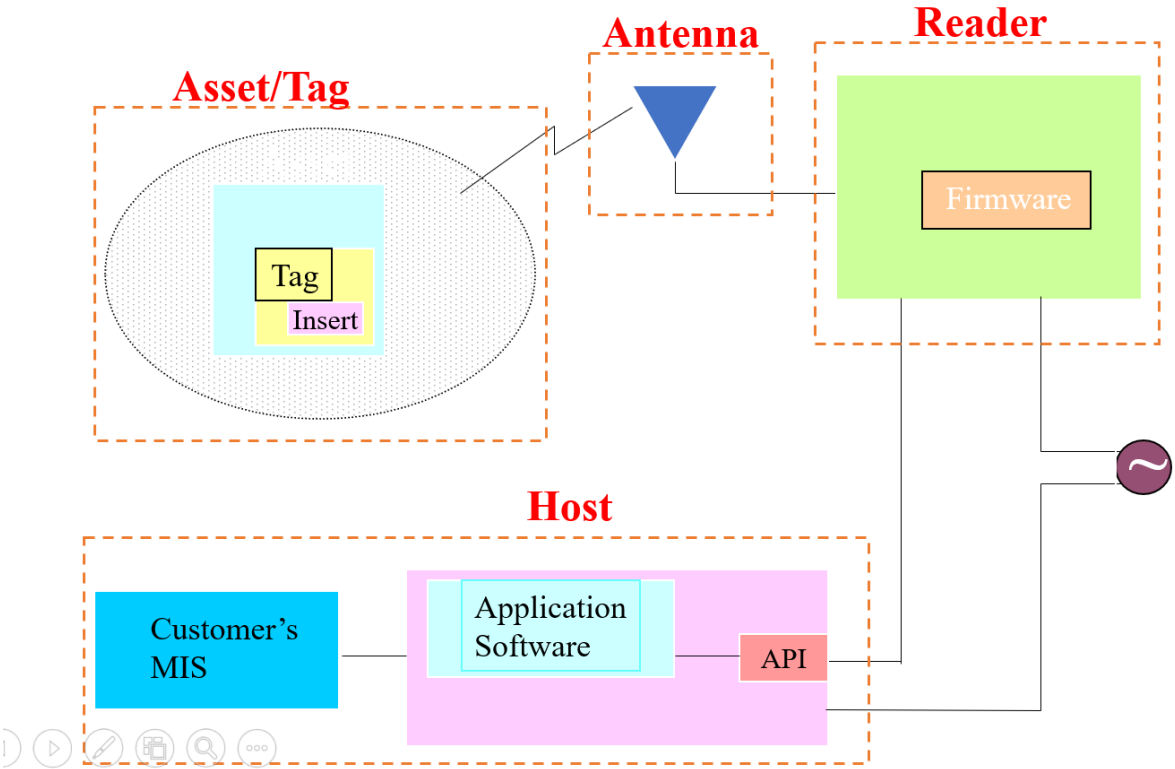
1. **Host Application** sends a command (e.g., "Read tags") via API.
2. **API** communicates with **RFID API software**.
3. **RFID API** sends command to **Reader**.
4. **Reader** energizes the **Antenna** to send RF signals.
5. **Tag(s)** respond by modulating the RF signal (with UID or memory content).
6. **Antenna** picks up the signal and sends it back to the **Reader**.
7. **Reader** decodes the signal and returns it to the **RFID API**.
8. **RFID API** passes the decoded tag data to the **Application Software**.
9. **Application Software** logs, processes, displays, or makes decisions (e.g., grant access, record entry, process payment).

✅ Summary Table

Component	Function
Host Computer	Controls system, runs application software
Application SW	Custom business logic (e.g., ATM, gate, library system)
API	Connects application with hardware-level commands
RFID API	Talks to the RFID Reader

Reader	Powers antenna, communicates with tags
Antenna	Transmits and receives RF signals
Tags	Store identification data and respond to reader

RFID System Components



1. Asset/Tag Section

This is the **object or person** that carries the **RFID tag**.

- **Tag**
  - A microchip + antenna encapsulated in plastic, paper, etc.
  - Stores a **Unique ID** or data.
  - Can be:
    - **Read-only** (like EM4100)
    - **Read/Write** (like MIFARE 1K)
  - No power source (passive) or internal battery (active/semi-active).
- **Insert**
  - Optional part (like ID card insert or embedded chip).
  - Indicates the tag is embedded or attached to the asset being tracked.

---

2. Antenna Section

- The antenna is responsible for **transmitting and receiving RF signals**.
- The reader uses the antenna to:
  - **Send energy** to passive tags.
  - **Receive modulated signals** carrying tag data.
- Can be **built-in** (e.g., in handheld readers) or **external** for extended range.

🟩 3. Reader Section

- The **core electronic unit** that controls RF communication.
- Sends carrier signal via antenna.
- Receives modulated tag response.
- **Decodes** data and forwards it to the host system.

🔸 Firmware

- The reader runs internal **firmware** (low-level control software).
- Manages RF operations, anti-collision, tag decoding, protocol handling.

🟪 4. Host Section

This is the **user interface and processing environment**.

🔹 Customer's MIS (Management Information System)

- Main enterprise system.
- Stores, manages, and displays RFID-related data (inventory, logs, transactions).

🔹 Application Software

- Custom software written for specific RFID tasks.
- Could be part of:
  - An ATM software
  - Access control
  - Inventory tracking

🔹 API (Application Programming Interface)

- Bridge between the software and RFID hardware.
- Converts high-level application commands (like "scan tag") into low-level reader instructions.

🔌 Power Input

- Reader is powered by an **external power source** (typically DC or AC adapter).
- Supplies energy to operate the firmware and energize the antenna.

🔁 Complete Operational Flow

1. **Power is applied** to the RFID reader.
2. The **reader sends RF energy** through the antenna.
3. The **RF field reaches a tag** near the antenna.

4. The **tag responds** by reflecting and modulating the signal (in passive systems).
5. **Reader receives** the modulated signal, decodes it using **firmware**.
6. Reader sends the decoded ID/data to the **host** via an interface.
7. The **API** translates raw data into usable format for the **Application Software**.
8. The **Customer's MIS** uses the data for higher-level decision-making (e.g., record entry, unlock door, approve transaction).

Key Points Summary

Component	Function
Tag	Stores and responds with ID/data when energized
Antenna	Sends energy to tag and receives response signal
Reader	Controls communication, decoding, and interaction with host
Firmware	Embedded software inside the reader, handles low-level tag protocols
API	Interface to connect software with hardware
Application Software	High-level software that uses tag data for specific tasks
MIS	Organizational system that integrates RFID data into business processes

---

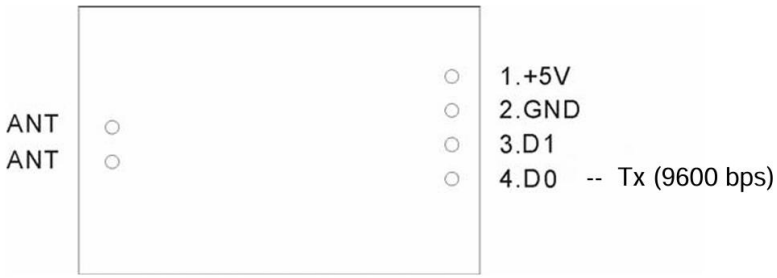
125kHz Embedded RFID Reader Module (YET-125K)

1. Module Overview

Feature	Description
Model Name	YET-125K
Frequency	125 kHz (Low Frequency RFID)
Type	Embedded RFID Reader Module (PCB board type)
Supported Cards	EM4100, TK4100, EM4305, compatible 125kHz tags
Output Interfaces	UART (TTL), Wiegand 26/34, RS232 (depends on model variant)
Typical Range	2–10 cm depending on tag size and antenna quality
Voltage Input	3.3V to 5V DC
Current Consumption	≈ 20–40 mA
Data Output Format	10-digit card UID in ASCII (via UART), Wiegand bit stream, or RS232

✅ 2. Pin Configuration (UART Model)

PIN Description



Note: For TTL – D0: Tx – Connected 9600bps, D1: NC (Not Connected)

Pin Name	Function
VCC	Power Supply (3.3–5V DC)
GND	Ground
TX	Data Output (UART Transmit)
RX	(Sometimes not used, depends on variant)
LED	Tag detect indicator output (LOW when tag detected)
ANT1/ANT2	External antenna terminals

You can directly connect **TX pin to a microcontroller UART RX** pin (like LPC2148 UART0 RXD).

✅ 3. Working Principle

1. Reader emits **low-frequency RF field (125kHz)** through the connected antenna.
2. Passive RFID tag within the field **gets energized**.
3. The tag responds by **modulating the RF signal** with its UID (usually 10 hex digits).
4. YET-125K **demodulates** the signal and sends the data to host via **UART/Wiegand/RS232**.
5. Host system processes UID and performs action (e.g., unlock gate, log attendance).

✅ 4. Output Format (UART)

- Data is sent in **ASCII format**, 10 digits + optional start/end symbols.
- Example output for tag with UID 0A0035BC29:

0x02 30 41 30 30 33 35 42 43 32 39 0x03

  - 0x02 = Start byte
  - ASCII of 0A0035BC29
  - 0x03 = End byte

✅ 5. Application Areas

Application	Description
Access Control Systems	Door unlocking using RFID cards
Time Attendance	Logging employee presence via UID
Parking Systems	Vehicle entry using passive RFID tags
Automation Systems	Triggering events based on tag presence
RFID Locks	Used in lockers or security boxes
Asset Tracking	Identification of items or personnel

✅ **6. Advantages**

- **Low Power** operation (ideal for battery-based systems)
- **Simple UART interface** – directly connects to microcontrollers
- **Works with widely available 125kHz tags**
- **Compact and Embeddable** – easy to integrate into custom enclosures

---

EM4095

✅ **1. What is EM4095?**

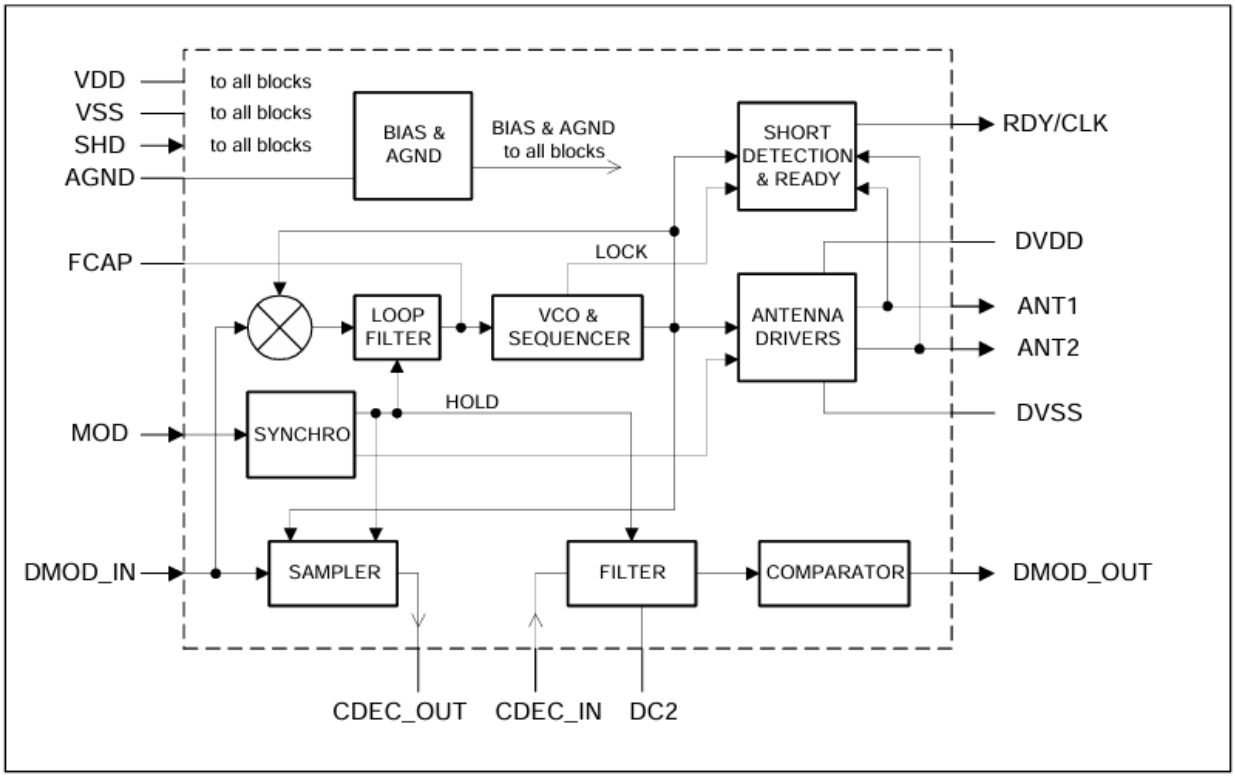
The **EM4095** is a **CMOS integrated RFID front-end chip** that performs the following:

- Drives the antenna at the carrier frequency (125 kHz)
- Supports **amplitude modulation (AM)** for data transmission
- Performs **AM demodulation** of the signal modulated by RFID tags (transponders)
- Interfaces directly with a **microcontroller**

✅ **2. Key Features**

Feature	Description
Carrier Frequency	100 kHz – 150 kHz (typically tuned to 125 kHz)
PLL System	Automatically locks to antenna's resonant frequency
Transponder Protocols	Compatible with EM4100, EM4102, EM4200, EM4305, EM4450, etc.
Modulation Support	OOK (100% AM), adjustable modulation index
No External Oscillator	Built-in PLL eliminates need for external crystal
Antenna Drive	Integrated bridge driver or single-ended driver
Sleep Mode	1 µA for ultra-low power applications
Power Supply	4.1 V – 5.5 V (compatible with USB supply)
Temperature Range	-40 °C to +110 °C
Package	SOIC-16

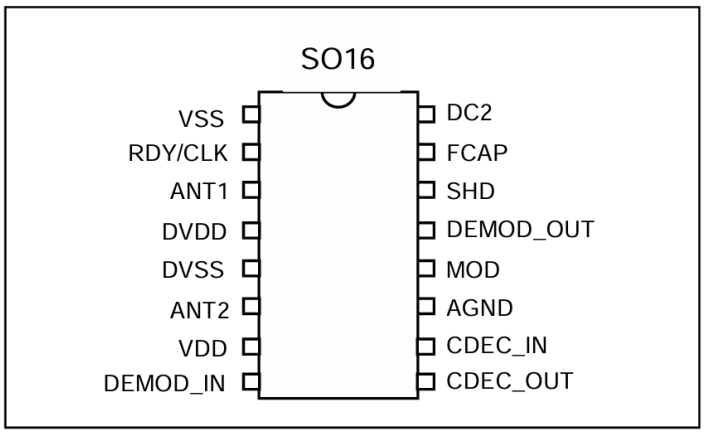
✅ 3. Block Diagram (Functions)



➤ Main Functions

- **Transmission**: Drives the antenna coil using a PLL-controlled oscillator
- **Reception**: Demodulates the AM signal from the RFID tag and outputs digital data
- **Modulation**: Supports on-off keying (OOK) by disabling drivers
- **Microcontroller Interface**: Provides DEMOD_OUT, MOD, RDY/CLK for communication

✅ 4. Pins and Their Functions



Pin No	Name	Description
1	VSS	Ground
2	RDY/CLK	Clock output / Ready signal for MCU
3	ANT1	Antenna drive output
4	DVDD	Digital power for antenna drivers
5	DVSS	Ground for antenna drivers
6	ANT2	Antenna drive output
7	VDD	Main power supply
8	DEMOD_IN	Antenna voltage input for PLL + demodulation
9–10	CDEC_IN/OUT	Capacitor for DC offset cancellation
11	AGND	Analog ground
12	MOD	Logic input to modulate antenna field
13	DEMOD_OUT	Digital output of demodulated tag signal
14	SHD	Logic input to enter/exit sleep mode
15	FCAP	Loop filter capacitor for PLL
16	DC2	Decoupling capacitor

✅ 5. Operating Principle

A. Transmission

- Antenna is driven at resonance frequency (~125 kHz) via a bridge driver.
- MOD pin controls modulation: pulling it high disables drivers for AM signal transmission (OOK).
- PLL ensures carrier frequency matches antenna's resonant frequency.

B. Reception

- The tag modulates the field via **load modulation**.
- The signal from the antenna is sensed through DEMOD_IN.
- AM demodulator processes the signal and outputs it on DEMOD_OUT.

✅ 6. Interface with Microcontroller

Signal	Role
DEMOD_OUT	Connect to MCU RX pin or digital input
RDY/CLK	Indicates reader is active and clock-ready
MOD	MCU pulls high to send data to a tag
SHD	Put HIGH to enter sleep, LOW to activate

✅ 7. Example Use Cases

Application	Use
Low-frequency RFID reader	Read EM4100/EM4305 cards or animal tags
Car immobilizers	Recognize unique RFID keys
Access control	Unlock door on authorized card detection
Handheld readers	Portable systems using 125 kHz RFID
Tool tracking systems	Identify RFID-tagged equipment

✅ 8. Electrical Characteristics (Highlights)

Parameter	Typical Value
Sleep mode current	1 µA
Active mode current (excluding driver)	5–7 mA
Antenna peak current (recommended)	≤ 250 mA
Input logic high (MOD, SHD)	≥ 0.8 × VDD
Output logic low (DEMOD_OUT)	≤ 0.1 × VDD
Antenna driver resistance	3–9 Ω
Operating voltage	4.1 V – 5.5 V

✅ 9. Application Circuit (Basic Read-Only Setup)

- Connect a **resonant LC antenna** (125 kHz tuned) to ANT1/ANT2.
- Use capacitors CDC2, CDEC, CFCAP for filtering.
- Connect DEMOD_OUT to MCU's digital input.
- Keep MOD low (grounded) for read-only mode.
- Pull SHD low to enable operation.

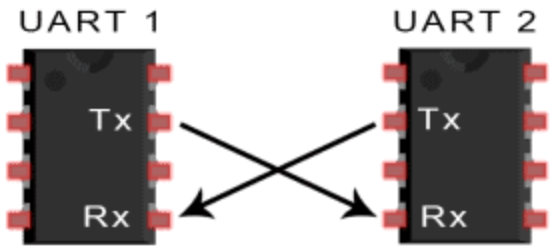
✅ 10. Advantages Over EM4094/Discrete Designs

- Integrated PLL eliminates need for external crystals.
- High tag compatibility (EM4102/4305 etc.).
- Compact integration with minimal external components.
- Better power efficiency and stable carrier generation.

UART (Universal Asynchronous Receiver Transmitter)

Introduction to UART Communication

In UART communication, two UARTs communicate directly with each other. The transmitting UART converts parallel data from a controlling device like a CPU into serial form, transmits it in serial to the receiving UART, which then converts the serial data back into parallel data for the receiving device. Only two wires are needed to transmit data between two UARTs. Data flows from the Tx pin of the transmitting UART to the Rx pin of the receiving UART:



UARTs transmit data *asynchronously*, which means there is no clock signal to synchronize the output of bits from the transmitting UART to the sampling of bits by the receiving UART.

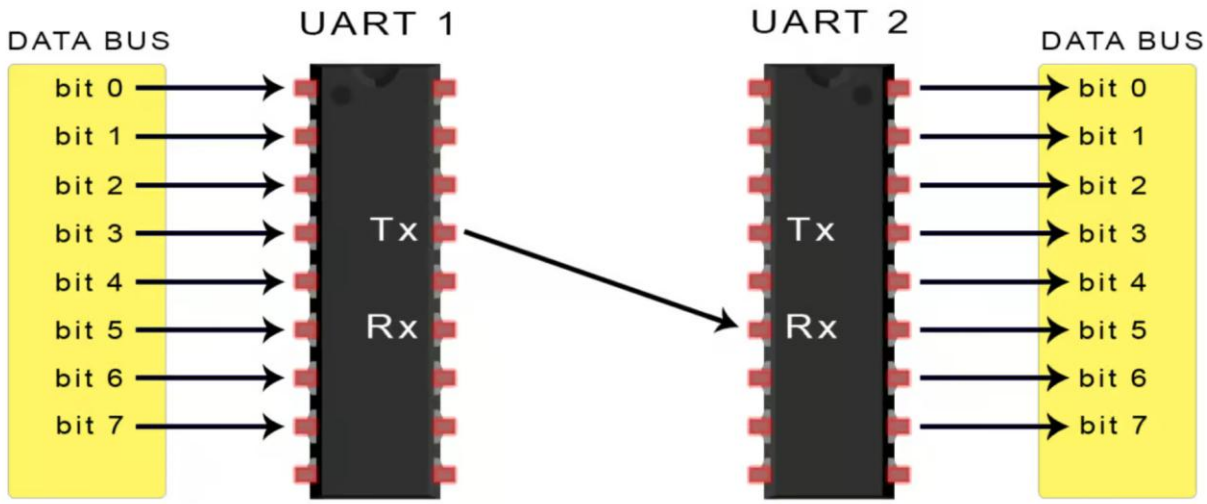
Instead of a clock signal, the transmitting UART adds start and stop bits to the data packet being transferred. These bits define the beginning and end of the data packet so the receiving UART knows when to start reading the bits.

When the receiving UART detects a start bit, it starts to read the incoming bits at a specific frequency known as the *baud rate.* Baud rate is a measure of the speed of data transfer, expressed in bits per second (bps). Both UARTs must operate at about the same baud rate. The baud rate between the transmitting and receiving UARTs can only differ by about 10% before the timing of bits gets too far off.
Both UARTs must also must be configured to transmit and receive the same data packet structure.

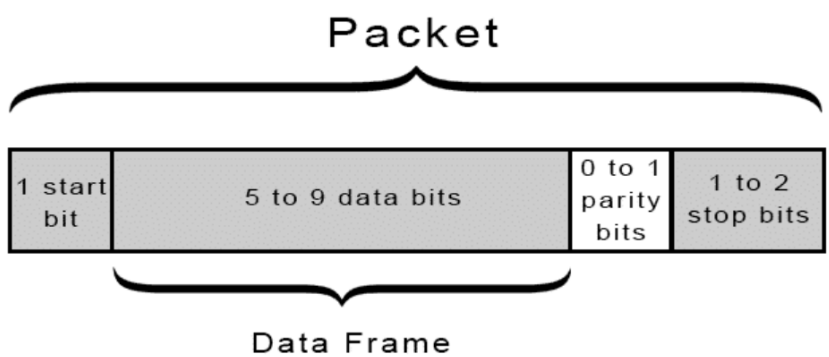
Parameter	Details
Wires Used	2
Maximum Speed	Any speed up to 115200 baud, usually 9600 baud
Synchronous or Asynchronous?	Asynchronous
Serial or Parallel?	Serial
Max # of Masters	1
Max # of Slaves	1

How UART Works

The UART that is going to transmit data receives the data from a data bus. The data bus is used to send data to the UART by another device like a CPU, memory, or microcontroller. Data is transferred from the data bus to the transmitting UART in parallel form. After the transmitting UART gets the parallel data from the data bus, it adds a start bit, a parity bit, and a stop bit, creating the data packet. Next, the data packet is output serially, bit by bit at the Tx pin. The receiving UART reads the data packet bit by bit at its Rx pin. The receiving UART then converts the data back into parallel form and removes the start bit, parity bit, and stop bits. Finally, the receiving UART transfers the data packet in parallel to the data bus on the receiving end:



UART transmitted data is organized into *packets*. Each packet contains 1 start bit, 5 to 9 data bits (depending on the UART), an optional *parity* bit, and 1 or 2 stop bits:



Start Bit

The UART data transmission line is normally held at a high voltage level when it's not transmitting data. To start the transfer of data, the transmitting UART pulls the transmission line from high to low for one clock cycle. When the receiving UART detects the high to low voltage transition, it begins reading the bits in the data frame at the frequency of the baud rate.

Data Frame

The data frame contains the actual data being transferred. It can be 5 bits up to 8 bits long if a parity bit is used. If no parity bit is used, the data frame can be 9 bits long. In most cases, the data is sent with the least significant bit first.

Parity

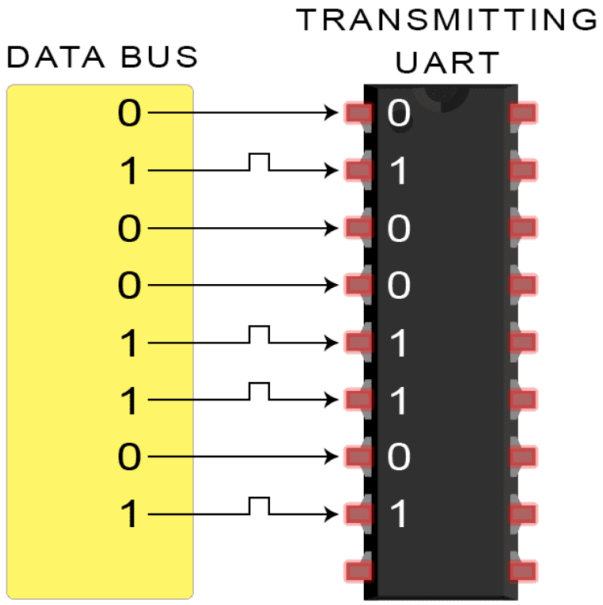
Parity describes the evenness or oddness of a number. The parity bit is a way for the receiving UART to tell if any data has changed during transmission. Bits can be changed by electromagnetic radiation, mismatched baud rates, or long distance data transfers. After the receiving UART reads the data frame, it counts the number of bits with a value of 1 and checks if the total is an even or odd number. If the parity bit is a 0 (even parity), the 1 bits in the data frame should total to an even number. If the parity bit is a 1 (odd parity), the 1 bits in the data frame should total to an odd number. When the parity bit matches the data, the UART knows that the transmission was free of errors. But if the parity bit is a 0, and the total is odd; or the parity bit is a 1, and the total is even, the UART knows that bits in the data frame have changed.

Stop Bits

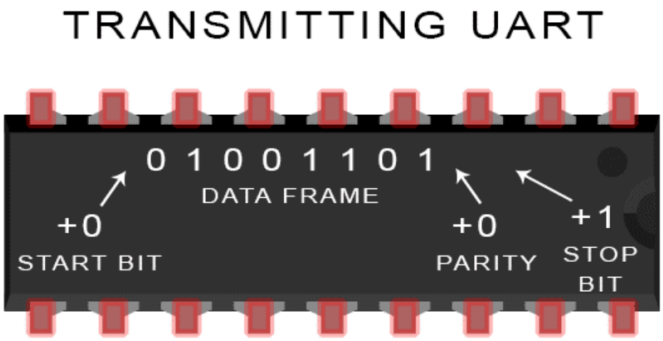
To signal the end of the data packet, the sending UART drives the data transmission line from a low voltage to a high voltage for at least two bit durations.

Steps of UART Transmission

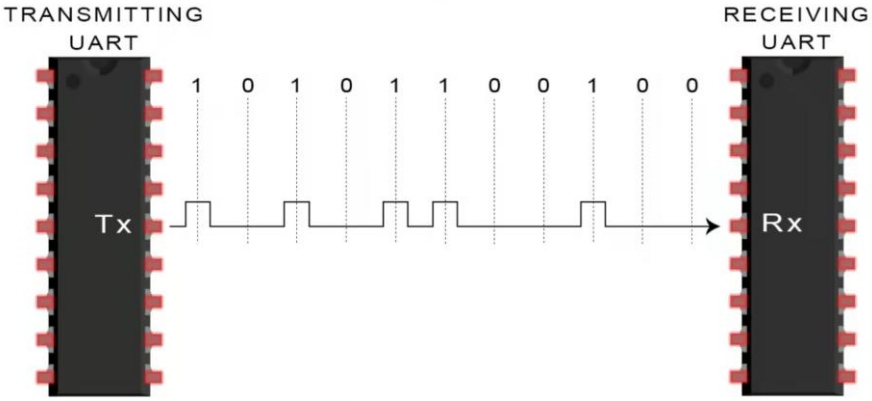
1. The transmitting UART receives data in parallel from the data bus:



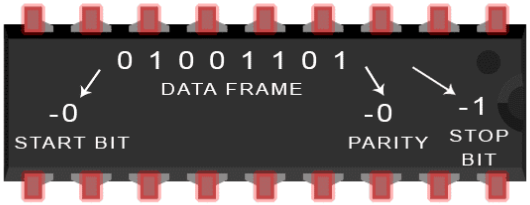
2. The transmitting UART adds the start bit, parity bit, and the stop bit(s) to the data frame:



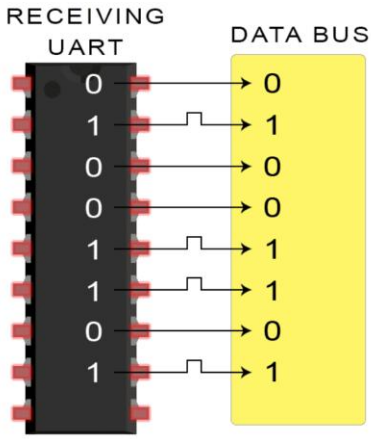
3. The entire packet is sent serially from the transmitting UART to the receiving UART. The receiving UART samples the data line at the pre-configured baud rate:



4. The receiving UART discards the start bit, parity bit, and stop bit from the data frame:



5. The receiving UART converts the serial data back into parallel and transfers it to the data bus on the receiving end:



Advantages and Disadvantages of UARTs

No communication protocol is perfect, but UARTs are pretty good at what they do. Here are some pros and cons to help you decide whether or not they fit the needs of your project:

Advantages

- Only uses two wires
- No clock signal is necessary
- Has a parity bit to allow for error checking
- The structure of the data packet can be changed as long as both sides are set up for it
- Well documented and widely used method

Disadvantages

- The size of the data frame is limited to a maximum of 9 bits
- Doesn't support multiple slave or multiple master systems
- The baud rates of each UART must be within 10% of each other

---

📌 1. What is UART?

UART (Universal Asynchronous Receiver/Transmitter) is a **serial communication protocol** used for asynchronous data transmission between two devices.

Unlike SPI or I2C, UART does **not use a clock line**. Instead, both communicating devices agree on the **baud rate** (data transfer speed).

📐 2. UART Communication Basics

➤ 2.1 Transmission Medium:

- **2 wires:**
  - **TX** (Transmit Data)
  - **RX** (Receive Data)
  - (Optional: GND shared)

➤ **2.2 No Clock:**

- UART is **asynchronous**, meaning the **timing is inferred** by each device using agreed **baud rate**.

📶 3. UART Frame Format

A UART frame is a **series of bits** sent in a particular format:

| Start Bit | Data Bits | Optional Parity Bit | Stop Bit(s) |

➤ **3.1 Start Bit:**

- Always **1 bit**, logic **LOW (0)**.
- Signals receiver to start reading the incoming data.

➤ **3.2 Data Bits:**

- Typically **5 to 9 bits**.
- Most commonly **8 bits** (1 byte).

➤ **3.3 Parity Bit (optional):**

Used for **error detection**:

- **Even parity**: Total 1s (including parity bit) is even.
- **Odd parity**: Total 1s is odd.

➤ **3.4 Stop Bit(s):**

- **1 or 2 bits**.
- Logic HIGH (1) used to indicate end of data frame.

⚙️ 4. UART Configuration Parameters

These must match on both transmitter and receiver:

Parameter	Description
Baud Rate	Transmission speed (e.g., 9600, 115200 bps)
Data Bits	Usually 8
Parity	None, Even, or Odd
Stop Bits	1 or 2
Flow Control	None (default), or RTS/CTS

🧮 5. Baud Rate Calculation

In microcontrollers, the **baud rate** is calculated using formula:
For example,

In **LPC2148 (VIC + UxDLL/UxDLM + UxFDR)**:
Baud Rate = (PCLK) / (16 × (256 × UxDLM + UxDLL) × MULVAL / DIVADDVAL)

In **ATmega328P**:
UBRR = (F_CPU / (16 * BAUD)) - 1
Where:

- UBRR – UART Baud Rate Register
- F_CPU – System clock
- BAUD – Desired baud rate

🔁 6. UART Communication Types

Type	Description
Full-Duplex	Both TX and RX can operate simultaneously
Half-Duplex	TX and RX share a line; communication alternates
Simplex	One-way only (either TX or RX, not both)

📡 7. Hardware Flow Control

RTS/CTS (Request to Send / Clear to Send)
Used to manage flow of data:

- **RTS (output):** From UART device to request permission.
- **CTS (input):** From other device to grant permission.

🧪 8. UART Error Types

Error Type	Meaning
Overrun Error	Data overwritten before being read
Parity Error	Parity bit does not match expected
Framing Error	Stop bit not detected correctly (e.g., noise)
Break Condition	Line held low for longer than a full-frame duration

---

🧱 9. UART Registers (Generic & MCU-specific)

LPC2148 UART0 Registers:

Register Name	Description
U0RBR	Receiver Buffer Register
U0THR	Transmit Holding Register
U0DLL/U0DLM	Divisor Latch LSB/MSB
U0FDR	Fractional Divider Register
U0LCR	Line Control Register
U0LSR	Line Status Register
U0IER	Interrupt Enable Register
U0IIR	Interrupt Identification Register

📲 10. Steps to Initialize UART (Embedded C Pseudocode)

```
void UART0_Init() {
    PINSEL0 |= 0x00000005;    // Enable UART0 RXD/TXD on P0.0 & P0.1
    U0LCR = 0x83;             // Enable DLAB and set 8-bit data
    U0DLL = 97;               // 9600 Baud Rate (with PCLK = 15MHz)
    U0DLM = 0;
    U0LCR = 0x03;             // Disable DLAB, 8-bit data
}
```

📥 11. Transmitting Data (Example)

```
void UART0_TxChar(char ch) {
    while (!(U0LSR & 0x20)); // Wait till THR is empty
    U0THR = ch;
}
```

📤 12. Receiving Data (Example)

```
char UART0_RxChar(void) {
    while (!(U0LSR & 0x01)); // Wait till data is available
    return U0RBR;
}
```

🧠 13. Applications of UART

- GPS, GSM, Bluetooth Modules
- Serial Consoles/Debugging (e.g., TeraTerm, PuTTY)
- Serial Bootloaders
- Microcontroller ↔ PC communication

🛠️ 14. Advanced UART Features

- **DMA support for high-speed transfers**
- **Ring buffers for efficient data storage**
- **Interrupt-driven UART for non-blocking operation**
- **Multi-drop UART (9-bit mode)** for multi-device networks

💡 15. Differences from Other Protocols

Feature	UART	SPI	I2C
Clock	No	Yes (Master-driven)	Yes (Master-driven)
Wires	2 (TX, RX)	4 (MISO, MOSI, SCK, SS)	2 (SCL, SDA)
Speed	Moderate (up to Mbps)	Very High	Moderate
Complexity	Low	Medium	High
Addressing	No	No	Yes
Full-Duplex	Yes	Yes	No

UART – LPC2148

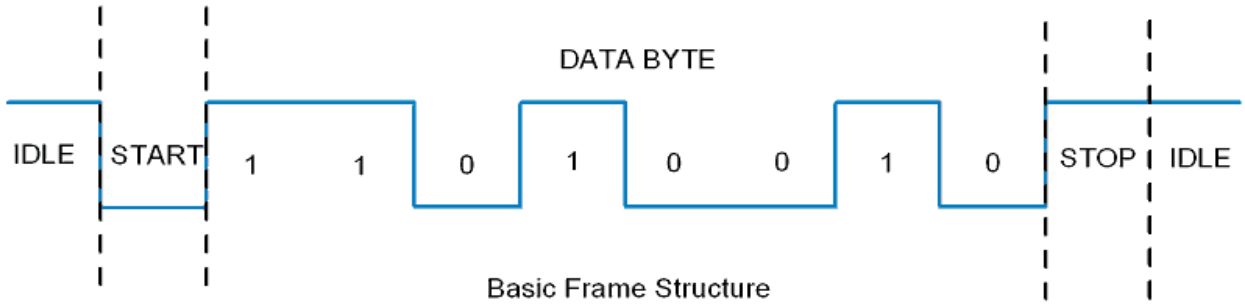
Introduction

UART (Universal Asynchronous Receiver/Transmitter) is a serial communication protocol in which data is transferred serially bit by bit at a time. Asynchronous serial communication is widely used for byte oriented transmission. In Asynchronous serial communication, a byte of data is transferred at a time.

UART serial communication protocol uses a defined frame structure for their data bytes. Frame structure in Asynchronous communication consists :

- **START bit:** It is a bit with which indicates that serial communication has started and it is always low.
- **Data bits packet**: Data bits can be packets of 5 to 9 bits. Normally we use 8 bit data packet, which is always sent after the START bit.

STOP bit: This usually is one or two bits in length. It is sent after data bits packet to indicate the end of frame. Stop bit is always logic high.



UART Frame structure

Usually, an asynchronous serial communication frame consists of a START bit (1 bit) followed by a data byte (8 bits) and then a STOP bit (1 bit), which forms a 10-bit frame as shown in the figure above. The frame can also consist of 2 STOP bits instead of a single bit, and there can also be a PARITY bit after the STOP bit.

LPC2148 UART

LPC2148 has two inbuilt UARTs available i.e. UART0&UART1. So, we can connect two UART enabled devices (GSM module, GPS module, Bluetooth module etc.) with LPC2148 at a time. UART0 and UART1 are identical other than the fact that UART1 has modem interface included.

Features of UART0

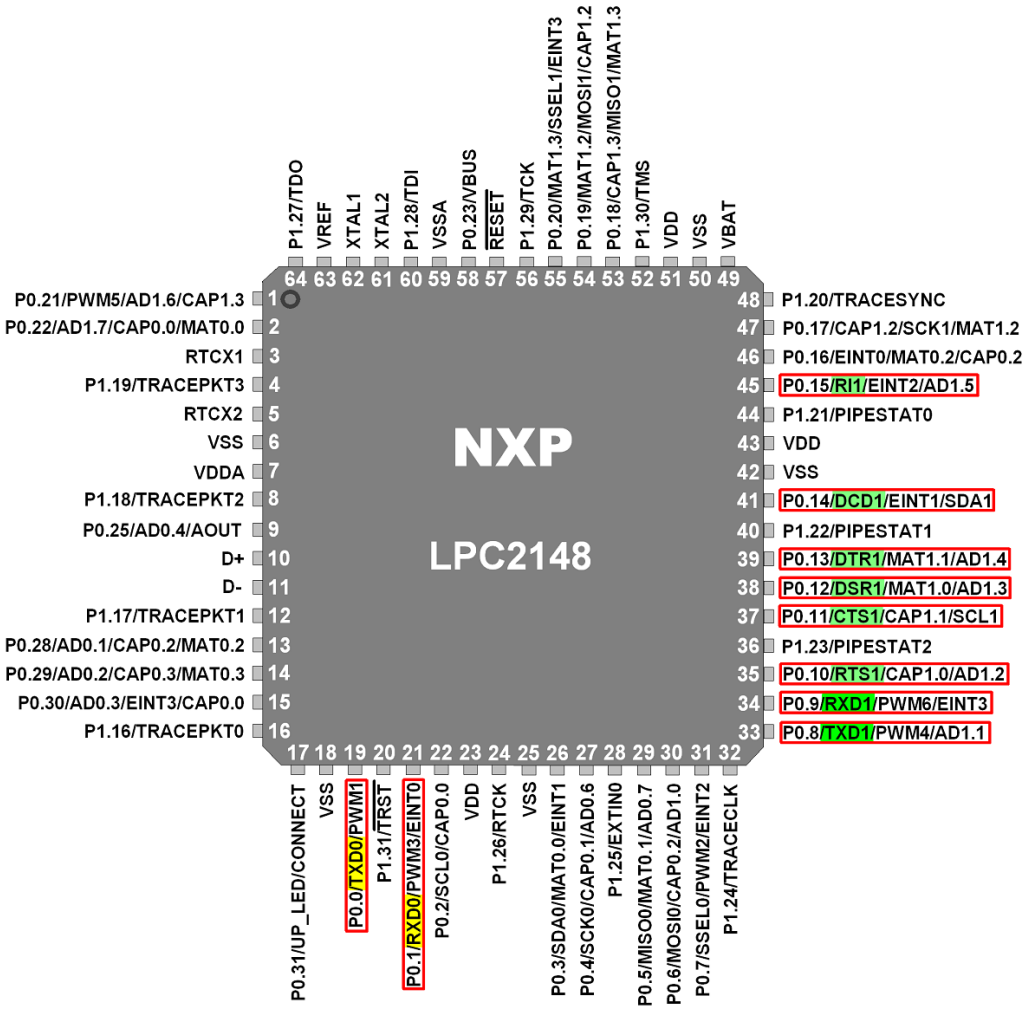
- 16 byte Receive and Transmit FIFOs
- Built-in fractional baud rate generator with autobauding capabilities

- Software flow control through TXEN bit in Transmit Enable Register

Features of UART1

- 16 byte Receive and Transmit FIFOs
- Built-in fractional baud rate generator with autobauding capabilities
- Software and hardware flow control implementation possible
- Standard modem interface signals included with flow control (auto-CTS/RTS) fully supported in hardware

LPC2148 UART Pins



LPC2148 UART Pins

UART0 :

- **TXD0** (Output pin): Serial Transmit data pin.
- **RXD0** (Input pin): Serial Receive data pin.

UART1 :

- **TXD1** (Output pin): Serial Transmit data pin.
- **RXD1** (Input pin): Serial Receive data pin.
- **RTS1** (Output pin): Request To Send signal pin. Active low signal indicates that the UART1 would like to transmit data to the external modem.
- **CTS1** (Input pin): Clear To Send signal pin. Active low signal indicates if the external modem is ready to accept transmitted data via TXD1 from the UART1.
- **DSR1** (Input pin): Data Set Ready signal pin. Active low signal indicates if the external modem is ready to establish a communication link with the UART1.
- **DTR1** (Output pin): Data Terminal Ready signal pin. Active low signal indicates that the UART1 is ready to establish connection with external modem.
- **DCD1** (Input pin): Data Carrier Detect signal pin. Active low signal indicates if the external modem has established a communication link with the UART1 and data may be exchanged.
- **RI1** (Input pin): Ring Indicator signal pin. Active low signal indicates that a telephone ringing signal has been detected by the modem.

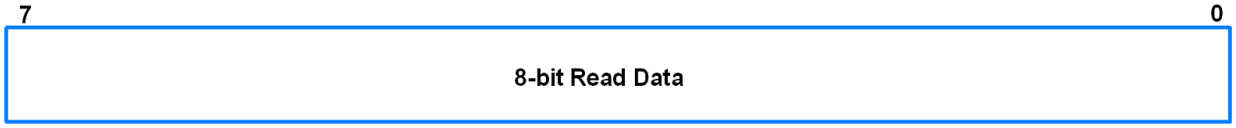
We will be seeing how to use UART0 here. UART1 can be used in a similar way by using the corresponding registers for UART1.

UART0 Registers

Table 74: UART0 Register Map

Name	Description	BIT 7	BIT 6	BIT 5	BIT 4	BIT 3	BIT 2	BIT 1	BIT 0	Access	Reset Value*	Address
U0RBR	Receiver Buffer Register	MSB			READ DATA				LSB	RO	un-defined	0xE000C000 DLAB = 0
U0THR	Transmit Holding Register	MSB			WRITE DATA				LSB	WO	NA	0xE000C000 DLAB = 0
U0IER	Interrupt Enable Register	0	0	0	0	0	Enable Rx Line Status Interrupt	Enable THRE Interrupt	Enable Rx Data Available Interrupt	R/W	0	0xE000C004 DLAB = 0
U0IIR	Interrupt ID Register	FIFOs Enabled		0	0	IIR3	IIR2	IIR1	IIR0	RO	0x01	0xE000C008
U0FCR	FIFO Control Register	Rx Trigger		Reserved		-	Tx FIFO Reset	Rx FIFO Reset	FIFO Enable	WO	0	0xE000C008
U0LCR	Line Control Register	DLAB	Set Break	Stick Parity	Even Parity Select	Parity Enable	Number of Stop Bits	Word Length Select		R/W	0	0xE000C00C
U0LSR	Line Status Register	Rx FIFO Error	TEMT	THRE	BI	FE	PE	OE	DR	RO	0x60	0xE000C014
U0SCR	Scratch Pad Register	MSB							LSB	R/W	0	0xE000C01C
U0DLL	Divisor Latch LSB	MSB							LSB	R/W	0x01	0xE000C000 DLAB = 1
U0DLM	Divisor Latch MSB	MSB							LSB	R/W	0	0xE000C004 DLAB = 1

1. **U0RBR (UART0 Receive Buffer Register)**
   - It is an 8-bit read only register.
   - This register contains the received data.
   - It contains the “oldest” received byte in the receive FIFO.
   - If the character received is less than 8 bits, the unused MSBs are padded with zeroes.
   - The Divisor Latch Access Bit (DLAB) in U0LCR must be zero in order to access the U0RBR. (DLAB = 0)



U0RBR (UART0 Receive Buffer Register)

2. **U0THR (UART0 Transmit Holding Register)**
   - It is an 8-bit write only register.
   - Data to be transmitted is written to this register.
   - It contains the “newest” received byte in the transmit FIFO.
   - The Divisor Latch Access Bit (DLAB) in U0LCR must be zero in order to access theU0THR. (DLAB = 0)

U0THR (UART0 Transmit Holding Register)

3. **U0DLL and U0DLM (UART0 Divisor Latch Registers)**
   - U0DLL is the Divisor Latch LSB.
   - U0DLM is the Divisor Latch MSB.
   - These are 8-bit read-write registers.
   - UART0 Divisor Latch holds the value by which the PCLK(Peripheral Clock) will be divided. This value must be 1/16 times the desired baud rate.
   - A 0x0000 value is treated like a 0x0001 value as division by zero is not allowed.
   - The Divisor Latch Access Bit (DLAB) in U0LCR must be one in order to access the UART0 Divisor Latches. (DLAB = 1)

U0DLL

7	0
8-bit Data	

U0DLM

4. **U0FDR (UART0 Fractional Divider Register)**
   - It is a 32-bit read write register.
   - It decides the clock pre-scalar for baud rate generation.
   - If fractional divider is active (i.e. DIVADDVAL>0) and DLM = 0, DLL must be greater than 3.

31	8	7	4	3	0
RESERVED		MULVAL		DIVADDVAL	

U0FDR (UART0 Fractional Divider Register)

- If DIVADDVAL is 0, the fractional baudrate generator will not impact the UART0 baudrate.
- Reset value of DIVADDVAL is 0.
- MULVAL must be greater than or equal to 1 for UART0 to operate properly, regardless of whether the fractional baudrate generator is used or not.
- Reset value of MULVAL is 1.
- The formula for UART0 baudrate is given below

$$UART0Baudrate = \frac{Pclk}{16 * (256 * U0DLM + U0DLL) * (1 + \frac{DIVADDVAL}{MULVAL})}$$

- MULVAL and DIVADDVAL should have values in the range of 0 to 15. If this is not ensured, the output of the fractional divider is undefined.
- The value of the U0FDR should not be modified while transmitting/receiving data. This may result in corruption of data.

5. **U0IER (UART0 Interrupt Enable Register)**
   - It is a 32-bit read-write register.
   - It is used to enable UART0 interrupt sources.
   - DLAB should be zero (DLAB = 0).

31	10	9	8	7	3	2	1	0
RESERVED		ABTO int Enable	ABEO int Enable	RESERVED		RX Status int Enable	THRE int Enable	RBR int Enable

U0IER (UART0 Interrupt Enable Register)

- **Bit 0 - RBR Interrupt Enable. It also controls the Character Receive Time-Out interrupt.**
  0 = Disable Receive Data Available interrupt
  1 = Enable Receive Data Available interrupt
- **Bit 1 - THRE Interrupt Enable**
  0 = Disable THRE interrupt
  1 = Enable THRE interrupt
- **Bit 2 - RX Line Interrupt Enable**
  0 = Disable UART0 RX line status interrupts
  1 = EnableUART0 RX line status interrupts
- **Bit 8 - ABEO Interrupt Enable**
  0 = Disable auto-baud time-out interrupt
  1 = Enable auto-baud time-out interrupt
- **Bit 9 - ABTO Interrupt Enable**
  0 = Disable end of auto-baud interrupt
  1 = Enable the end of auto-baud interrupt

6. **U0IIR (UART0 Interrupt Identification Register)**
   - It is a 32-bit read only register.

31 … 10	9	8	7 … 6	5 … 4	3 … 1	0
RESERVED	ABTO Interrupt	ABEO Interrupt	FIFO Enable	RESERVED	Interrupt Identification	Interrupt Pending

U0IIR (UART0 Interrupt Identification Register)

- It provides a status code that denotes the priority and source of a pending interrupt.
- It must be read before exiting the Interrupt Service Routine to clear the interrupt.
- **Bit 0 - Interrupt Pending**
  0 = At least one interrupt is pending
  1 = No interrupts pending
- **Bit 3:1 - Interrupt Identification**
  Identifies an interrupt corresponding to theUART0 Rx FIFO.
  011 = Receive Line Status (RLS) Interrupt
  010 = Receive Data Available (RDA) Interrupt
  110 = Character Time-out Indicator (CTI) Interrupt
  001 = THRE Interrupt
- **Bit 7:6 - FIFO Enable**These bits are equivalent to FIFO enable bit in FIFO Control Register,
  0 = If FIFOs are disabled
  1 = FIFOs are enabled

- **Bit 8 - ABEO Interrupt**
  If interrupt is enabled,
  0 = No ABEO interrupt
  1 = Auto-baud has finished successfully
- **Bit 9 - ABTO Interrupt**
  If interrupt is enabled,
  0 = No ABTO interrupt
  1 = Auto-baud has timed out

7. **U0LCR (UART0 Line Control Register)**
   - It is an 8-bit read-write register.
   - It determines the format of the data character that is to be transmitted or received.

7	6	5	4	3	2	1	0
DLAB	Set Break	Stick Parity	Even Parity Select	Parity Enable	No. of Stop Bits	Word Length Select	

U0LCR (UART0 Line Control Register)

- **Bit 1:0 - Word Length Select**
  00 = 5-bit character length
  01 = 6-bit character length
  10 = 7-bit character length
  11 = 8-bit character length
- **Bit 2 - Number of Stop Bits**
  0 = 1 stop bit
  1 = 2 stop bits
- **Bit 3 - Parity Enable**
  0 = Disable parity generation and checking
  1 = Enable parity generation and checking
- **Bit 5:4 - Parity Select**
  00 = Odd Parity
  01 = Even Parity
  10 = Forced “1” Stick Parity
  11 = Forced “0” Stick Parity
- **Bit 6 - Break Control**
  0= Disable break transmission
  1 = Enable break transmission
- **Bit 7 - Divisor Latch Access Bit (DLAB)**
  0 = Disable access to Divisor Latches
  1 = Enable access to Divisor Latches

8. U0LSR (UART0 Line Status Register)
   - It is an 8-bit read only register.

7	6	5	4	3	2	1	0
RX FIFO Error	TEMT	THRE	BI	FE	PE	OE	RDR

U0LSR (UART0 Line Status Register)

- It provides status information on UART0 RX and TX blocks.
- **Bit 0 - Receiver Data Ready**
  0 = U0RBR is empty
  1 = U0RBR contains valid data
- **Bit 1 - Overrun Error**
  0 = Overrun error status inactive
  1 = Overrun error status active
  This bit is cleared when U0LSR is read.
- **Bit 2 - Parity Error**
  0 = Parity error status inactive
  1 = Parity error status active
  This bit is cleared when U0LSR is read.
- **Bit 3 - Framing Error**
  0 = Framing error status inactive
  1 = Framing error status active
  This bit is cleared when U0LSR is read.
- **Bit 4 - Break Interrupt**
  0 = Break interrupt status inactive
  1 = Break interrupt status active
  This bit is cleared when U0LSR is read.
- **Bit 5 - Transmitter Holding Register Empty**
  0 = U0THR has valid data
  1 = U0THR empty
- **Bit 6 - Transmitter Empty**
  0 = U0THR and/or U0TSR contains valid data
  1 = U0THR and U0TSR empty
- **Bit 7 - Error in RX FIFO (RXFE)**
  0 = U0RBR contains no UART0 RX errors
  1 = U0RBR contains at least one UART0 RX error
  This bit is cleared when U0LSR is read.

9. **U0TER (UART0 Transmit Enable Register)**
   - It is an 8-bit read-write register.

7	6	5	4	3	2	1	0
TXEN	—	—	—	—	—	—	—

U0TER (UART0 Transmit Enable Register)

- The U0TER enables implementation of software flow control. When TXEn=1, UART0 transmitter will keep sending data as long as they are available. As soon as TXEn becomes 0, UART0 transmission will stop.
- Software implementing software-handshaking can clear this bit when it receives an XOFF character (DC3). Software can set this bit again when it receives an XON (DC1) character.
- **Bit 7 : TXEN**
  0 = Transmission disabled
  1 = Transmission enabled
- If this bit is cleared to 0 while a character is being sent, the transmission of that character is completed, but no further characters are sent until this bit is set again.

Programming of UART0

1. **Initialization of UART0**
   - Configure P0.0 and P0.1 as TXD0 and RXD0 by writing 01 to the corresponding bits in PINSEL0.
   - Using U0LCR register, make DLAB = 1. Also, select 8-bit character length and 1 stop bit.
   - Set appropriate values in U0DLL and U0DLM depending on the PCLK value and the baud rate desired. Fractional divider can also be used to get different values of baudrate.
     - Example,

     PCLK = 15MHz. For baud rate 9600, without using fractional divider register, from the baud rate formula, we have,

$$9600 = \frac{15000000}{16 * (256 * U0DLM + U0DLL)} * \frac{MulVal}{MulVal + DivAddVal}$$

On reset, MulVal = 1 and DivAddVal = 0 in the Fractional Divider Register.

Hence, $$(256xU0DLM + U0DLL) = \frac{15000000}{16 * 9600}$$

(256 x U0DLM + U0DLL) = 97.65

We can consider it to be 98 or 97. It will make the baud rate slightly less or more than 9600. This small change is tolerable. We will consider 97.

Since 97 is less than 256 and register values cannot contain fractions, we will take U0DLM = 0. This will give U0DLM = 97.

- Make DLA = 0 using U0LCR register.

```
void UART0_init(void)
{
	PINSEL0 = PINSEL0 | 0x00000005;	/* Enable UART0 Rx0 and Tx0 pins of UART0 */
	U0LCR = 0x83;	/* DLAB = 1, 1 stop bit, 8-bit character length */
	U0DLM = 0x00;	/* For baud rate of 9600 with Pclk = 15MHz */
	U0DLL = 0x61;		/* We get these values of U0DLL and U0DLM from formula */
	U0LCR = 0x03;	/* DLAB = 0 */
}
```

2. **Receiving character**
   - Monitor the RDR bit in U0LSR register to see if valid data is available in U0RBR register.

```
unsigned char UART0_RxChar(void) /*A function to receive a byte on UART0 */
{
	while( (U0LSR & 0x01) == 0); /*Wait till RDR bit becomes 1 which tells that receiver
contains valid data */
		return U0RBR;
}
```

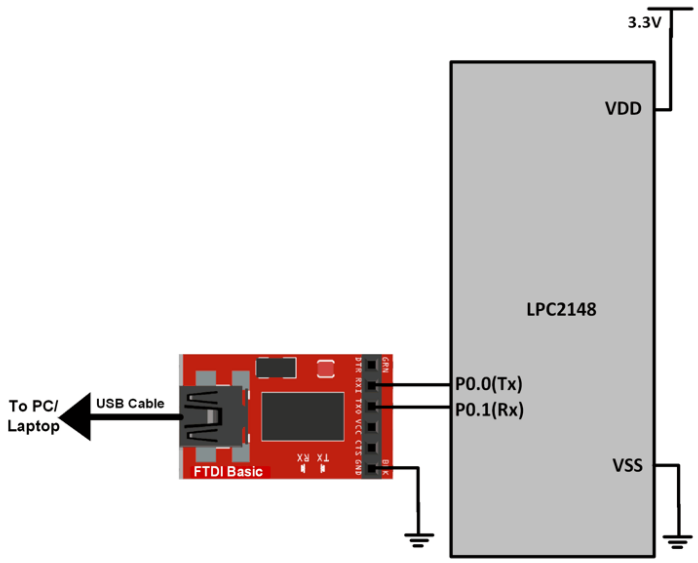
3. **Transmitting character**
   - Monitor the THRE bit in U0LSR register. When this bit becomes 1, it indicates that U0THR register is empty and the transmission is completed.

```
void UART0_TxChar(char ch) /*A function to send a byte on UART0 */
{
	U0THR = ch;
	while( (U0LSR & 0x40) == 0 );/* Wait till THRE bit becomes 1 which tells that
transmission is completed */
}
```

Example

Program for echo test. LPC 2148 will receive a character through PC terminal and will transmit the same character back to the PC terminal.

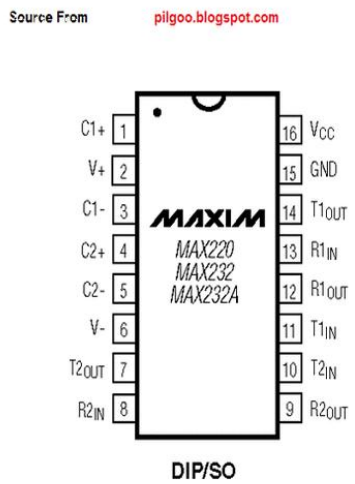
Software like Real Term, Putty can be used for this on PC.



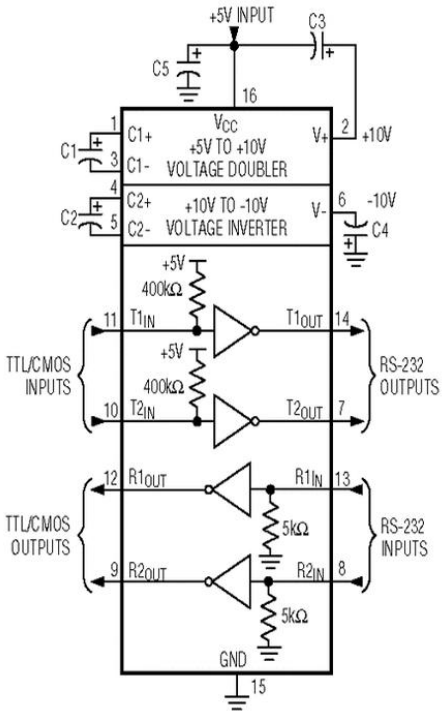
Program

```
#include <lpc214x.h>
#include <stdint.h>
#include "UART.h"

int main(void)
{
        char receive;
        UART0_init();
        while(1)
        {
                receive = UART0_RxChar();
                UART0_SendString("Received:");
                UART0_TxChar(receive);
                UART0_SendString("\r\n");
        }
}
```

MAX232 (for UART level shifting)



CAPACITANCE (µF)					
DEVICE	C1	C2	C3	C4	C5
MAX220	0.047	0.33	0.33	0.33	0.33
MAX232	1.0	1.0	1.0	1.0	1.0
MAX232A	0.1	0.1	0.1	0.1	0.1



1. Why MAX232 is Needed?

UART on microcontrollers uses **TTL/CMOS voltage levels**:

- Logic HIGH = 3.3V or 5V
- Logic LOW = 0V

Whereas **RS-232 (used in PCs, serial ports)** uses:

- Logic HIGH = **-3V to -15V**
- Logic LOW = **+3V to +15V**

These are **inverted** and **higher voltage levels**, so direct connection will damage your microcontroller.

Solution: Use **MAX232**, which converts TTL ⇌ RS-232 voltage levels.

---

2. MAX232 Block Diagram Overview

Internally, MAX232 consists of:

- **Two transmitters (line drivers)**
- **Two receivers (line receivers)**
- **Voltage level shifter**
- **Charge pump** (generates ±10V from a single 5V supply using external capacitors)

Microcontroller		MAX232		RS232 Device
TX (TTL, 5V)	--->	T1IN T1OUT	--->	RX (RS232, ±12V)
RX (TTL, 5V)	<---	R1OUT R1IN	<---	TX (RS232, ±12V)

What is MAX232 IC?

MAX232 IC is widely used for serial communication among Microcontrollers & PC. The main function of this integrated circuit is to change the logic levels of TTL/CMOS to RS232 throughout the serial communication process.

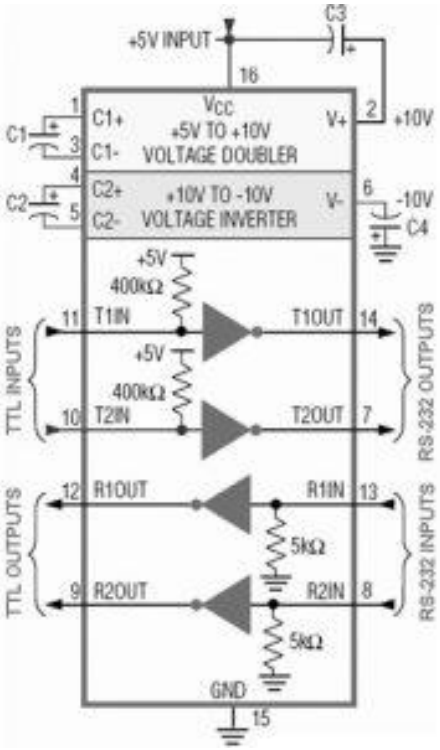
Generally, a microcontroller works at TTL of about 0 to 5V while a computer operates on the standards of RS232 like -25 to +25V. So interfacing a computer with a microcontroller is not possible without a MAX232 IC. For example; if you want to connect a PIC microcontroller to the serial pin of a laptop, then you must place MAX232IC among them.

MAX232 IC

Here, laptop serial ports like 9 pin work through RS232 Voltage Logic whereas the serial ports of Microcontroller like Tx & RX pins work on transistor Logic. The equivalent RS232 Converters of MAX232 are RS232 module, MC1488, RS232 adapter cable & FT232RL

Structure of IC

MAX232 IC includes three main blocks shown in the following diagram. It operates with 5 volts of power supply so that it is compatible with almost all embedded systems. The voltage doubler is the first block and in this IC, a switched capacitor method is utilized to make the voltage twice. Whenever the voltage is doubled then the next block will convert the voltage to +10 & -10. And the final block includes two transmitters & two receivers which convert the levels of voltage.



MAX232 IC Structure

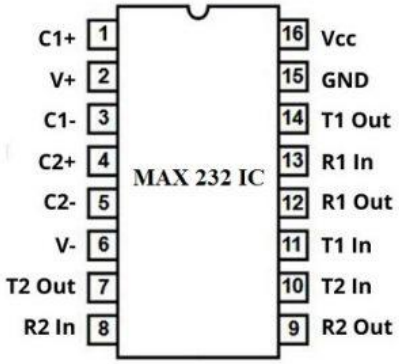
Types of MAX232 IC

MAX232 ICs are available in two types like MAX232N & MAX232D

- In MAX232N IC, "N" signifies the package style of PDIP and this package is simple to sell & most extensively used.
- In MAX232D IC, "D" signifies the SOIC package & it is hard to sell but requires a skilled professional to be utilized properly.

MAX232 IC Pin Configuration

MAX232 IC includes 16 pins where each pin is described below. This IC needs four capacitors externally for its proper design. These four capacitors range from 8 to10



MAX232 IC Pin Configuration

uf with up to 16 volts. Here each pin with its function is described below.

- Pin-1(C1+): The positive terminal of a capacitor is connected to this pin
- Pin-2(Vs+): The capacitor's positive leg is connected to it by grounding the negative leg.
- Pin-3(C1-): The negative pin of the capacitor is connected to this pin and the positive pin is connected to pin1
- Pin-4(C2+): The positive terminal of a capacitor is connected to this pin
- Pin-5(C2-): The negative terminal of a capacitor is connected where the positive terminal is connected to Pin4.
- Pin-6(Vs-): The negative terminal of a capacitor is connected to this pin & 5 volts is provided to the positive terminal of the capacitor.
- Pin-7(T2OUT): It provides the converted TTL signal in the form of RS-232. Here TTL signal can be obtained by T2IN Pin from the microcontroller and this pin is connected to Pin2 of DB-9 port of your computer like Rxd.
- Pin-8(R2IN): This Pin gets the signal of RS-232 like an input & provides the changed signal in the form of TTL on the R2OUT pin. This pin is connected to DB9 Port's Txd pin, which is Pin3.
- Pin-9(R2OUT): It provides the signal changed within TTL form. The signal is received from Pc at R1In Pin. Connect this pin to your module (TTL) Rxd pin which receives the signal.
- Pin-10(T2IN): This pin gets transmitted signal from microcontroller & gives the changed RS-232 signal over T2OUT pin. Here, the signal can be transmitted from the microcontroller serial port's txd pin. This pin can be connected to your Txd pin of the module.
- Pin-11(T1IN): This pin works like a T2IN.
- Pin-12(R1OUT): This pin works like an R2OUT.
- Pin-13(R1IN): This pin works like an R2IN.
- Pin-14(T1OUT): This pin works like aT2OUT.
- Pin-15(GND): This pin is a GND pin.
- Pin-16(VCC): This pin is a voltage supply pin where 5V is provided to this pin.

MAX232 IC Working

We know that the main function of Max 232 IC is to change the signal of TTL logic level into its corresponding RS-232c level signal & then Rs-232c level signal to its corresponding TTL level signal.

This IC is very useful while making a connection between two devices to transfer the data.
For instance, almost all the microcontroller's functions are on the waveform of TTL logic. These microcontrollers include a UART that can be used to transmit & receive serial data because they

work over the TTL level thus they send & receive composed TTL waves from data while typical PC functions on the level of RS-232 waveform.

If we want to send the data from the microcontroller to a personal computer then we require changing data to Rs 232 level from TTL. So MAX232 IC is the best solution for converting this data. This IC works up to 120 Kbits per sec and it includes two receivers & two driver channels. The highest distance it can transmit data ranges from 12 – 15 meters. So this IC is applicable in terminals, modems, computers, scanners, printers, fax machines, etc.

Features

The main **features of MAX232 IC** include the following.

- It is available in 16-pin SOIC, PDIP & SO packages
- Simple to Setup & Initialize
- Operating Speed is120kbit/s
- Operating Current is 8mA
- Input Levels ±30V
- Conversion IC from TTL or CMOS to RS232
- At a time it supports two conversions like dual drivers & receivers
- The simple way to change the TIA/ EIA 32 to CMOS/TTL is simply a 5V power supply.
- A 1.0uF capacitor is used to resolve the issue of power decimation.
- The voltage levels of RS232 devices are different in some cases
- Its highest working temperature is 150o C.

Packages, Dimensions & Ratings

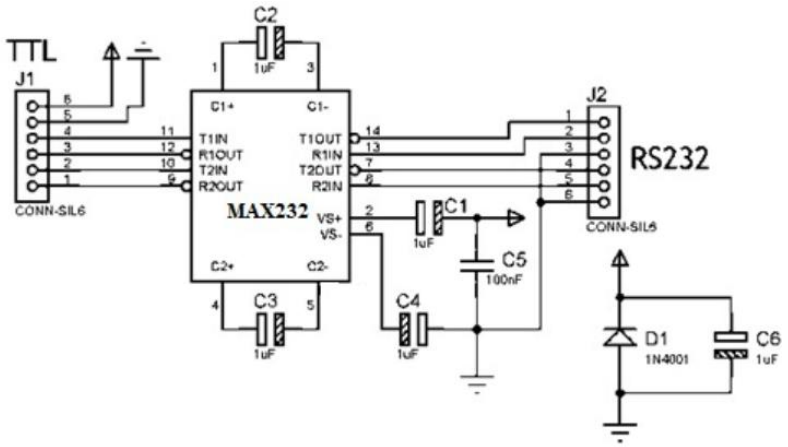
The **packages of MAX232 IC** include the following.

- The packages of MAX-232 IC are SOIC (16), SOP (16) & PDIP (16)
- The package dimensions of MAX 232 IC include the following.
- The dimension of the SOIC (16) package is 9.90 X 3.91 mm
- The dimension of the PDIP (16) package is 19.03 X 6.35 mm
- The dimension of the SOP (16) package is 10.03 X 5.3 mm
- The ratings of MAX232 IC are the current, voltage & power ratings of IC that show its power requirement so that how much current & voltage is used for its operation can be known.
- Input voltage ranges from -0.3 to 6v
- Output voltage ranges from 0.3 to +0.3
- The operating temperature is 150oC

MAX232 IC Circuit Diagram

The circuit diagram of MAX32IC for level shifting of RS232-TTL is shown below. The following circuit is applicable for microcontrollers. This circuit mainly permits the levels of shifting in between TTL & RS232.

The circuit can be built with the MAX232 IC and four exterior capacitors for producing maximum voltage internally. This IC has different clones which have similar functionality but are usually not expensive. It's achievable to exchange them without any problem.



Level Shifting from RS232 to TTL

If the signal of TTL is low voltage within some modern microcontrollers, then it is essential to swap the MAX232 IC in support of the MAX3232, here both are similar except it functions from approximately 3V up. If the IC is not accessible, it might be feasible to utilize a usual MAX232 through a less voltage supply, however, the output might not be within the limits described through the standard of RS232.

The highest suggested distance is approximately 10 meters at 115200 bps. If maximum distances are required, it is achievable to utilize other protocols or standards like RS485 that employ differential signaling, thus the noise doesn't change them a lot. Usually, this technique needs more wires.

Where to Use MAX232IC/Applications

The applications of MAX232 IC include the following.

- It is used to interface microcontroller through PC
- Computers
- Used to convert from TTL or CMOS logic to RS232
- Modem.
- Terminals
- Utilized within RS232 cables
- Portable Computing

- Transactions of Interface
- RS232 System Powered with a Battery
- Fewer Power Modes

DB-9 Cable (db9 rs232 9pin serial cable male to female)

What is it?

A **DB-9 cable** is a **9-pin D-sub connector cable** used for **RS-232 serial communication**, typically for connecting **computers, microcontrollers, modems, or other embedded systems**.

- **Male to Female**:
  - **Male connector** has 9 pins.
  - **Female connector** has 9 holes.
  - This configuration is used to extend or connect devices like PCs to peripherals (e.g., modems, serial-enabled devices).

RS-232 Overview:

- **Standard**: RS-232 (Recommended Standard 232)
- **Type**: Asynchronous serial communication
- **Voltage Levels**:
  - Logic 1 = -3V to -15V
  - Logic 0 = +3V to +15V

RS-232 is not TTL compatible directly; a level shifter (like **MAX232**) is used in microcontroller applications.

DB-9 Pinout and Description

Pin No	Name	Direction	Description
1	DCD	Input	Data Carrier Detect
2	RXD	Input	Receive Data
3	TXD	Output	Transmit Data
4	DTR	Output	Data Terminal Ready

5	GND	—	Signal Ground
6	DSR	Input	Data Set Ready
7	RTS	Output	Request to Send
8	CTS	Input	Clear to Send
9	RI	Input	Ring Indicator

Common pins for basic serial communication:

- **Pin 2 (RXD)**
- **Pin 3 (TXD)**
- **Pin 5 (GND)**

🧠 **Applications:**

- Serial communication with PCs, microcontrollers, and PLCs
- Interfacing GSM/GPS modules
- Serial debug console for embedded boards
- Connecting to legacy devices or industrial systems

🧪 **Cable Types:**

- **Null Modem Cable (Crossed TX and RX)**: Used to connect two DTEs (like PC to PC)
- **Straight Cable**: Connects DTE (PC) to DCE (modem or microcontroller with MAX232)

⚠️ **Notes for Embedded Systems:**

- Microcontrollers use **TTL-level UART**, not RS-232 directly.
- Use **MAX232 or equivalent IC** to shift levels from RS-232 (±12V) to TTL (0V/5V or 0V/3.3V).

🧰 **Example Use Case:**

PC ↔ MAX232 ↔ Microcontroller (LPC2148/AVR/8051)

DB9 Male connects to PC, Female connects to RS232-level converter (MAX232), which talks to the UART of the microcontroller.

What is RS-232?

RS-232 (Recommended Standard 232) is a **serial communication protocol** used for data transmission between **Data Terminal Equipment (DTE)** (like a computer) and **Data Communication Equipment (DCE)** (like a modem).
It's an **asynchronous, point-to-point, full-duplex protocol** using voltage level signals.

⚙️ Key Characteristics

Feature	Description
Communication Type	Serial (bit-by-bit transmission)
Standard	RS-232 (EIA/TIA)
Mode	Asynchronous (no clock signal shared)
Voltage Levels	Logic '1': -3V to -15V Logic '0': +3V to +15V
Max Cable Length	~15 meters (at 19.2 kbps), shorter for higher speeds
Speed	Typically up to 115.2 kbps (some implementations faster)
Connector	DB-9 (9-pin) or DB-25 (25-pin)
Signal Type	Unbalanced

🧩 Signal Description (DB9 Pins)

Pin	Signal Name	Direction	Function
1	DCD	IN	Carrier detect
2	RXD	IN	Receive Data
3	TXD	OUT	Transmit Data
4	DTR	OUT	Data Terminal Ready
5	GND	—	Signal Ground
6	DSR	IN	Data Set Ready
7	RTS	OUT	Request to Send
8	CTS	IN	Clear to Send
9	RI	IN	Ring Indicator

For **simple serial communication**, only **TXD, RXD, and GND** are essential.

🧠 Working Mechanism

1. **Asynchronous Transfer**:
   - No separate clock line.
   - Timing is maintained using a **baud rate** agreed by both ends.
2. **Framing Format** (per character):

| Start Bit | Data Bits (5–8) | Parity Bit (optional) | Stop Bit(s) |

- **Start Bit**: 1 bit (always logic 0, i.e., + voltage)
- **Data Bits**: Typically 8 bits
- **Parity Bit**: Optional (for error checking)
- **Stop Bit(s)**: 1 or 2 bits (logic 1, i.e., - voltage)

📊 RS-232 Data Frame Example

Bit	Logic Level	Voltage Range (RS-232)
Idle	1	-3V to -15V
Start	0	+3V to +15V
Data Bits	0/1	±3V to ±15V
Parity	0/1	Optional
Stop	1	-3V to -15V

🔌 Common Baud Rates

Baud Rate	Description
9600	Common standard
19200	Faster
38400	Fastest standard
115200	High-speed serial

Both devices must be configured with **same baud rate, data bits, parity, and stop bits**.

🧩 Application in Embedded Systems

Example: Microcontroller to PC Serial Communication

- Microcontroller UART (TTL) → **MAX232 (Level Shifter) → DB9 (RS-232) → PC**

Use Cases:

- Serial terminal debugging
- Firmware uploads
- GPS/GSM interfacing
- HMI or serial sensors
- Interfacing with SCADA/PLC

🔁 RS-232 vs TTL UART

Feature	RS-232	TTL UART (Microcontroller)
Voltage Level	±3V to ±15V	0V to 5V or 3.3V
Inverted Logic	Yes	No
Interface	DB9, DB25	Tx, Rx pins

Converter Req	Yes (e.g., MAX232)	No

🧰 RS-232 Advantages

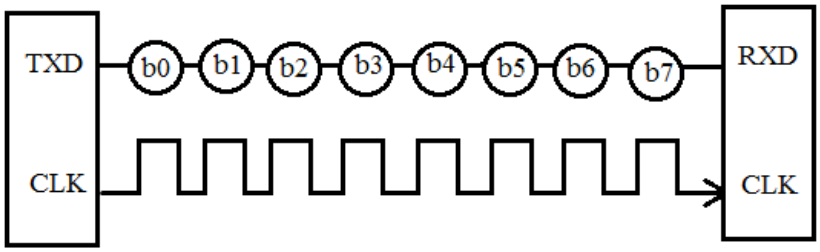
- Simple and reliable for point-to-point links
- No clock line needed
- Widely supported (legacy systems)

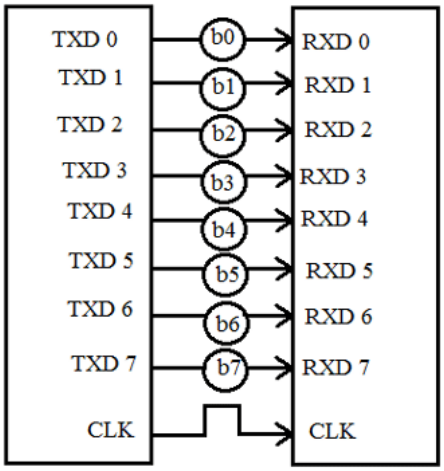
⚠️ Limitations

- Short range
- Low speed compared to USB, Ethernet
- Only one-to-one communication (not multi-drop like RS-485)

What is a serial communication?

In telecommunication, the process of sending data sequentially over a computer bus is called as **serial communication**, which means the data will be transmitted bit by bit. While in parallel communication the data is transmitted in a byte (8 bit) or character on several data lines or buses at a time. Serial communication is slower than parallel communication but used for long data transmission due to lower cost and practical reasons.





Example to understand:
Serial communication – you are shooting a target using machine guns, where bullets reach one by one to the target.
Parallel communication- you are shooting a target using a shotgun, where many number of the bullets reach at the same time.

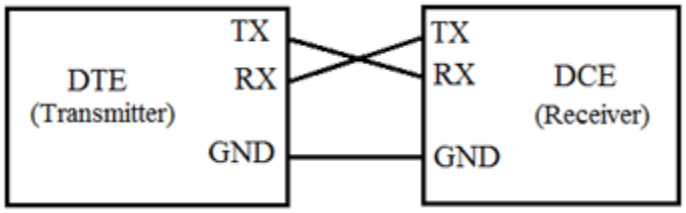
Modes of Data Transfer in Serial Communication:

- **Asynchronous Data Transfer –** The mode in which the bits of data are not synchronized by a clock pulse. Clock pulse is a signal used for synchronization of operation in an electronic system.
- **Synchronous Data Transfer –** The mode in which the bits of data are synchronized by a clock pulse.

Characteristics of Serial Communication:

- **Baud rate** is used to measure the speed of transmission. It is described as the **number of bits passing in one second.** For example, if the baud rate is 200 then 200 bits per Sec passed. In telephone lines, the baud rates will be **14400, 28800 and 33600**.
- **Stop Bits** are used for a single packet to stop the transmission which is denoted as "T". Some typical values are **1, 1.5 & 2 bits**.
- **Parity Bit** is the simplest form of checking the errors. There are of four kinds, i.e., even odd, marked and spaced. **For example**, If 011 is a number the parity bit=0, i.e., even parity and the parity=1, i.e., odd parity.

What is RS232?
RS232C "Recommended Standard 232C" *is the recent version of Standard 25 pin whereas,* ***RS232D*** *which is of 22 pins. In new PC's male D-type which is of 9 pins.*
RS232 is a standard protocol used for serial communication, it is used for connecting computer and its peripheral devices to allow serial data exchange between them. As it obtains the voltage for the path used for the data exchange between the devices. It is used in serial communication up to 50 feet with the rate of 1.492kbps. As EIA defines, the RS232 is used for connecting **Data Transmission Equipment (DTE)** and **Data Communication Equipment (DCE)**.



Universal Asynchronous Data Receiver &Transmitter (UART) used in connection with RS232 for transferring data between printer and computer. The microcontrollers are not able to

handle such kind of voltage levels, connectors are connected between RS232 signals. These connectors are known as the **DB-9 Connector** as a serial port and they are of two type's **Male connector (DTE) & Female connector (DCE).**

Electrical Specifications

Let us discuss the electrical specifications of RS232 given below:

- **Voltage Levels:** RS232 also used as ground & 5V level. Binary 0 works with voltages up to +5V to +15Vdc. It is called as 'ON' or spacing (high voltage level) whereas Binary 1 works with voltages up to -5V to -15Vdc. It is called as 'OFF' or marking (low voltage level).
- **Received signal voltage level:** Binary 0 works on the received signal voltages up to +3V to +13 Vdc & Binary 1 works with voltages up to -3V to -13 Vdc.
- **Line Impedances:** The impedance of wires is up to 3 ohms to 7 ohms & the maximum cable length are 15 meters, but new maximum length in terms of capacitance per unit length.
- **Operation Voltage:** The operation voltage will be 250v AC max.
- **Current Rating:** The current rating will be 3 Amps max.
- **Dielectric withstanding voltage:** 1000 VAC min.
- **Slew Rate:** The rate of change of signal levels is termed as Slew Rate. With its slew rate is up to 30 V/microsecond and the maximum bitrate will be 20 kbps.

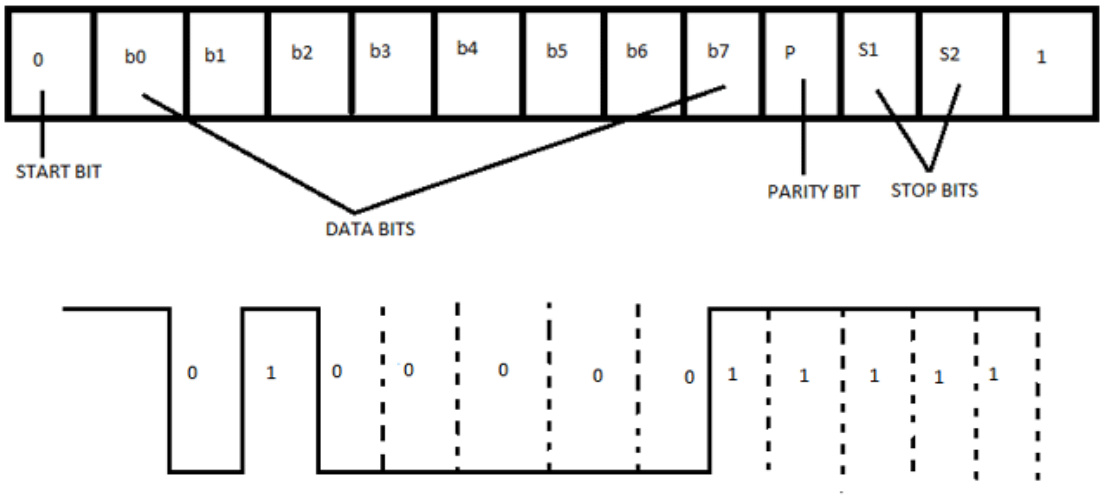
The ratings and specification changes with the change in equipment model.

How RS232 Works?

RS232 works on the two-way communication that exchanges data to one another. There are two devices connected to each other, **(DTE) Data Transmission Equipment& (DCE) Data Communication Equipment** which has the pins like **TXD, RXD, and RTS& CTS**. Now, from **DTE** source, the **RTS** generates the *request to send* the data. Then from the other side **DCE**, the **CTS,** clears the path for receiving the data. After clearing a path, it will give a signal to **RTS** of the **DTE** source to send the signal. Then the bits are transmitted from **DTE** to **DCE**. Now again from **DCE** source, the request can be generated by **RTS** and **CTS** of **DTE** sources clears the path for receiving the data and gives a signal to send the data. This is the whole process through which data transmission takes place.

TXD	TRANSMITTER
RXD	RECEIVER
RTS	REQUEST TO SEND
CTS	CLEAR TO SEND
GND	GROUND

For example: The signals set to logic 1, i.e., -12V. The data transmission starts from next bit and to inform this, DTE sends start bit to DCE. The start bit is always '0', i.e., +12 V & next 5 to 9 characters is data bits. If we use parity bit, then 8 bits data can be transmitted whereas if parity doesn't use, then 9 bits are being transmitted. The stop bits are sent by the transmitter whose values are 1, 1.5 or 2 bits after the data transmission.

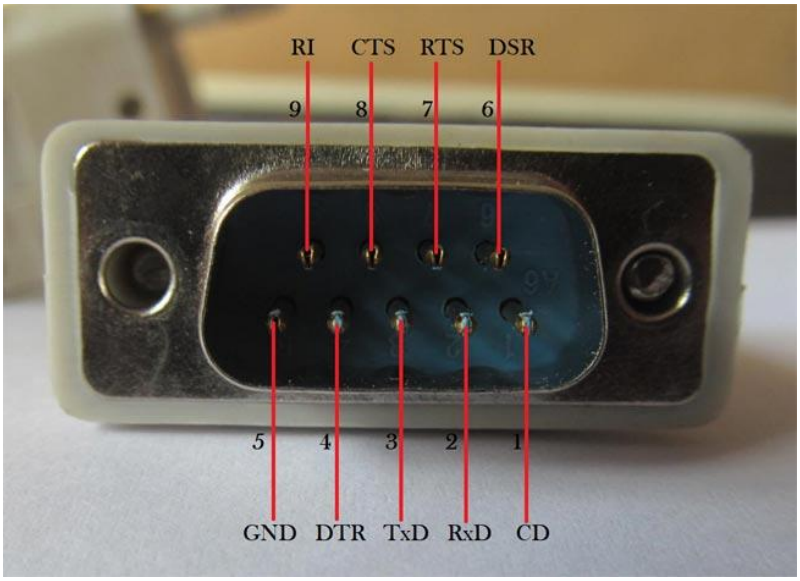


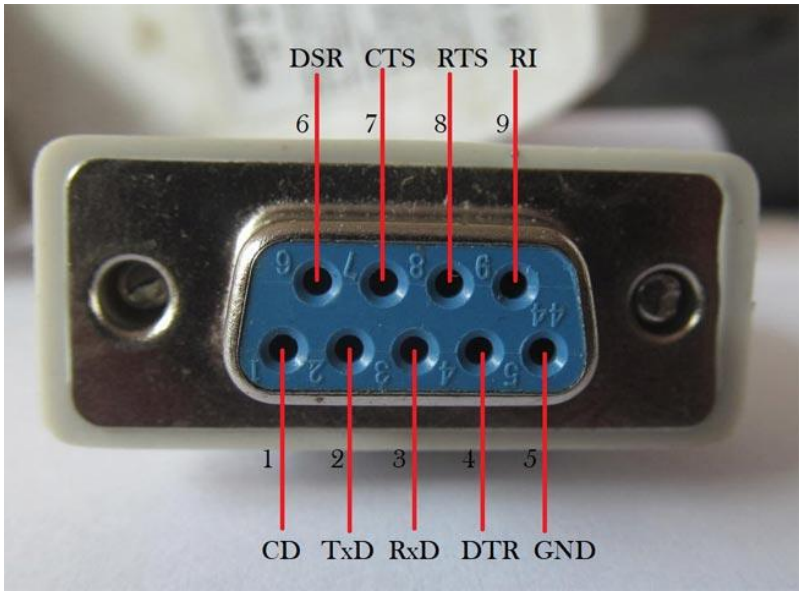
Mechanical Specification

For mechanical specifications, we have to study about two types of connectors that is **DB-25** and **DB-9**. In DB-25, there are 25 pins available which are used for many of the applications, but some of the applications didn't use the whole 25 pins. So, the 9 pin connector is made for the convenience of the devices and equipments.

Now, here we are discussing the **DB-9** pin connector which is used for connection between microcontrollers and connector. These are of two types: **Male Connector (DTE)** & **Female Connector (DCE)**. There are 5 pins on the top row and 4 pins in the bottom row. It is often called **DE-9 or D-type connector.**

Pin Structure of DB-9 Connector:





Pin Description DB-9 Connector:

PIN No.	Pin Name	Pin Description
1	CD (Carrier Detect)	Incoming signal from DCE
2	RD (Receive Data)	Receives incoming data from DTE
3	TD (Transmit Data)	Send outgoing data to DCE
4	DTR (Data Terminal Ready)	Outgoing handshaking signal

5	GND (Signal ground)	Common reference voltage
6	DSR (Data Set Ready)	Incoming handshaking signal
7	RTS (Request to Send)	Outgoing signal for controlling flow
8	CTS (Clear to Send)	Incoming signal for controlling flow
9	RI (Ring Indicator)	Incoming signal from DCE

What is Handshaking?

How can a transmitter, transmits and the receiver receives data successfully. So, the Handshaking defines, for this reason.

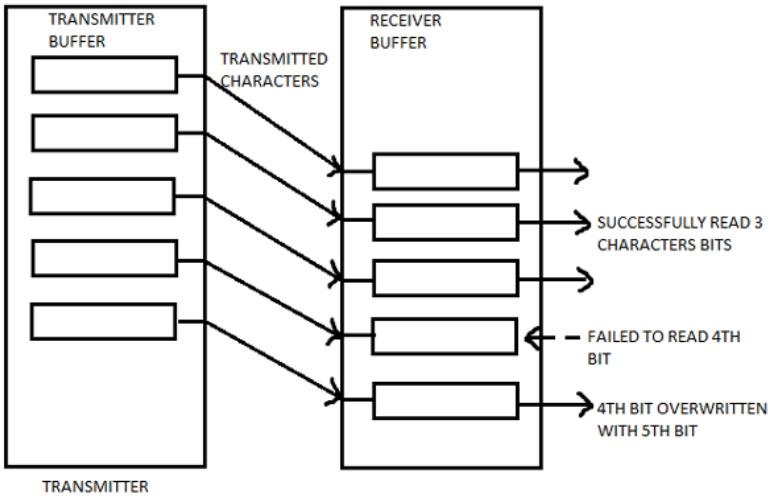
Handshaking is the process which is used to transfer the signal from DTE to DCE to make the connection before the actual transfer of data. The messaging between transmitter & receiver can be done by handshaking.

There are **3 types of handshaking processes** named as:-

No Handshaking:

If there is no handshaking, then DCE reads the already received data while DTE transmits the next data. All the received data stored in a memory location known as receiver's buffer. This buffer can only store one bit so receiver must read the memory buffer before the next bit arrives. If the receiver is not able to read the stored bit in the buffer and next bit arrives then the stored bit will be lost.

As shown in below diagram, a receiver was unable to read the 4th bit till the 5th bit arrival and this result overriding of 4th bit by 5th bit and 4th bit is lost.

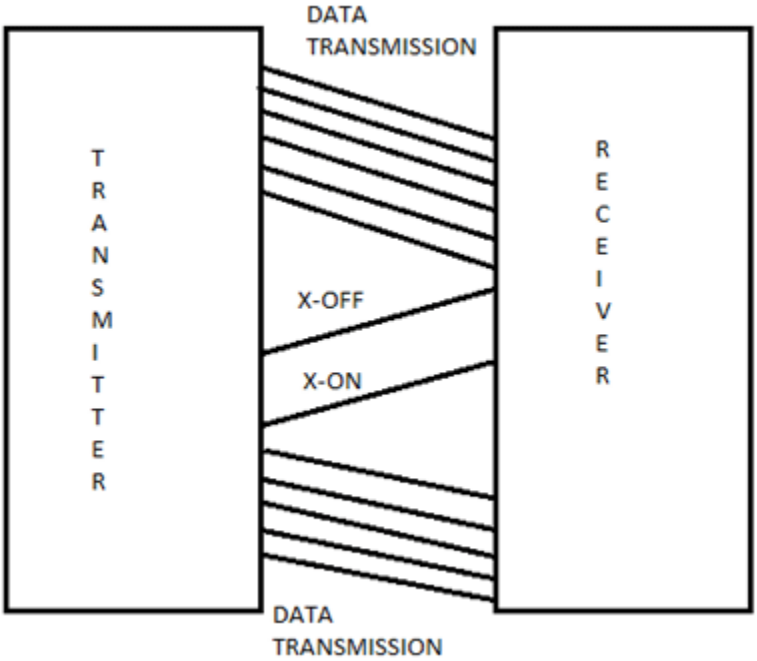


Hardware Handshaking:

- It uses specific serial ports, i.e., RTS & CTS to control data flow.
- In this process, transmitter asks the receiver that it is ready to receive data then receiver checks the buffer that it is empty, if it is empty then it will give signal to the transmitter that I am ready to receive data.
- The receiver gives the signal to transmitter not to send any data while already received data cannot be read.
- Its working process is same as above described in handshaking.

Software Handshaking:

- In this process, there are two forms, i.e., X-ON & X-OFF. Here, 'X' is the transmitter.
- X-ON is the part in which it resumes the data transmission.
- X-OFF is the part in which it pauses the data transmission.
- It is used to control the data flow and prevent loss during transmission.



Applications of RS232 Communication

- RS232 serial communication is used in old generation PCs for connecting the peripheral devices like mouse, printers, modem etc.
- Nowadays, RS232 is replaced by advanced USB.
- It is also used in PLC machines, CNC machines, and servo controllers because it is far cheaper.
- It is still used by some microcontroller boards, receipt printers, point of sale system (PoS), etc.

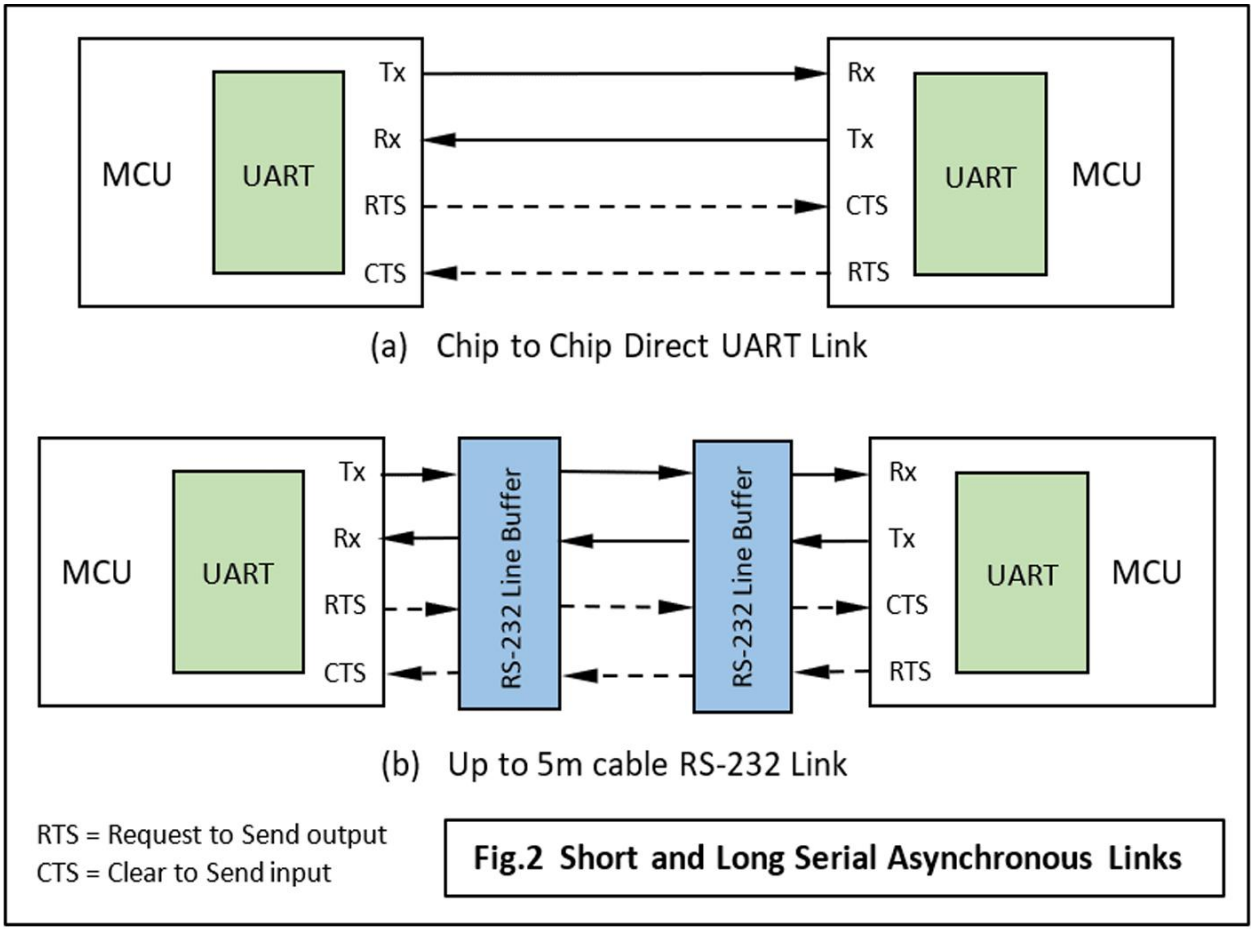


Fig.2 Short and Long Serial Asynchronous Links

(a) Chip to Chip Direct UART Link

Description:

- This setup is a **direct UART connection** between two microcontrollers.
- No voltage conversion or buffering is used — it's typically used for **short-distance communication** on the **same PCB or nearby boards**.

Connections:

- **Tx (Transmit)** of MCU1 is connected to **Rx (Receive)** of MCU2.
- **Rx** of MCU1 is connected to **Tx** of MCU2.
- Optional control lines:
  - **RTS (Request To Send)** of MCU1 connects to **CTS (Clear To Send)** of MCU2.
  - **CTS** of MCU1 connects to **RTS** of MCU2.

Key Points:

- Voltage levels are **TTL/CMOS** (0V to 3.3V/5V).
- Suitable for **short, interference-free links** (within a few inches/cm).
- **No RS-232 transceivers or buffers** are needed.

(b) Up to 5m cable RS-232 Link

Description:

- This setup allows communication over a **longer distance (up to 5 meters)** using **RS-232 standard signaling**.
- It uses **RS-232 Line Buffers** (like MAX232 ICs) between microcontrollers to convert logic levels to RS-232 voltage levels and vice versa.

Connections:

- **MCU UART → RS-232 Line Buffer → Cable → RS-232 Line Buffer → MCU UART**
- The RS-232 buffers handle voltage level shifting:
  - From TTL (0V–5V or 3.3V) to RS-232 (±3V to ±15V)
  - And vice versa for receiving.

Signal Flow:

- **Tx of MCU1 → RS-232 Buffer → RS-232 Cable → RS-232 Buffer → Rx of MCU2**
- **Rx of MCU1 ← RS-232 Buffer ← RS-232 Cable ← RS-232 Buffer ← Tx of MCU2**
- Control lines (RTS/CTS) also flow similarly through buffers.

Key Points:

- Supports **longer distances** (up to 5 meters reliably).
- Requires **external hardware (line drivers/receivers)**.
- **More immune to noise** and signal degradation.

🔁 RTS / CTS Lines (Hardware Flow Control)

Signal	Meaning	Direction
RTS	Request To Send	Output
CTS	Clear To Send (acknowledge)	Input

These lines are **optional** and used for **flow control** to avoid data loss when one device is not ready to receive data.

✅ Summary

Feature	(a) Direct UART	(b) RS-232 with Buffers
Distance	Short (cm)	Up to 5 meters
Voltage Levels	TTL (0–5V/3.3V)	RS-232 (±3V to ±15V)
Hardware Needed	None	RS-232 Line Buffers (e.g., MAX232)
Application	Same board, short range	PC-to-MCU, long cable connections
Signal Reliability	Less robust	More robust over distance

Software

Keil uVision

What is Keil µVision?
Keil µVision is an **IDE (Integrated Development Environment)** developed by **Arm Keil** specifically for embedded software development on **ARM, 8051, and Cortex-M** based microcontrollers.

◆ Key Features of Keil µVision (for Interview)

Feature	Description
IDE Integration	Combines editor, compiler, assembler, linker, debugger, and simulator in one interface.
Compiler	Uses **Arm Compiler** or **C51 Compiler** to build code for various MCUs.
Device Support	Supports a wide range of microcontrollers like **8051, LPC2148 (ARM7TDMI), STM32 (Cortex-M)**.
Simulator	Allows simulating code without hardware (register-level simulation).
Debugger	Real-time debugging with **breakpoints, watch windows, memory view, and register access**.
Flash Programming	Supports programming microcontroller flash memory using **ULINK, J-Link, ST-Link**, etc.
Project Management	Provides an intuitive project structure with groups, files, and configuration settings.

◆ µVision Components (In Depth)

1. Project Manager

- Creates and manages multiple source files (.c, .h, .s).
- Allows **grouping** files logically (like LCD, UART, etc.).
- Handles compiler options, device selection, and toolchain settings.

2. Text Editor

- Smart code editor with syntax highlighting, auto-complete, and error detection.
- Supports C, C++, Assembly, linker scripts.

3. Build Tools

- Includes:
  - **Compiler (C51 or ArmCC)**: Compiles C files.
  - **Assembler**: For .s (assembly) files.
  - **Linker/Locator**: Links compiled objects and places them in MCU memory.
  - Output formats: .hex, .elf, .axf.

4. Debugger/Simulator

- Step-by-step execution.
- **Run**, **Stop**, **Step Into/Over**, **Reset** options.
- Views:
  - **Memory Window**: See RAM/ROM contents.
  - **Watch Window**: Track variables live.
  - **Call Stack + Locals**: For function-level debugging.
  - **Peripherals View**: Check registers like UART0, GPIO, TIMER0.
- Can simulate **serial input/output (via UART)**.

5. Flash Utility

- Programs the target microcontroller using ULINK or USB to Serial interfaces.
- Supports various flash devices.
- **ISP (In-System Programming)** and **IAP (In-Application Programming)** support.

🔹 Build Process in Keil

1. **Write Code** (main.c, uart.c, etc.)
2. **Create Project**
3. **Add files to project**
4. **Select Target Device** (e.g., LPC2148)
5. **Configure Options for Target** (Crystal freq, memory map, output format)
6. **Build Project** (compiles to .hex or .axf)
7. **Flash to MCU** via Flash Utility
8. **Debug and Verify**

🔹 Real-Time Applications

Use Case	Description
UART Communication	Debug UART transmit/receive registers (THR, RBR, LSR)
GPIO Toggling	Simulate LED blinking and check I/O port registers
Interrupt Handling	Test ISR triggering for timers or UART
Timer Counters	Debug TMR0/TMR1 behavior and match registers

Memory Map Access	Modify and monitor memory at specific locations

◆ Interview Tips

Question	Suggested Answer
Why use Keil uVision?	Because it's optimized for 8051 and ARM MCUs, provides simulation, easy flash programming, and a rich debugging interface.
What is the role of a linker in Keil?	It links object files, resolves addresses, and prepares the final binary to load into MCU.
Can you simulate peripherals in Keil?	Yes, UART, timers, GPIO, interrupts can be simulated without hardware.
How do you program a microcontroller in Keil?	Build the code, generate HEX file, and use Flash Utility with ULINK or ISP tool.
How to debug UART in Keil?	Use Watch window for UART registers and Terminal for simulating serial I/O.

◆ Important Output Files

File	Description
.hex	Intel HEX file used for flashing
.axf	Arm Executable Format (with debug info)
.lst	Assembly listing file
.map	Memory map file
.o	Object files (intermediate output)

◆ Tools Often Used With Keil µVision

Tool	Purpose
Flash Magic	For programming LPC2148 using UART
Proteus / Multisim	For simulating circuits
ULINK / J-Link	For flashing and debugging hardware
Putty / RealTerm / Docklight	For viewing UART communication

Keil uVision is a popular Integrated Development Environment (IDE) used for embedded systems development, particularly for ARM-based microcontrollers. It is widely used in industries for firmware development, making it a common topic in embedded systems interviews.

---

1. Overview of Keil uVision

- **Developer**: Keil (now part of ARM, acquired by ARM in 2005).
- **Purpose**: Used for developing, debugging, and testing embedded applications.
- **Supported Architectures**:

    - ARM Cortex-M, Cortex-R, Cortex-A.
    - Legacy 8051, C166, 251 microcontrollers.
- **License Model**: Proprietary (Free MDK-Lite version available with code size limitations).

---

2. Key Components of Keil uVision

a. µVision IDE

- Provides a user-friendly interface for code editing, compiling, and debugging.
- Features:
    - Project Management.
    - Syntax-highlighting editor.
    - Build automation.
    - Integrated debugger (simulator & hardware debugging).

b. ARM Compiler (armcc, armclang)

- Optimized C/C++ compiler for ARM-based chips.
- Supports:
    - C99, C11, and C++14 standards.
    - ARM/Thumb instruction sets.
    - Inline & embedded assembly.

c. Debugger

- Supports:
    - **Simulator**: Debug without hardware.
    - **Hardware Debugging**: Via JTAG/SWD (using ULINK, J-Link, ST-Link).
    - **Real-Time Trace** (with Cortex-M3/M4/M7 ETM).

d. RTOS Support

- Integrated middleware (RTX RTOS, FreeRTOS).
- Task-aware debugging.

e. Device Support

- Includes **Device Family Packs (DFPs)** for various MCUs (STM32, NXP, TI, etc.).
- CMSIS (Cortex Microcontroller Software Interface Standard) compliant.

---

3. Workflow in Keil uVision

1. **Create a Project**
    - Select target microcontroller (e.g., STM32F103).
    - Configure device settings (clock, peripherals via **Device Configuration Wizard**).
2. **Write Code**
    - C/C++ with ARM-specific extensions (e.g., __attribute__((section(".text")))).
3. **Build & Compile**
    - Compiler generates .axf (ELF) or .hex/.bin files.
    - Checks for syntax errors, memory allocation.

4. **Debugging**
    - Use **Breakpoints, Watch Windows, Memory View**.
    - **Peripheral Registers View** (for debugging GPIO, UART, etc.).
    - **Logic Analyzer** (for real-time signal monitoring).
5. **Flash to Target**
    - Program the MCU via debug probe (ULINK, J-Link).

---

4. Important Features for Embedded Development

- **Memory Optimization** (Code & Data segments).
- **Inline Assembly Support** (for low-level control).
- **CMSIS Integration** (Standardized HAL for ARM Cortex).
- **Trace Capabilities** (SWV – Serial Wire Viewer for RTOS awareness).
- **Powerful Macros & Scripting** (for automation).

---

5. Common Interview Questions on Keil uVision

a. Basic Questions

1. What is Keil uVision used for?
    - Embedded firmware development for ARM/8051 microcontrollers.
2. What file formats are generated after compilation?
    - .axf (ELF debug file), .hex (Intel Hex), .bin (Binary).
3. How do you configure a new microcontroller in Keil?
    - Select the device from the database or install DFPs.

b. Debugging-Related Questions

4. How do you debug a HardFault in Keil?
    - Use **Call Stack + Disassembly**, check **SCB registers**, and analyze **fault handlers**.
5. What is the difference between **Simulator** and **Hardware Debugging**?
    - Simulator runs code virtually; hardware debugging requires a physical MCU & debug probe.

c. Optimization & Memory Questions

6. How do you reduce code size in Keil?
    - Use compiler optimizations (-O1, -O2, -O3), remove unused code, and use __packed for structs.
7. Where is the stack and heap defined in Keil projects?
    - Configured in the **startup file** (startup_*.s) or via linker script (scatter file).

d. Advanced Questions

8. How do you use RTOS with Keil?
    - Integrate RTX or FreeRTOS via **Software Packs**.
9. Explain how to use **SWV (Serial Wire Viewer)**.

- Enable ITM tracing, configure ITM_SendChar(), and view data in **Debug → Serial Wire Viewer**.

10. What is a **scatter file** in Keil?
    - A linker script that defines memory regions (Flash, RAM) and section placements.

---

6. Pros & Cons of Keil uVision

Pros	Cons
Easy to use for ARM development.	Limited free version (32KB code limit in MDK-Lite).
Strong debugging capabilities.	Proprietary (not open-source).
CMSIS & RTOS integration.	Steeper learning curve for advanced features.
Wide MCU support.	Windows-only (no Linux/macOS support).

---

7. Alternatives to Keil uVision

- **STM32CubeIDE** (Free, Eclipse-based, for STM32).
- **IAR Embedded Workbench** (Paid, high optimization).
- **PlatformIO** (Open-source, VS Code-based).
- **GCC ARM Embedded (Arm-none-eabi-gcc)** (Open-source toolchain).

---

8. Tips for Interview Preparation

- **Hands-on Practice**: Build a simple LED blink project in Keil.
- **Debugging Skills**: Learn to use breakpoints, watchpoints, and logic analyzer.
- **Memory Management**: Understand .map file analysis.
- **RTOS Basics**: Try a simple RTOS task in Keil RTX.

#Flash Magic

🔹 What is Flash Magic?

Flash Magic is a **Windows-based flash programming utility** developed by **Embedded Systems Academy (ESA)** in partnership with **NXP Semiconductors**. It is used to **program NXP microcontrollers**, especially **ARM7-based (LPC2148, LPC2129)**, via **UART (serial port)** using **In-System Programming (ISP)** mode.

🔹 Why is Flash Magic Important?

Reason	Explanation
✅ **No external debugger needed**	Can program MCUs using just a serial port (COM/USB to UART).
✅ **Supports ISP mode**	Uses built-in bootloader in LPC series for flashing.
✅ **Reliable & user-friendly**	Easy GUI interface and detailed device configuration.
✅ **Supports HEX file flashing**	Works with .hex files generated by Keil µVision.

🔹 Key Features

Feature	Description
Device Selection	Choose specific MCU model (e.g., LPC2148).
Baud Rate Configuration	Adjust communication speed (commonly 9600 or 38400).
COM Port Setup	Select correct port connected to MCU via USB-to-Serial.
Hex File Input	Load .hex file generated from Keil or other compilers.
Erase Options	Full erase, page erase, or no erase before writing.
Verify Option	Ensures written content matches hex file.
Read/Protect Memory	Allows write/read protection and block protection.

🔷 How Flash Magic Works (Step-by-Step)

⚙️ Preconditions:

- Microcontroller must be connected to PC via **UART/USB**.
- MCU should be in **ISP (In-System Programming)** mode:
  - For LPC2148: Hold **P0.14 = LOW** during RESET (to enter ISP mode).

🧠 Steps to Program Using Flash Magic:

1. **Connect Hardware**
   - LPC2148/2129 board with USB-to-UART (FTDI/PL2303).
   - P0.14 pulled LOW and RESET pressed → MCU enters bootloader mode.
2. **Launch Flash Magic**
3. **Select Settings:**
   - **Device**: LPC2148 / LPC2129
   - **COM Port**: COM3 / COM4 (based on USB-to-Serial device)
   - **Baud Rate**: 9600 or 38400
   - **Interface**: None (Default)
4. **Load HEX File**
   - Browse and select your compiled .hex from Keil output folder.
5. **Erase Option**
   - Choose **Erase all flash** (recommended for clean flash).
6. **Verify After Programming**
   - Check this to ensure integrity of flash.
7. **Click "Start"**
   - Programming starts. It shows progress, status, and success/failure.

🔷 Interview-Level Concepts

Question	Suggested Answer
What is Flash Magic?	It's a PC software tool to flash NXP MCUs like LPC2148 via UART using In-System Programming mode.
What is ISP in LPC2148?	ISP is a bootloader mode where the MCU listens on UART to receive hex files for programming flash memory.
How does MCU enter ISP mode?	Pull P0.14 LOW, press RESET. Bootloader checks P0.14 at reset to decide ISP entry.
What files does Flash Magic use?	It uses .hex files generated by compilers like Keil.
Can Flash Magic verify written data?	Yes, it compares written flash memory with the original hex file.
What happens if ISP is not properly entered?	Flash Magic will show a communication failure – device won't respond on UART.

🔷 Common Errors and Troubleshooting

Error	Cause	Solution
Unable to communicate (failed to autobaud)	ISP not entered	Check P0.14 LOW, Reset timing
No COM port detected	USB driver not installed	Install FTDI/PL2303 driver
Flash programming failed	Bad hex file or communication error	Rebuild project in Keil, re-check UART wiring
Verify failed	Data mismatch	Re-flash with erase all, check memory protection

---

🔷 Practical Application in Gopal's Projects

Project	Use of Flash Magic
Time-Driven Access System (LPC2148)	Use Flash Magic to load new firmware each time you test RTC or password system.
CAN-Based Vehicle System (LPC2129)	Easily flash each node's code using a single USB-to-UART setup.
ATM Simulation (LPC2148)	Whenever banking logic updates, use Flash Magic to load .hex from Keil.

🔷 Useful Tips

- Use **Flash Magic command-line** for automated flashing in factories or large batches.
- Flash Magic also supports **read-back** and **protecting flash memory**.
- Always double-check **jumper settings** and **UART line connections** before flashing.

🔷 Flash Magic Workflow Diagram

Keil µVision → Build Project → Generates .hex File
↓
Flash Magic → Select Device, COM, Baud, HEX File
↓
Microcontroller (in ISP Mode) ← via UART
↓
Flash Memory Gets Programmed → Ready to Run

🔷 What is HyperTerminal?

HyperTerminal is a **Windows-based serial communication program** used to **interact with microcontrollers or embedded systems via UART**. It allows you to:

- Send/receive data through serial ports (COM).
- Communicate with microcontroller bootloaders (like in ISP).
- Manually program or debug via textual commands.

🔷 Flash Magic vs HyperTerminal – What's the Difference?

Feature	Flash Magic	HyperTerminal
GUI Tool	✅ Yes	❌ No (Text terminal)
Auto Flashing	✅ Yes (with HEX)	❌ No
Manual ISP Commands	❌ Not visible	✅ Yes (visible command entry)
Debugging via UART	❌ Not ideal	✅ Best for UART-level debugging

---

🔷 Why Use HyperTerminal with Flash Magic?

Flash Magic **automates the ISP programming**, but **HyperTerminal** helps in:

- **Manually accessing the bootloader** (to test if the microcontroller entered ISP mode).
- **Sending ISP commands** directly (e.g., U, J, R, W).
- **Debugging UART communication** (baud rate mismatch, character loss).
- **Learning how ISP protocol works internally**.

🔷 Entering ISP Mode in LPC2148/LPC2129

Step	Action
1	Pull **P0.14 LOW**
2	Press and release **RESET**
3	MCU enters bootloader mode and listens via **UART0**

🔷 Using HyperTerminal to Test ISP Mode

✅ Setup

1. **Connect USB-to-Serial Adapter**
   - TX → RXD0 (P0.0)
   - RX → TXD0 (P0.1)
   - GND → GND
2. **Open HyperTerminal**
   - Name: Any
   - COM Port: COMx (match your USB-to-Serial port)
   - Baud rate: **9600**
   - Data Bits: 8
   - Parity: None
   - Stop Bits: 1
   - Flow Control: None
3. **Press 'Enter' key**
   - You should receive **"Synchronized"** from the microcontroller if it's in ISP mode.

🔷 LPC2148 Bootloader Protocol via HyperTerminal

Once in ISP mode, LPC2148 expects a sequence of commands:

Command Sent	MCU Response	Meaning
?	Synchronized	Shows bootloader is active
Synchronized	OK	Confirms sync
12000	OK	Clock frequency in kHz
J	OK	Jump to user code (starts application)
R	OK	Read memory
W	OK	Write memory
E	OK	Erase sectors
P	OK	Prepare sectors for write

🔷 Flashing via HyperTerminal (Manual Way)

Note: Flash Magic automates this internally, but HyperTerminal is useful for understanding what's happening.

1. **Send '?'**
   → MCU sends: Synchronized
2. **Send 'Synchronized'**
   → MCU responds: OK
3. **Send '12000'**
   (Clock freq in kHz)
   → MCU responds: OK
4. **Now send commands like:**
   - E 0 15 → Erase sectors 0 to 15
   - P 0 15 → Prepare sectors
   - W address size <data> → Write data to flash

You must calculate checksum and use Base64 encoding manually if doing full programming – **Flash Magic handles all this**.

🔷 Interview-Level Questions & Answers

Question	Answer
What is HyperTerminal used for in embedded systems?	To send/receive data over serial communication (UART), especially for debugging and manual ISP mode communication.
How do you confirm if LPC2148 entered ISP mode?	Use HyperTerminal and send '?' → expect 'Synchronized' as a response.
What is the benefit of using HyperTerminal with Flash Magic?	To verify manual communication with bootloader and debug serial issues if Flash Magic fails.

Can you flash firmware using HyperTerminal?	Technically yes, by sending bootloader commands and encoded data manually, but it's complex.
Why is P0.14 used in LPC2148 ISP mode?	It determines whether the MCU will enter ISP (when LOW at reset).
Difference between Flash Magic and HyperTerminal?	Flash Magic is automated and user-friendly for flashing; HyperTerminal allows low-level command interaction for learning/debugging.

🔷 Real-Time Use Case

You're flashing your LPC2148 project and Flash Magic shows:
Unable to communicate. (Failed to autobaud)
Solution:

- Open HyperTerminal.
- Send ? and check for Synchronized.
- If no response: check P0.14, reset, COM port, baud rate.
- Once Synchronized appears, retry with Flash Magic.

🔷 Final Advice

- 🔹 Use Flash Magic for flashing .hex files.
- 🔹 Use HyperTerminal **only when debugging** Flash Magic issues or learning the protocol.
- 🔹 Understanding HyperTerminal usage shows you have **low-level control** and **in-depth microcontroller knowledge**—this **impresses interviewers**.

🔷 Is HyperTerminal Part of Flash Magic?

Not exactly — but here's the **correct technical breakdown**:

✅ Flash Magic (Tool)

- **Flash Magic** is a **standalone Windows application** developed by **Embedded Systems Academy**.
- It provides a **Graphical User Interface (GUI)** for **ISP (In-System Programming)** of NXP microcontrollers like **LPC2148, LPC2129** using UART0.
- It automates the **entire ISP communication process**, including:
  - Entering bootloader mode.
  - Sending required boot commands.
  - Erasing memory.
  - Writing .hex or .bin files.
  - Verifying flash.
  - Setting boot vectors or security features.

🔶 **Where Does HyperTerminal Come In?**

- **Flash Magic includes a menu option** to **launch a terminal interface**, which allows **basic serial communication**, but **this terminal is not the classic Windows HyperTerminal**.
- If the **original Windows HyperTerminal** is installed, you can also launch it **separately** and use it **in parallel** for **debugging or monitoring** the serial communication that Flash Magic uses.

🔹 In some versions of Flash Magic, there's a **built-in terminal tool** that mimics HyperTerminal behavior.

🔹 It can be found under:

Flash Magic → Tools → Terminal (or HyperTerminal)

🔷 **So What's the Correct Understanding?**

Feature	Flash Magic	HyperTerminal
Developed By	Embedded Systems Academy	Microsoft (or Hilgraeve Inc.)
Flashing Firmware	✅ Yes	❌ No (manual only)
Graphical UI	✅ Yes	✅ Yes (basic)
ISP Bootloader Automation	✅ Yes	❌ No
Manual Serial Commands	❌ No (hidden)	✅ Yes
Terminal Support	✅ Built-in terminal in Tools menu	✅ External tool

---

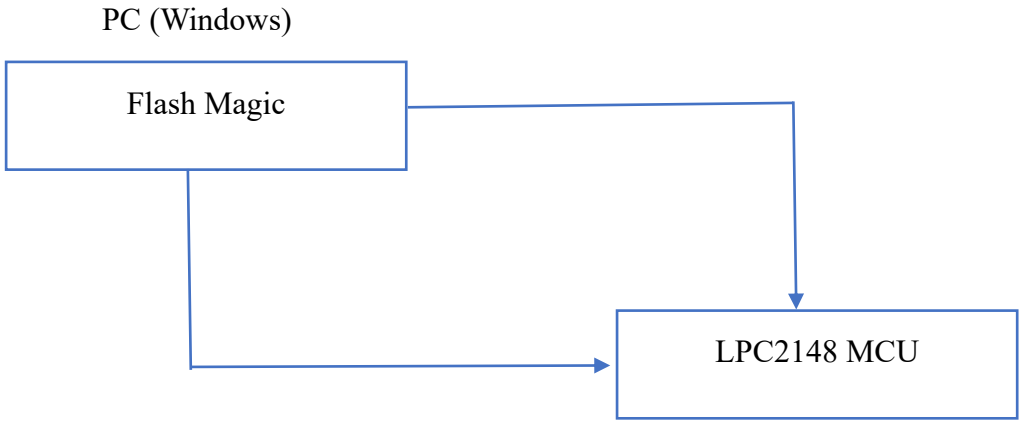
🔷 **Interview-Focused Answer**

❓ **Q: Is HyperTerminal part of Flash Magic?**

Answer:

No, HyperTerminal is **not inherently part of Flash Magic**, but Flash Magic **offers a built-in terminal utility** similar to HyperTerminal. This built-in terminal allows UART-based serial communication with the microcontroller for **manual ISP testing** or **bootloader debugging**. Alternatively, developers often use the **external Windows HyperTerminal** tool in parallel to Flash Magic for monitoring serial data or troubleshooting bootloader communication issues.

◆ Visual Flow (Conceptual)



(Also optionally use HyperTerminal to see UART TX/RX)

◆ Real-World Developer Practice

- Use **Flash Magic** to burn .hex file to LPC2148.
- If Flash Magic fails (e.g., no response, "Unable to Autobaud"):
  - Open **HyperTerminal** or Flash Magic's **built-in terminal**.
  - Send ? manually to check for Synchronized response.
  - This confirms whether MCU entered **ISP mode**.
- This combined approach ensures **robust debugging** during embedded development.

BANK DATABASED

bank.h

```
#ifndef _BANK_H_       // Header guard to prevent multiple inclusions
#define _BANK_H_       // Define the header guard

// Transaction limits (in rupees)
#define MAX_DEPOSIT      30000    // Maximum deposit amount per transaction
#define MAX_WITHDRAW     30000    // Maximum withdrawal amount per transaction
#define MAX_TRANSFER     100000   // Maximum transfer amount per transaction

// Length limits for strings
#define NAME_LEN 30          // Maximum length for names
#define MAX_PASS_LEN 20      // Maximum password length
#define MAX_USRN_LEN 20      // Maximum username length

// Admin credentials
#define ADMIN_USRN "admin"   // Admin username
#define ADMIN_PASS "admin"   // Admin password
#define ADMIN_EXIT "exit"    // Exit command

// Transaction type constants
#define WITHDRAW 1          // Withdrawal transaction
#define DEPOSIT  2          // Deposit transaction
#define TRANSFER_IN 3       // Incoming transfer
#define TRANSFER_OUT 4      // Outgoing transfer

// Macro to convert character to uppercase
#define CAPS(ch) (ch &=~(32))  // Clears bit 5 to make uppercase

// Type definitions for cleaner code
typedef unsigned long long int u64;  // 64-bit unsigned integer
typedef double f64;                 // 64-bit floating point

// Transaction structure definition
typedef struct A{
    f64 amt;        // Transaction amount (double)
    u64 id;         // Unique transaction ID (unsigned long long)
    char type;      // Transaction type (WITHDRAW/DEPOSIT/etc)
    struct A *nxt;  // Pointer to next transaction in linked list
}Tran;

// Account structure definition
typedef struct B{
    u64 num;        // Unique account number
    f64 bal;        // Account balance
    u64 phno;       // Phone number
    char usrName[MAX_USRN_LEN];  // Username string
    char pass[MAX_PASS_LEN];     // Password string
    char rfid[9];   // RFID card number (8 chars + null terminator)
    char pin[5];    // 4-digit PIN + null terminator
    int cardStat;   // Card status:
                    // 1-active, 0-blocked, 2-blocked during PIN change
    char *name;     // Pointer to dynamically allocated name string

    Tran *tranHist; // Pointer to transaction history linked list
    u64 tranCnt;    // Count of transactions
    struct B *nxt;  // Pointer to next account in linked list
```

```
}Acc;

// External declaration of database size constant
extern const int szDb;

// Function declarations grouped by category:

// Account-handling main functions
void loginMenu(void);       // Display login menu
void adminMenu(void);       // Display admin menu
void userMenu(void);        // Display user menu
void accMenu(void);         // Display account menu
void menu(void);            // Display main menu

Acc* isValid(Acc*,char*,char*);  // Validate credentials
int isUnq(Acc*,char*);           // Check if username is unique
int isNewRFID(Acc *head,char *rf); // Check if RFID is unique
void dispAcc(Acc*);              // Display account details

void newAcc(Acc**);         // Create new account
void updateAcc(Acc**,Acc*); // Update account info
void dltAcc(Acc**);         // Delete account (unimplemented)
Acc* getAcc(Acc*);          // Get account by search criteria

// Transaction functions
void withdraw(Acc*);        // Process withdrawal
void deposit(Acc*);         // Process deposit
void transfer(Acc*,Acc*);   // Process transfer
void addTran(Acc*,f64,char);// Add transaction to history
void balance(Acc*);         // Show balance
void statement(Acc*);       // Show statement
void database(Acc*);        // Display all accounts

// Helper functions
u64 getUnqId(Acc*);         // Generate unique account ID
u64 getTranId(Acc*);        // Generate unique transaction ID
u64 getTimeStamp(void);     // Get current timestamp
char getKey(void);          // Get single keypress
char* getStr(void);         // Get string input
void format(char*);         // Format name string

// File-handling functions
void saveData(Acc *);       // Save all data to files
void syncData(Acc **);      // Load data from files

#endif  // End of header guard
```

Key points about this header file:

1. **Organization**: Clearly separates constants, type definitions, and function declarations
2. **Data Structures**: Defines the core Acc (Account) and Tran (Transaction) structures
3. **Modularity**: Groups related functions together with comments
4. **Type Safety**: Uses typedefs (u64, f64) for consistent data sizes
5. **Security**: Defines size limits for sensitive fields (passwords, PINs)
6. **Persistence**: Declares file handling functions for data storage
7. **Completeness**: Covers all major banking operations (accounts, transactions, admin)

bank.c

Header Includes and Global Definitions

```
#include <stdio.h>        // Standard input/output functions (printf, scanf)
#include <stdio_ext.h>    // Extended stdio functions (__fpurge for buffer clearing)
#include <stdlib.h>       // Memory allocation (malloc, free), exit, etc.
#include <stdint.h>       // Standard integer types (uint64_t, etc.)
#include <string.h>       // String manipulation functions (strcmp, strcpy)
#include <unistd.h>       // POSIX API (getpid for process ID)
#include <time.h>         // Time functions (time, localtime)
#include "bank.h"         // Custom header with bank-specific definitions
#include <termios.h>      // Terminal I/O control (for getch implementation)
#include <fcntl.h>        // File control options (for non-blocking input)

const int szDb = (sizeof(u64)*2 + sizeof(f64) + sizeof(char)*(MAX_USRN_LEN + MAX_PASS_LEN));
// Calculates size needed for database fields:
// 2 unsigned long longs (account num + phone)
// + 1 double (balance)
// + username and password strings
```

Menu Display Functions

```
void loginMenu(void) {
    printf("\n-------------WELCOME TO VECTOR BANK--------------\n");
    printf("Please enter login credentials.\n");
}
// Displays the initial login screen with bank name and credential prompt

void adminMenu(void) {
    printf("\nHI ADMIN:\n"
           "[KEY]  : ACTION\n"
           "c/C    : Create New account.\n"
           "u/U    : Update Existing account.\n"
           "h/H    : Transaction history.\n"
        "w/W    : Withdraw amount.\n"
        "d/D    : Deposit amount.\n"
        "b/B    : Balance enquery.\n"
        "t/T    : Transfer money.\n"
        "x/X    : Activate card.\n"
        "e/E    : Display all accounts details.\n"
        "f/F    : Finding/searching for specific account.\n"
           "q/Q    : Quit from app.\n"
           "Enter choice:");
}
// Shows all administrative options in formatted columns
// Each letter corresponds to a different banking operation

void userMenu(void) {
    printf("\nHI CUSTOMER:\n"
           "[KEY]  : ACTION\n"
        "c/C    : Create New account.\n"
        "u/U    : Update Old account info.\n"
        "h/H    : Transaction history.\n"
        "w/W    : Withdraw amount.\n"
```

```
        "d/D    : Deposit amount.\n"
        "b/B    : Balance enquery.\n"
        "t/T    : Transfer money.\n"
        "e/E    : Display all accounts details.\n"
        "f/F    : Finding/searching for specific account.\n"
           "q/Q    : Quit from app.\n"
           "Enter choice:");
}
// Displays reduced set of options for regular customers
```

Account Validation

```
Acc* isValid(Acc *head, char *usr, char *pass) {
    while(head) {  // Traverse linked list of accounts
        if((!strcmp(usr,head->usrName)) && (!strcmp(pass,head->pass)))
            break;  // Found matching credentials
        head=head->nxt;  // Move to next account
    }
    return head;  // Returns account pointer if found, NULL otherwise
}
// Compares input username/password against stored values
```

Account Creation

```
void newAcc(Acc **head) {
    char *temp=NULL, flag=1;
    double d;
    Acc *new=calloc(1,sizeof(Acc));  // Allocate zero-initialized account struct
    new->tranHist=NULL;  // Initialize empty transaction history
    new->tranCnt=0;      // Set transaction count to zero
    new->num=getUnqId(*head);  // Generate unique account number

    // Name Collection and Validation
    printf("Enter Name:");
    __fpurge(stdin);  // Clear input buffer
    new->name=getStr();  // Dynamically allocate name string
    if(strlen(new->name)<3) {  // Minimum length check
        puts("no name");
        free(new->name);
        free(new);
        return;
    }
    format(new->name);  // Capitalize proper name formatting

    // Phone Number Validation Loop
    flag=1;
    while(flag) {
        if(flag!=1) puts("Mobile must be 10 digit, and min:60000 0000");
        printf("Enter Mobile No.:");
        scanf("%llu",&(new->phno));
        if((new->phno>=6000000000)&&(new->phno<9999999999)) break;
        else flag++;
    }

    // Username Validation Loop
    flag=1;
    while(flag) {
        printf("Enter Login Username:");
        temp=getStr();
```

```
        if(strlen(temp)>=MAX_USRN_LEN) {
            puts("Username too long,try again!!");
            free(temp);
            continue;
        }
        if(isUnq(*head,temp)) {  // Check uniqueness
            strcpy(new->usrName,temp);
            free(temp);
            flag=0;
        } else {
            puts("User name aldready exits!");
            free(temp);
        }
    }

    // Password Entry with Confirmation
    flag=1;
    while(flag) {
        if(flag>1) puts("Password mismatch!! Retry.");
        printf("Enter Login Password:");
        temp=getStr();
        if(strlen(temp)>=MAX_PASS_LEN) {
            puts("Password too long,try again!!");
            free(temp);
            continue;
        }
        strcpy(new->pass,temp);
        free(temp);
        printf("Re-enter Login Password:");
        temp=getStr();
        flag=(strcmp(temp,new->pass)?flag+1:0);  // 0 if match, increment if not
    }
    free(temp);

    // RFID Card Number Entry
    flag=1;
    while(flag) {
        if(flag>1) printf("8 digit RFID please:");
        temp=getStr();
        if(strlen(temp)!=8) {
            free(temp);
            flag++;
            continue;
        }
        if(isNewRFID(*head,temp)) {  // Check RFID uniqueness
            strcpy(new->rfid,temp);
            free(temp);
            break;
        }
        free(temp);
        flag++;
    }

    // PIN Entry with Confirmation
    flag=1;
    while(flag) {
        if(flag>1) puts("Pin mismatch!! Retry.");
        printf("Enter ATM Pin:");
        temp=getStr();
```

```
        if(strlen(temp)!=4) {
            free(temp);
            continue;
        }
        strcpy(new->pin,temp);
        free(temp);
        printf("Re-enter ATM Pin:");
        temp=getStr();
        flag=(strcmp(temp,new->pin)?flag+1:0);
    }
    free(temp);

    new->cardStat=1;  // Set card status to active (1)

    // Initial Balance Entry
    flag=1;
    while(flag) {
        if(flag>1) printf("Enter an posivite amount:");
        scanf("%lf",&d);
        if(d>0) {
            new->bal=d;
            break;
        }
        flag++;
    }

    addTran(new,new->bal,DEPOSIT);  // Record opening deposit
    dispAcc(new);                   // Show account summary

    // Add to account list (at head for O(1) insertion)
    new->nxt=*head;
    *head=new;
    puts("Account Created.");
}
```

Helper Functions

```
int isUnq(Acc *head, char *str) {
    if(!strcmp(str,ADMIN_USRN)) return 0;  // Can't use admin username
    while(head) {
        if(!strcmp(str,head->usrName)) return 0;  // Username exists
        head=head->nxt;
    }
    return 1;  // Username is unique
}
int isNewRFID(Acc *head,char *rf)
{
        while(head)
        {
                if(!strcmp(rf,head->rfid))
                        return 0;
                head=head->nxt;
        }
        return 1;
}

u64 getUnqId(Acc *head) {
    u64 num;
    Acc *temp=NULL;
```

```
    int found=1;
    while(found) {
        srand(getpid() + (unsigned int)head);  // Seed with process ID + memory address
        num=getTimeStamp()*10000 + (rand()%10000);  // Timestamp + random component
        temp=head;
        found=0;
        while(temp) {  // Verify uniqueness
            if(num==temp->num) { found=1; break; }
            temp=temp->nxt;
        }
    }
    return num;  // 18-digit unique ID
}

u64 getTimeStamp(void) {
    time_t rawtime;
    struct tm *timeinfo;
    time(&rawtime);  // Get current time
    timeinfo = localtime(&rawtime);
    // Format as YYYYMMDDHHMMSS (14 digits)
    return (timeinfo->tm_year+1900)*10000000000ULL +
           (timeinfo->tm_mon+1)*100000000ULL +
           timeinfo->tm_mday*1000000ULL +
           timeinfo->tm_hour*10000ULL +
           timeinfo->tm_min*100ULL +
           timeinfo->tm_sec;
}

char *getStr(void) {
    char buff[NAME_LEN]={0};  // Initialize buffer
    fflush(stdout);  // Flush output stream
    __fpurge(stdin);  // Clear input buffer
    scanf("%[^\n]",buff);  // Read until newline
    return strdup(buff);  // Return heap-allocated copy
}

char getch(void) {
    struct termios oldt, newt;
    int oldf;
    int ch;
    tcgetattr(STDIN_FILENO, &oldt);  // Save terminal settings
    newt = oldt;
    newt.c_lflag &= ~(ICANON);  // Disable canonical mode
    tcsetattr(STDIN_FILENO, TCSANOW, &newt);  // Apply new settings
    oldf = fcntl(STDIN_FILENO, F_GETFL, 0);
    fcntl(STDIN_FILENO, F_SETFL, oldf | O_NONBLOCK);  // Non-blocking mode
    ch = getchar();  // Get single character
    // Restore original settings
    tcsetattr(STDIN_FILENO, TCSANOW, &oldt);
    fcntl(STDIN_FILENO, F_SETFL, oldf);
    return (ch != EOF) ? ch : -1;  // Return char or -1 if none
}
char getKey(void)
{
        char key[3],ch;
        int i=0;
        fflush(stdout);
        __fpurge(stdin);
        key[0]=getchar();
```

```
        CAPS(key[0]);
        puts("");
        return key[0];
}
void format(char *str)
{
        CAPS(str[0]);
        str++;
        while(*str)
        {
                if(*(str-1)==' ')CAPS(*str);
                str++;
        }
}
void updateAcc(Acc **head,Acc *usr)
{
        char key,*temp=NULL;
        int flag=1;
        accMenu();

        key=getKey();
        __fpurge(stdin);
        puts("");
        switch(key)
        {
                case 'P':printf("Enter New phone number:");
                         scanf("%llu",&(usr->phno));
                         puts("Mobile Updated.");
                         break;
                case 'O':printf("Enter New Holder name:");
                         __fpurge(stdin);
                         temp=getStr();
                         if(strlen(temp)<3)
                         {
                                 puts("invalid name");
                                 free(temp);
                                 return;
                         }
                         format(temp);
                         usr->name=temp;
                         puts("Name Updated.");
                         break;
                case 'U':flag=1;
                         while(flag)
                         {
                                 printf("Enter New Username:");
                                 temp=getStr();
                                 if(strlen(temp)>MAX_USRN_LEN)
                                 {
                                         puts("Username too long,try again!!");
                                         continue;
                                 }
                                 if(!strcmp(temp,usr->usrName))
                                 {
                                         puts("New Username is same as your Existing Username.");
                                         puts("Do you wanna change the username?(y/n):");
                                         key=getKey();
                                         if(key=='Y')
                                         {
```

```
                                        free(temp);
                                        continue;
                                }
                                else if(key=='N')
                                {
                                        free(temp);
                                        return;
                                }

                        }
                        else if(isUnq(*head,temp))
                        {
                                strcpy(usr->usrName,temp);
                                free(temp);
                                flag=0;
                        }
                        else
                        {
                                puts("User name aldready exits!");
                                puts("Try differnt one.");
                        }
                }
                if(!flag)
                        puts("Username Updated.");
                break;
        case 'Q':printf("Enter old password:");
                temp=getStr();
                if(strcmp(temp,usr->pass))
                {
                        puts("Wrong password");
                        break;
                }
                flag=1;
                while(flag)
                {
                        if(flag>1)
                                puts("Password mismatch Retry.");
                        printf("Enter New Login Password:");
                        temp=getStr();
                        if(strlen(temp)>=MAX_PASS_LEN)
                        {
                                puts("Password too long,try again!!");
                                flag=1;
                                continue;
                        }
                        strcpy(usr->pass,temp);
                        free(temp);
                        printf("Re-enter New Login Password:");
                        temp=getStr();
                        flag=(strcmp(temp,usr->pass)?flag+1:0);
                }
                puts("Password Updated.");
                break;
        case 'A':printf("Enter Old ATM Pin:");
                temp=getStr();
                if(strcmp(temp,usr->pin))
                {
                        puts("Wrong pin!!");
                        free(temp);
```

```
                                break;
                        }
                        flag=1;
                        free(temp);
                        while(flag)
                        {
                                if(flag>1)puts("Pin mismatch!! Retry.");
                                printf("Enter New ATM Pin:");
                                temp=getStr();
                                if(strlen(temp)!=4)
                                {
                                        puts("4 digit pin pls,try again!!");
                                        flag=1;
                                        free(temp);
                                        continue;
                                }
                                strcpy(usr->pin,temp);
                                free(temp);
                                printf("Re-enter New ATM Pin:");
                                temp=getStr();
                                flag=(strcmp(temp,usr->pin)?flag+1:0);
                                free(temp);
                        }
                        puts("Pin Updated.");
                        break;
                default:puts("invalid input.");
                        break;
        }
}

void accMenu(void)
{
        printf("\nChange:\n"
                        "[KEY]-ACTION\n"
                        "p/P  -Phone number.\n"
                        "o/O  -Holder's name.\n"
                        "u/U  -Username.\n"
                        "q/Q  -Password.\n"
                        "Enter choice:");
}

void dispAcc(Acc *usr)
{
        puts("\n==:Account Details:==");
        printf("AccNo.:%llu\n",usr->num);
        printf("Name  :%s\n",usr->name);
        printf("Ph.No.:%llu\n",usr->phno);
        printf("Balanc:%lf\n",usr->bal);
        printf("Usrnam:%s\n",usr->usrName);
        printf("Passwd:%s\n",usr->pass);
        printf("RFID  :%s\n",usr->rfid);
        printf("Pin   :%s\n",usr->pin);
        printf("Card  :%s\n",(usr->cardStat==1)?"ACTIVE":"BLOCKED");
        printf("TranNo:%llu\n",usr->tranCnt);
}

Acc* getAcc(Acc *head)
{
        u64 num=0;
```

```
        char key='\0',*str=NULL;
        printf( "Find by:\n"
                        "[KEY]-ACTION\n"
                        "p/P  -Phone number.\n"
                        "n/N  -Account number.\n"
                        "o/O  -Holder name.\n"
                        "u/U  -Username.\n"
                        "Enter choice:");
        key=getKey();
        __fpurge(stdin);
        puts("");
        switch(key)
        {
                case 'P':printf("Enter phone number:");
                         scanf("%llu",&num);
                         while(head)
                         {
                                 if(num==head->phno)break;
                                 head=head->nxt;
                         }
                         break;
                case 'N':printf("Enter account number:");
                         scanf("%llu",&num);
                         while(head)
                         {
                                 if(num==head->num)break;
                                 head=head->nxt;
                         }
                         break;
                case 'O':printf("Enter holder name:");
                         str=getStr();
                         format(str);
                         while(head)
                         {
                                 if(!strcmp(str,head->name))break;
                                 head=head->nxt;
                         }
                         free(str);
                         break;
                case 'U':printf("Enter username:");
                         str=getStr();
                         while(head)
                         {
                                 if(!strcmp(str,head->usrName))break;
                                 head=head->nxt;
                         }
                         free(str);
                         break;
                default:puts("invalid choice");
                        head=NULL;
                        break;
        }
        return head;
}

void dltAcc(Acc **head)
{
        if(!(*head))return;
}
```

```
void balance(Acc *usr)
{
        printf("\nAccount Number : %llu\n",usr->num);
        printf("Holder Name    : %s\n",usr->name);
        printf("Current Balance: %+lf Rs/-\n",usr->bal);
}
```

Transaction Processing

```
void deposit(Acc *usr) {
    f64 amt=0;
    printf("\nEnter Deposit Amount:");
    scanf("%lf",&amt);
    if(amt<=0) {
        puts("Amount cannot be negative!!");
    } else if(amt<MAX_DEPOSIT) {
        usr->bal += amt;  // Update balance
        addTran(usr,+amt,DEPOSIT);  // Record transaction
        puts("Amount Deposited.");
    } else {
        puts("Amount exceeds Max.Deposit limit!!");
    }
}

void withdraw(Acc *usr) {
    f64 amt=0;
    printf("\nEnter Withdrawal Amount:");
    scanf("%lf",&amt);
    if(amt<=0) {
        puts("Amount cannot be negative!!");
    } else if(amt<MAX_WITHDRAW) {
        if(amt<=usr->bal) {  // Check sufficient funds
            usr->bal -= amt;
            addTran(usr,-amt,WITHDRAW);
            puts("Amount Withdrawn.");
        } else {
            puts("Low Balance!!");
        }
    } else {
        puts("Amount exceeds Max.Withdraw limit!!");
    }
}

void transfer(Acc *from, Acc *to) {
    f64 amt=0;
    printf("\nEnter Transfer Amount:");
    scanf("%lf",&amt);
    if(amt<=0) {
        puts("Amount cannot be negative!!");
    } else if(amt<MAX_TRANSFER) {
        if(amt<=from->bal) {  // Check sender's balance
            from->bal -= amt;
            to->bal += amt;
            addTran(to,+amt,TRANSFER_IN);  // Record for recipient
            addTran(from,-amt,TRANSFER_OUT);  // Record for sender
            puts("Amount Transfered.");
        } else {
            puts("Low Balance!!");
```

```
        }
    } else {
        puts("Amount exceeds Max.Transfer limit!!");
    }
}

void addTran(Acc *usr, f64 amt, char type) {
    Tran *new=calloc(1,sizeof(Tran));  // Allocate new transaction
    new->amt=amt;  // Set amount
    new->id=getTranId(usr);  // Generate unique ID
    new->type=type;  // Set transaction type
    new->nxt=usr->tranHist;  // Add to front of list
    usr->tranHist=new;  // Update head pointer
    (usr->tranCnt)++;  // Increment count
}
u64 getTranId(Acc *usr)
{
        //17 digit unq TranID
        srand(usr->num);
        return getTimeStamp()*1000 +(rand()%1000);
}

void statement(Acc *usr)
{
        Tran *temp = usr->tranHist;
        if(temp)
        {
                printf("\n%-20s%-23s%-12s\n", "Transaction ID", "Amount (Rs)","Type");
                puts("----------------------------------------");
                while (temp) {
                        printf("%-20llu%-+20.2lf",temp->id,temp->amt);
                        if(temp->type==DEPOSIT)   printf("%-12s\n","Deposit");
                        else if(temp->type==WITHDRAW)printf("%-12s\n","Withdraw");
                        else if(temp->type==TRANSFER_IN)printf("%-12s\n","Tranfer IN");
                        else if(temp->type==TRANSFER_OUT)printf("%-12s\n","Tranfer OUT");
                        temp = temp->nxt;
                }
        }
        else
        {
                puts("No Transaction History!");
        }
}

void database(Acc *head)
{
        if(!head)
        {
                puts("Empty Database!!WTF");
                return;
        }
        printf("\n%-20s|%-40s|%-14s|%-12s\n",
                        "Account ID",
                        "Holder Name",
                        "Mobile(+91)",
                        "Transactions");
        while(head)
        {
                printf("%-20llu|%-40s|+91-%-10llu|%-12llu\n",
```

```
                                head->num,
                                head->name,
                                head->phno,
                                head->tranCnt);
                head=head->nxt;
        }
}
```

Database Operations

```
void saveData(Acc *head) {
    // Create data directory if missing
    struct stat st = {0};
    if(stat("../data", &st) == -1 && mkdir("../data", 0700) == -1) {
        perror("Failed to create data directory");
        return;
    }

    // Open files for writing
    FILE *userFile = fopen("../data/users.txt", "w");
    FILE *transFile = fopen("../data/transactions.txt", "w");

    // Write CSV headers
    fprintf(userFile, "RFID,PIN,Name,Balance,AccountNumber,Phone,Username,CardStatus,TransactionCount\n");
    fprintf(transFile, "RFID,Type,Amount,DateTime,TransactionID\n");

    // Write all account data
    Acc *current = head;
    while (current) {
        // Handle NULL fields
        if (current->name == NULL) current->name = strdup("");
        if (current->usrName[0] == '\0') strncpy(current->usrName, "default", MAX_USRN_LEN-1);
        if (current->rfid[0] == '\0') strncpy(current->rfid, "00000000", 8);
        if (current->pin[0] == '\0') strncpy(current->pin, "0000", 4);

        // Write user record
        fprintf(userFile, "%s,%s,%s,%.2f,%llu,%llu,%s,%d,%llu\n",
                current->rfid, current->pin, current->name, current->bal,
                current->num, current->phno, current->usrName,
                current->cardStat, current->tranCnt);

        // Write all transactions
        Tran *t = current->tranHist;
        while (t) {
            time_t rawtime = time(NULL);
            struct tm *timeinfo = localtime(&rawtime);
            fprintf(transFile, "%s,%d,%.2f,%04d-%02d-%02d %02d:%02d:%02d,%llu\n",
                    current->rfid, t->type, t->amt,
                    timeinfo->tm_year + 1900, timeinfo->tm_mon + 1,
                    timeinfo->tm_mday, timeinfo->tm_hour,
                    timeinfo->tm_min, timeinfo->tm_sec,
                    t->id);
            t = t->nxt;
        }
        current = current->nxt;
    }
    fclose(userFile);
    fclose(transFile);
}
```

```
void syncData(Acc **head) {
    *head = NULL;  // Initialize empty list
    Acc *tail = NULL;
    FILE *userFile = fopen("../data/users.txt", "r");
    if (!userFile) return;

    char line[512];
    fgets(line, sizeof(line), userFile);  // Skip header

    while (fgets(line, sizeof(line), userFile)) {
        Acc *newUser = calloc(1, sizeof(Acc));
        newUser->name = calloc(NAME_LEN, sizeof(char));

        // Parse CSV line
        if (sscanf(line, "%8[^,],%4[^,],%[^,],%lf,%llu,%llu,%19[^,],%d,%llu",
                newUser->rfid, newUser->pin, newUser->name, &newUser->bal,
                &newUser->num, &newUser->phno, newUser->usrName,
                &newUser->cardStat, &newUser->tranCnt) != 9) {
            free(newUser->name);
            free(newUser);
            continue;
        }

        // Add to linked list
        if (!*head) *head = newUser;
        else tail->nxt = newUser;
        tail = newUser;
    }
    fclose(userFile);
// Load transactions if users were loaded successfully
        if (*head) {
                FILE *transFile = fopen("../data/transactions.txt", "r");
                if (!transFile) {
                        return;
                }

                // Skip header
                if (!fgets(line, sizeof(line), transFile)) {
                        fclose(transFile);
                        return;
                }

                // Read transaction records
                while (fgets(line, sizeof(line), transFile)) {
                        char rfid[9];
                        int type;
                        double amount;
                        char datetime[20];
                        u64 transId;

                        if (sscanf(line, "%8[^,],%d,%lf,%19[^,],%llu",
                                                rfid, &type, &amount, datetime, &transId) != 5) {
                                continue;
                        }

                        // Find the user for this transaction
                        Acc *user = *head;
                        while (user && strcmp(user->rfid, rfid)) {
```

```
                                user = user->nxt;
                        }

                        if (!user) continue;

                        // Create new transaction
                        Tran *newTrans = calloc(1, sizeof(Tran));
                        if (!newTrans) continue;

                        newTrans->type = type;
                        newTrans->amt = amount;
                        newTrans->id = transId;
                        newTrans->nxt = user->tranHist;
                        user->tranHist = newTrans;
                }
                fclose(transFile);
        }
}
```

---

bankMain.c

Header Includes

```
#include<stdio.h>         // Standard I/O functions (printf, scanf)
#include<stdio_ext.h>     // Extended stdio (__fpurge for buffer clearing)
#include<stdlib.h>        // Memory allocation, exit, etc.
#include<string.h>        // String operations (strcmp)
#include<sys/stat.h>      // Directory creation (mkdir)
#include<sys/types.h>     // System types
#include"bank.h"          // Custom bank definitions and functions
```

Main Function

```
int main() {
    // Variable Declarations
    Acc *db=NULL, *from=NULL, *to=NULL;  // Account pointers: database, current user, transfer recipient
    char key='\0', *pass=NULL, *usr=NULL; // User input variables
    int bye=0;                            // Exit flag

    // Data Directory Setup
    mkdir("../data", 0777);  // Create data directory with full permissions
    syncData(&db);           // Load existing data from files
    puts("Hello All!!");     // Welcome message
    puts("");                // Empty line

    // Main Program Loop
    while(1) {
        // Login Screen
        loginMenu();                    // Display login prompt
        printf("Enter Username:");      // Prompt for username
        usr=getStr();                   // Get username string
        printf("Enter Password:");      // Prompt for password
        pass=getStr();                  // Get password string

        // Authentication Check
        if(!strcmp(usr,ADMIN_USRN)) {   // Check for admin username
            if(!strcmp(pass,ADMIN_PASS))
```

```
        key='A';                // Set admin mode
    else if(!strcmp(pass,ADMIN_EXIT))
        exit(1);                // Exit if special admin password
}
else if(from=isValid(db,usr,pass))
    key='C';                    // Set customer mode if valid credentials
else {
    puts("Invalid credentials!"); // Error message
    continue;                    // Restart login
}
free(usr); free(pass);          // Clean up input buffers

// ADMIN MODE
if(key=='A') {
    bye=0;                      // Reset exit flag
    while(1) {                  // Admin command loop
        adminMenu();            // Show admin options
        key=getKey();           // Get single character input

        // Handle empty database cases
        if(!db) {               // If no accounts exist
            if((key=='C')||(key=='Q')); // Only allow create or quit
            else {
                puts("Empty DataBase!!");
                continue;
            }
        }

        // Operations requiring account selection
        if(key=='H'||key=='W'||key=='D'||key=='T'||key=='B'||key=='F'||key=='U'||key=='X') {
            puts("=== Enter Customer/Sender Info ===");
            from=getAcc(db);     // Find account
            if(!from) {
                puts("Not found!! Try again.");
                continue;
            }
        }

        // Command Processing Switch
        switch(key) {
            case 'C': newAcc(&db); break;               // Create account
            case 'U': updateAcc(&db,from); break;       // Update account
            case 'H': statement(from); break;           // Show transactions
            case 'W': withdraw(from); break;            // Withdraw money
            case 'D': deposit(from); break;             // Deposit money
            case 'B': balance(from); break;             // Check balance
            case 'X':                                   // Card activation
                if(from->cardStat !=1) {                // If blocked
                    puts("Card status :BLOCKED");
                    puts("Activate card?(y/n)");
                    key=getKey();
                    if(key=='Y') {
                        from->cardStat=1;
                        puts("card is activated.");
                    }
                }
                else {                                  // If active
                    puts("Card status :ACTIVE");
                    puts("de-Activate card?(y/n)");
```

```
                    key=getKey();
                    if(key=='Y') {
                        from->cardStat=0;
                        puts("card is blocked.");
                    }
                }
                break;
            case 'T':                                   // Transfer funds
                puts("==:Enter Receiver's info:==");
                to=getAcc(db);                          // Find recipient
                if(!to) {
                    puts("Not found!! Try again.");
                    break;
                }
                transfer(from,to);                      // Execute transfer
                break;
            case 'E': database(db); break;              // Show all accounts
            case 'F': dispAcc(from); break;             // Show account details
            case 'Q':                                   // Quit
                saveData(db);                           // Save to files
                bye=1;                                  // Set exit flag
                break;
            default: puts("invalid option!.");          // Invalid input
        }
        if(bye) {                                       // Exit condition
            puts("Thank you for your work,Admin <3");
            break;
        }
    }
}
// CUSTOMER MODE
else if(key=='C') {
    bye=0;                      // Reset exit flag
    while(1) {                  // Customer command loop
        userMenu();             // Show customer options
        key=getKey();           // Get single character input

        // Command Processing Switch
        switch(key) {
            case 'H': statement(from); break;           // Show transactions
            case 'W': withdraw(from); break;            // Withdraw money
            case 'D': deposit(from); break;             // Deposit money
            case 'B': balance(from); break;             // Check balance
            case 'T':                                   // Transfer funds
                puts("==:Enter Receiver's info:==");
                to=getAcc(db);                          // Find recipient
                if(!to) {
                    puts("Not found!! Try again.");
                    break;
                }
                transfer(from,to);                      // Execute transfer
                break;
            case 'Q':                                   // Quit
                saveData(db);                           // Save to files
                bye=1;                                  // Set exit flag
                break;
            default: puts("invalid option!.");          // Invalid input
        }
        if(bye) {                                       // Exit condition
```

```
                puts("bye");
                break;
            }
        }
    }
}
}
```

Key Components Explained:

1. **Initialization**:
    - Creates data directory if missing (mkdir)
    - Loads existing account data (syncData)
2. **Authentication Flow**:
    - Checks for admin credentials first
    - Then checks regular customer accounts
    - Implements secure credential handling (immediate freeing of buffers)
3. **Admin Mode Features**:
    - Full account management (create/update)
    - Transaction processing (deposit/withdraw/transfer)
    - Card activation control
    - Database viewing options
4. **Customer Mode Features**:
    - Limited to own account operations
    - Basic transaction capabilities
    - Balance checking
5. **Memory Management**:
    - Proper freeing of username/password buffers
    - Clean exit with data saving
6. **Error Handling**:
    - Invalid credential detection
    - Empty database handling
    - Account not found cases
7. **User Interface**:
    - Clear menu prompts
    - Consistent input/output formatting
    - Confirmation dialogs for critical operations

ATM CONNECTED

atm.h

Header Guard and Debug Definition

```
#ifndef _ATM_H        // Header guard start - prevents multiple inclusions
#define _ATM_H        // Defines the header guard identifier

#define DBG           // Debug flag definition (can be used for debug output)
```

Standard Library Includes

```
#include<stdio.h>     // Standard I/O functions (printf, scanf)
#include<stdlib.h>    // Memory allocation, exit, etc.
#include<stdint.h>    // Standard integer types (uint64_t, etc.)
#include<string.h>    // String manipulation functions
#include<unistd.h>    // POSIX API (read, write, close)
#include<fcntl.h>     // File control options (for serial port)
#include<errno.h>     // Error number definitions
#include<time.h>      // Time functions (for timestamps)
#include<termios.h>   // Terminal I/O control (serial port configuration)
```

System Constants

```
#define MAX_DEPOSIT    30000    // Maximum deposit amount (₹30,000)
#define MAX_WITHDRAW   30000    // Maximum withdrawal amount (₹30,000)
#define MAX_TRANSFER  100000    // Maximum transfer amount (₹100,000)

#define NAME_LEN       30       // Maximum name length
#define MAX_PASS_LEN   20       // Maximum password length
#define MAX_USRN_LEN   20       // Maximum username length
```

Status Constants

```
#define BLOCKED 0              // Card blocked status
#define ACTIVE  1              // Card active status
```

Transaction Type Constants

```
#define WITHDRAW     1         // Withdrawal transaction type
#define DEPOSIT      2         // Deposit transaction type
#define TRANSFER_IN  3         // Incoming transfer type
#define TRANSFER_OUT 4         // Outgoing transfer type
```

Utility Macro

```
#define CAPS(ch) (ch &=~(32))  // Macro to convert character to uppercase
```

Type Definitions

```
typedef unsigned long long int u64;  // 64-bit unsigned integer type
typedef double f64;                  // 64-bit floating point type
```

Transaction Structure

```
typedef struct A {
    f64 amt;        // Transaction amount
```

```
    u64 id;         // Unique transaction ID
    char type;      // Transaction type (WITHDRAW/DEPOSIT/etc)
    struct A *nxt;  // Pointer to next transaction in linked list
} Tran;
```

Account Structure

```
typedef struct B {
    u64 num;                // Unique account number
    f64 bal;                // Account balance
    u64 phno;               // Phone number
    char usrName[MAX_USRN_LEN];  // Username
    char pass[MAX_PASS_LEN];     // Password
    char rfid[9];           // RFID card number (8 digits + null)
    char pin[5];            // 4-digit PIN + null
    int cardStat;           // Card status (ACTIVE/BLOCKED)
    char name[30];          // Account holder name
    Tran *tranHist;         // Transaction history linked list
    u64 tranCnt;            // Transaction count
    struct B *nxt;          // Pointer to next account in linked list
} Acc;
```

Serial Communication Functions

```
int initSerial();                    // Initialize serial port
void endSerial(const int fd);        // Close serial port
int tx_char(const int fd,const char ch);  // Transmit single character
int tx_str(const int fd,const char *str); // Transmit string
char rx_char(const int fd);          // Receive single character
void rx_str(const int fd,char *str,size_t len);  // Receive string
```

Message Processing Functions

```
int isMsgOk(const char *buf);        // Validate message format
void checkRFID(Acc *head,const int fd,const char *buf);  // Verify RFID card
void verifyPin(Acc *head,const int fd,const char *buf);  // Verify PIN
void act(Acc *head,const int fd,const char *buf);        // Process transaction
double extAmt(const char *buf);      // Extract amount from message
```

Banking Operations

```
void deposit(const int fd,Acc *usr,const f64 amt);    // Handle deposit
void withdraw(const int fd,Acc *usr,const f64 amt);   // Handle withdrawal
void balance(const int fd,Acc *usr);                  // Check balance
void pinChange(const int fd,Acc *usr,const char *pin);// Change PIN
void miniStatement(const int fd,Acc *usr,char txn);   // Get mini statement
```

Transaction Management

```
void addTran(Acc *usr,f64 amt,char type);  // Add transaction to history
u64 getTranId(Acc *usr);                   // Generate transaction ID
u64 getTimeStamp(void);                    // Get current timestamp
```

Hardware Control

```
void checkMC(const int fd);  // Check microcontroller connection
```

Data Persistence

```
void syncData(Acc **head);   // Load data from storage
void saveData(Acc *head);    // Save data to storage
void saveFile(Acc *head);    // Alternative save function
```

Account Lookup

```
Acc* getAcc(Acc *head,const char *rfid);  // Find account by RFID
```

Header Guard End

```
#endif  // End of header guard
```

Key Features of This Header:

1. **Complete Type System**: Defines all data structures needed for banking operations
2. **Serial Communication**: Functions for UART communication with hardware
3. **Transaction Processing**: All banking operation prototypes
4. **Data Management**: Functions for loading/saving account data
5. **Security Constants**: Defines limits and status values
6. **Hardware Abstraction**: Provides clean interface to hardware components

---

atm.c

Header Files

```
#include "atm.h"
#include <sys/stat.h>  // For stat() and mkdir()
#include <unistd.h>    // For POSIX functions
#define BAUD B9600
```

Serial Port Initialization

```
int initSerial(void) {
    int fd;  // File descriptor for serial port
    struct termios opt;  // Terminal options structure

    // Open serial port (blocking mode)
    fd = open("/dev/ttyUSB0", O_RDWR|O_NOCTTY);
    if(fd == -1) {
        perror("open");  // Print error if opening fails
        exit(1);
    }

    // Set file status flags (make reads blocking)
    fcntl(fd, F_SETFL, 0);

    // Get current port settings
    tcgetattr(fd, &opt);

    // Configure port for raw data mode
    cfmakeraw(&opt);
    cfsetispeed(&opt, BAUD);  // Set input baud rate
    cfsetospeed(&opt, BAUD);  // Set output baud rate

    // Set control flags:
    opt.c_cflag |= (CLOCAL | CREAD);  // Enable receiver
    opt.c_cflag &= ~PARENB;           // No parity
    opt.c_cflag &= ~CSTOPB;           // 1 stop bit
    opt.c_cflag &= ~CSIZE;            // Clear size bits
    opt.c_cflag |= CS8;               // 8 data bits
    opt.c_lflag &= ~(ICANON | ECHO | ECHOE | ISIG);  // Raw input
    opt.c_oflag &= ~OPOST;            // Raw output

    // Apply settings immediately
    tcsetattr(fd, TCSANOW | TCSAFLUSH, &opt);

    usleep(10000);  // 10ms delay for settings to take effect
    return fd;      // Return file descriptor
}
```

Serial Communication Functions

```
void flushSerial(const int fd) {
    tcflush(fd, TCIOFLUSH);  // Flush both input and output buffers
}

void endSerial(const int fd) {
    close(fd);  // Close serial port
}

int tx_char(const int fd, const char ch) {
```

```
    return write(fd, &ch, 1);  // Send single character
}

int tx_str(const int fd, const char *str) {
    write(fd, str, strlen(str));  // Send string
    write(fd, "\r\n", 2);         // Append CRLF
    #ifdef DBG
    printf("DBG_TX:%s\n",str);    // Debug output
    #endif
}

char rx_char(const int fd) {
    char ch;
    if(read(fd, &ch, 1) != 1) return -1;  // Read single char or error
    return ch;
}

void rx_str(const int fd, char *str, size_t len) {
    int i=0, j=0;
    char ch;
    for(; i<len-1; i++) {
        j = read(fd, &ch, 1);  // Read character
        if(j == -1) {           // Error case
            printf("i=%d\n",i);
            perror("read_str");
            exit(1);
        } else if(j == 0) {     // No data
            i--;
            continue;
        }
        str[i] = ch;            // Store character
        if(ch == '\n') break;   // End on newline
    }
    str[i-1] = '\0';            // Replace \r with null terminator
    #ifdef DBG
    printf("DBG_RX:%s\n",str);  // Debug output
    #endif
}
```

Message Processing

```
int isMsgOk(const char *buf) {
    // Check message format: starts with # and ends with $
    if((buf[0] == '#') && (buf[strlen(buf)-1] == '$'))
        return 1;
    return 0;
}

void checkRFID(Acc *head, const int fd, const char *buf) {
    char rfid[9], temp[32];
    strncpy(rfid, buf+3, 8);  // Extract RFID from message
    rfid[8] = '\0';

    Acc *usr = getAcc(head, rfid);  // Find account

    #ifdef INT
    checkMC(fd);  // Check microcontroller connection
    #endif
```

```
    if(usr) {
        if(usr->cardStat) {  // Check card status
            sprintf(temp,"@OK:ACTIVE:%s$",usr->usrName);
            tx_str(fd,temp);
        } else {
            tx_str(fd,"@ERR:BLOCK$");
        }
    } else {
        tx_str(fd,"@ERR:INVALID$");
    }
}
void verifyPin(Acc *head,const int fd,const char *buf)
{
        //#V:<rfid>:<pin>$
        //verify rfid with pin
        char rfid[9];
        char pin[5];

        strncpy(rfid,buf+3,8);
        rfid[8]='\0';
        strncpy(pin,buf+12,4);
        pin[4]='\0';

        Acc *usr=getAcc(head,rfid);
#ifdef INT
        checkMC(fd);
#endif
        if(!strcmp(pin,usr->pin)){
                tx_str(fd,"@OK:MATCHED$");

        }
        else
        {
                tx_str(fd,"@ERR:WRONG$");

        }
}

void act(Acc *head,const int fd,const char *buf)
{
        char rfid[9];
        char req[4];
        char pin[5];
        double amt=0;
        char txn=0;

        //extract rfid
        strncpy(rfid,buf+7,8);
        rfid[8]='\0';
        //get Acc
        Acc *usr=getAcc(head,rfid);
        //extract req
        strncpy(req,buf+3,3);
        req[4]='\0';

        printf("req=%s,rfid=%s\n",req,rfid);

        if(!strcmp(req,"WTD")){
                amt=extAmt(buf);
```

```
                withdraw(fd,usr,amt);
                saveData(head);
        }
        else if(!strcmp(req,"DEP"))
        {
                amt=extAmt(buf);
                deposit(fd,usr,amt);
                saveData(head);
        }
        else if(!strcmp(req,"BAL"))
        {
                balance(fd,usr);
        }
        else if(!strcmp(req,"MST"))
        {
                //mini statement
                txn=buf[16]-'0';
                miniStatement(fd,usr,txn);
        }
        else if(!strcmp(req,"TNF"))
        {

        }
        else if(!strcmp(req,"PIN"))
        {
                strncpy(pin,buf+16,4);
                pinChange(fd,usr,pin);
                saveData(head);
        }else if(!strcmp(req,"BLK"))
        {
                usr->cardStat=BLOCKED;
#ifdef INT
                checkMC(fd);
#endif
                tx_str(fd,"@OK:DONE$");
                saveData(head);
        }
        else
        {

        }
}

double extAmt(const char *buf)
{
        char dup[20];
        strcpy(dup,buf+16);
        dup[strlen(dup)-1]='\0';
        return atof(dup);
}

Acc* getAcc(Acc *head,const char *rfid)
{
        while(head)
        {
                if(!strcmp(rfid,head->rfid))
                        break;
                head=head->nxt;
        }
```

```
        return head;
}
```

Account Operations

```
void deposit(const int fd, Acc *usr, const f64 amt) {
    #ifdef INT
    checkMC(fd);
    #endif

    if(amt <= 0) {
        tx_str(fd,"@ERR:NEGAMT$");
    } else if(amt < MAX_DEPOSIT) {
        usr->bal += amt;  // Update balance
        addTran(usr, +amt, DEPOSIT);  // Record transaction
        tx_str(fd,"@OK:DONE$");
    } else {
        tx_str(fd,"@ERR:MAXAMT$");
    }
}

void withdraw(const int fd, Acc *usr, const f64 amt) {
    #ifdef INT
    checkMC(fd);
    #endif

    if(amt <= 0) {
        tx_str(fd,"@ERR:NEGAMT$");
    } else if(amt < MAX_WITHDRAW) {
        if(amt <= usr->bal) {  // Check sufficient funds
            usr->bal -= amt;
            addTran(usr, -amt, WITHDRAW);
            tx_str(fd,"@OK:DONE$");
        } else {
            tx_str(fd,"@ERR:LOWBAL$");
        }
    } else {
        tx_str(fd,"@ERR:MAXAMT$");
    }
}
void balance(const int fd,Acc *usr)
{
        //#A:BAL:<rfid>$ -> @OK:BAL=<amt>$
        puts("in bal.");
        char buf[50];
        sprintf(buf,"@OK:BAL=%.2lf$",usr->bal);
#ifdef INT
        checkMC(fd);
#endif
        tx_str(fd,buf);

}

void pinChange(const int fd,Acc *usr,const char *pin)
{
        //#A:PIN:<rfid>:<pin>$   -> @OK:DONE$
        strcpy(usr->pin,pin);
#ifdef INT
        checkMC(fd);
```

```
#endif
        tx_str(fd,"@OK:DONE$");

}

void miniStatement(const int fd,Acc *usr,char txn)
{
        //#A:MST:<rfid>:<txNo>$ -> @TXN:<type>:<ddmmyyyyhhmm>:<amt>$
        char buf[70];
        u64 dum;
        double amt;
        unsigned int dd,mon,yy,hh,mm;
        if(txn<=usr->tranCnt)
        {
                Tran *t=usr->tranHist;
                while(--txn)
                {
                        t=t->nxt;
                }
                dum=(t->id)/100000;
                mm=dum%100;
                dum/=100;
                hh=dum%100;
                dum/=100;
                dd=dum%100;
                dum/=100;
                mon=dum%100;
                dum/=100;
                yy=dum;
                amt=t->amt;
                amt=(amt<0)?-amt:amt;
                sprintf(buf,"@TXN:%d:%02u/%02u/%04u %02u:%02u:%.2lf$",t->type,dd,mon,yy,hh,mm,amt);
#ifdef INT
                checkMC(fd);
#endif
                tx_str(fd,buf);
        }
        else
        {
#ifdef INT
                checkMC(fd);
#endif
                tx_str(fd,"@TXN:7:0:0$");
        }
}
```

Transaction Management

```
void addTran(Acc *usr, f64 amt, char type) {
    Tran *new = calloc(1, sizeof(Tran));  // Allocate new transaction
    new->amt = amt;
    new->id = getTranId(usr);  // Generate unique ID
    new->type = type;
    new->nxt = usr->tranHist;  // Add to history list
    usr->tranHist = new;       // Update head pointer
    (usr->tranCnt)++;          // Increment count
}

u64 getTranId(Acc *usr) {
```

```
    srand(usr->num);  // Seed with account number
    return getTimeStamp()*1000 + (rand()%1000);  // Timestamp + random
}

u64 getTimeStamp(void) {
    time_t rawtime;
    struct tm *timeinfo;
    time(&rawtime);  // Get current time
    timeinfo = localtime(&rawtime);

    // Format as YYYYMMDDHHMMSS
    return (timeinfo->tm_year+1900)*10000000000ULL +
           (timeinfo->tm_mon+1)*100000000ULL +
           timeinfo->tm_mday*1000000ULL +
           timeinfo->tm_hour*10000ULL +
           timeinfo->tm_min*100ULL +
           timeinfo->tm_sec;
}
void checkMC(const int fd)
{
        char buf[20];
        while(1){
                tx_str(fd,"@Y:LINEOK$");
                rx_str(fd,buf,sizeof(buf));
                if(!strcmp(buf,"#Y:LINEOK$"))break;
        }
}
```

Data Persistence

```
void saveData(Acc *head) {
    // Create data directory if needed
    struct stat st = {0};
    if(stat("../data", &st) == -1) {
        mkdir("../data", 0700);
    }

    // Open files
    FILE *userFile = fopen("../data/users.txt", "w");
    FILE *transFile = fopen("../data/transactions.txt", "w");

    // Write headers
    fprintf(userFile, "RFID,PIN,Name,Balance,AccountNumber,Phone,Username,CardStatus\n");
    fprintf(transFile, "RFID,Type,Amount,DateTime,TransactionID\n");

    // Write all accounts and transactions
    Acc *current = head;
    while(current) {
        // Handle empty name case
        if(current->name[0] == '\0') {
            strncpy(current->name, "DEFAULT", NAME_LEN-1);
            current->name[NAME_LEN-1] = '\0';
        }

        // Write account data
        fprintf(userFile, "%s,%s,%s,%.2f,%llu,%llu,%s,%d\n",
                current->rfid, current->pin, current->name, current->bal,
                current->num, current->phno, current->usrName,
                current->cardStat);
```

```
        // Write all transactions
        Tran *t = current->tranHist;
        while(t) {
            time_t rawtime = time(NULL);
            struct tm *timeinfo = localtime(&rawtime);

            fprintf(transFile, "%s,%d,%.2f,%04d-%02d-%02d %02d:%02d:%02d,%llu\n",
                    current->rfid, t->type, t->amt,
                    timeinfo->tm_year + 1900, timeinfo->tm_mon + 1,
                    timeinfo->tm_mday, timeinfo->tm_hour,
                    timeinfo->tm_min, timeinfo->tm_sec,
                    t->id);
            t = t->nxt;
        }
        current = current->nxt;
    }
    fclose(userFile);
    fclose(transFile);
}
```

Data Loading

```
void syncData(Acc **head) {
    *head = NULL;  // Initialize empty list
    Acc *tail = NULL;

    // Open users file
    FILE *userFile = fopen("../data/users.txt", "r");
    if(!userFile) {
        perror("Failed to open users file");
        return;
    }

    // Skip header
    char line[512];
    fgets(line, sizeof(line), userFile);

    // Read user records
    while(fgets(line, sizeof(line), userFile)) {
        Acc *newUser = calloc(1, sizeof(Acc));

        // Initialize all fields
        memset(newUser->usrName, 0, MAX_USRN_LEN);
        memset(newUser->pass, 0, MAX_PASS_LEN);
        memset(newUser->rfid, 0, 9);
        memset(newUser->pin, 0, 5);
        memset(newUser->name, 0, NAME_LEN);

        // Parse CSV line
        if(sscanf(line, "%8[^,],%4[^,],%29[^,],%lf,%llu,%llu,%19[^,],%d",
                newUser->rfid, newUser->pin, newUser->name, &newUser->bal,
                &newUser->num, &newUser->phno, newUser->usrName,
                &newUser->cardStat) != 8) {
            free(newUser);
            continue;
        }

        newUser->tranHist = NULL;
```

```
        newUser->tranCnt = 0;
        newUser->nxt = NULL;

        // Add to linked list
        if(!*head) {
            *head = newUser;
        } else {
            tail->nxt = newUser;
        }
        tail = newUser;
    }
    fclose(userFile);

    // Load transactions if users exist
    if(*head) {
        FILE *transFile = fopen("../data/transactions.txt", "r");
        if(!transFile) return;

        // Skip header
        fgets(line, sizeof(line), transFile);

        // Read transaction records
        while(fgets(line, sizeof(line), transFile)) {
            char rfid[9];
            int type;
            double amount;
            char datetime[20];
            u64 transId;

            if(sscanf(line, "%8[^,],%d,%lf,%19[^,],%llu",
                    rfid, &type, &amount, datetime, &transId) != 5) {
                continue;
            }

            // Find matching account
            Acc *user = *head;
            while(user && strcmp(user->rfid, rfid)) {
                user = user->nxt;
            }

            if(!user) continue;

            // Add transaction
            Tran *newTrans = calloc(1, sizeof(Tran));
            newTrans->type = type;
            newTrans->amt = amount;
            newTrans->id = transId;
            newTrans->nxt = user->tranHist;
            user->tranHist = newTrans;
            user->tranCnt++;
        }
        fclose(transFile);
    }
}
```

This implementation provides a complete ATM system with:

- Serial communication for hardware interfacing
- Account and transaction management

- Data persistence to files
- Robust error handling
- Debugging support
- Secure operations with validation

---

atmMain.c

Header Inclusion

```
#include "atm.h"  // Includes all necessary definitions and prototypes
```

Main Function

```
int main() {
```

Variable Declarations

```
    int fd, type;        // fd: serial port file descriptor, type: unused in this scope
    char buf[100];       // Buffer for receiving serial messages
    Acc *db = NULL;      // Database head pointer initialized to NULL
```

Database Synchronization

```
    syncData(&db);       // Load account data from files into memory
    #ifdef DBG           // Debug mode check
    puts("synced");      // Print synchronization confirmation
    #endif
```

Serial Port Initialization

```
    fd = initSerial();   // Initialize serial communication
    #ifdef DBG
    puts("super loop");  // Debug message indicating main loop start
    #endif
```

Main Processing Loop

```
   while(1) {           // Infinite loop for continuous operation
```

Message Reception

```
      rx_str(fd, buf, sizeof(buf));  // Receive message from serial port
```

Message Validation

```
        if(!isMsgOk(buf)) continue;     // Skip if message format is invalid
        // Valid message format: #<opt>:<data>$
```

Message Processing Switch

```
        switch(buf[1]) {  // Check second character for operation type
```

RFID Check Case

```
            case 'C':     // RFID verification request
```

```
                checkRFID(db, fd, buf);  // Validate RFID card
                break;
```

PIN Verification Case

```
            case 'V':     // PIN verification request
                verifyPin(db, fd, buf);  // Validate PIN against RFID
                break;
```

Action Processing Case

```
            case 'A':     // Transaction request
                puts("acting.");        // Debug output
                act(db, fd, buf);       // Process transaction (deposit/withdraw/etc)
                break;
```

Connection Check Case

```
            case 'X':     // Connection test
                tx_str(fd, "@X:LINEOK$");  // Respond with connection OK
                break;
```

Save Data Case

```
            case 'Q':     // Save command
                saveData(db);           // Write all data to files
                puts("data saved");     // Debug confirmation
                break;
        }
    }
}
```

Key Components Explained:

1. **Initialization**:
    - Loads account data from storage (syncData)
    - Establishes serial communication (initSerial)
2. **Main Loop**:
    - Continuously listens for serial messages
    - Validates message format (must start with # and end with $)
    - Routes messages to appropriate handlers based on operation code
3. **Operation Handling**:
    - **'C'**: RFID verification - checks if card exists and is active
    - **'V'**: PIN verification - validates PIN against stored value
    - **'A'**: Transaction processing - handles deposits, withdrawals, etc.
    - **'X'**: Connection test - responds to keep-alive checks
    - **'Q'**: Data saving - writes all account data to disk
4. **Error Handling**:
    - Invalid messages are silently discarded (continue)
    - All operations include validation before processing

5. **Debug Support**:
    - Conditional debug output when DBG is defined
    - Status messages for key operations

MICROCONTROLLER SIDE

Type Aliases (types.h)

```
//Header Guard
#ifndef _TYPES_H       // Checks if _TYPES_H is not defined (prevents multiple inclusions)
#define _TYPES_H       // Defines _TYPES_H if it hasn't been defined yet

//Type Aliases (Platform-Independent Sizing)
// Unsigned 8-bit integer (0 to 255)
typedef unsigned char u8;

// Signed 8-bit integer (-128 to 127)
typedef char s8;

// Unsigned 16-bit integer (0 to 65,535)
typedef unsigned short u16;

// Signed 16-bit integer (-32,768 to 32,767)
typedef short s16;

// Unsigned 32-bit integer (0 to 4,294,967,295)
typedef unsigned int u32;

// Signed 32-bit integer (-2,147,483,648 to 2,147,483,647)
typedef int s32;

// 32-bit floating point (single precision)
typedef float f32;

// Volatile unsigned 32-bit integer (for hardware registers)
typedef volatile u32 vu32;

//Boolean Constants
#define HIGH 1         // Logical high state (typically 3.3V/5V)
#define LOW  0         // Logical low state (typically 0V/GND)

//Header Guard Closure
#endif                // Ends the conditional inclusion block
```

Key Features:

1. **Standardized Type Sizes**:
   - Provides consistent type definitions across different platforms/compilers
   - Eliminates ambiguity about data sizes (especially important for embedded systems)
2. **Naming Convention**:
   - u prefix for unsigned types
   - s prefix for signed types
   - Number indicates bit width (8, 16, 32)
   - v prefix for volatile types
3. **Volatile Keyword**:
   - vu32 is specifically for hardware registers that can change outside program flow
   - Prevents compiler optimization that might skip memory reads

4. **Common Constants**:
    - Standard HIGH/LOW definitions for digital I/O operations
5. **Portability**:
    - Abstracts away compiler-specific type sizes
    - Makes code easier to port between architectures

Header File (delay.h)

```
#ifndef _DELAY_H       // Header guard start - prevents multiple inclusions
#define _DELAY_H       // Defines the header guard identifier

#include "types.h"     // Includes custom type definitions (like u32 for unsigned 32-bit)

// Function prototypes:
void delayS(u32);      // Declaration for second delay function
void delayMs(u32 dly); // Declaration for millisecond delay function
void delayMs(u32 dly); // Declaration for microsecond delay function

#endif                // End of header guard
```

Implementation File (Software Delays)(delay.c)

```
#include "types.h"     // Includes the same type definitions as header

// Seconds delay function
void delayS(u32 dly) {
    dly *= 12000000;   // Convert seconds to loop iterations (for 12MHz clock)
    while(dly--);      // Busy-wait loop that decrements until zero
}

// Milliseconds delay function
void delayMs(u32 dly) {
    dly *= 12000;      // Convert milliseconds to loop iterations (12MHz clock)
    while(dly--);      // Busy-wait loop
}

// Microseconds delay function
void delayUs(u32 dly) {
    dly *= 12;         // Convert microseconds to loop iterations (12MHz clock)
    while(dly--);      // Busy-wait loop
}
```

Key Points:

1. **Clock Frequency Assumption**:
    - The magic numbers (12000000, 12000, 12) assume:
        - 12MHz processor clock frequency
        - Each loop iteration takes approximately 1 clock cycle
2. **Delay Calculations**:
    - delayS: 12,000,000 iterations ≈ 1 second (12MHz × 1s)

- delayMs: 12,000 iterations ≈ 1ms (12MHz × 0.001s)
- delayUs: 12 iterations ≈ 1μs (12MHz × 0.000001s)

3. **Implementation Characteristics**:
   - Busy-wait delays (blocking implementation)
   - No hardware timer usage (pure software)
   - Accuracy depends on:
     - Exact CPU clock speed
     - Compiler optimization
     - Pipeline effects in the processor
4. **Usage Considerations**:
   - These are approximate delays
   - Not suitable for precision timing
   - Will vary with different clock speeds
   - Consumes CPU cycles while waiting
5. **Improvement Suggestions**:
   - Use hardware timers for more accurate delays
   - Make clock frequency configurable
   - Add calibration routine
   - Implement non-blocking versions

---

gpio.h

Header Guard

```
#ifndef _GPIO_H       // Prevents multiple inclusions
#define _GPIO_H       // Defines the header guard
```

Includes and Basic Definitions

```
#include "types.h"    // Includes custom type definitions
#define OUT 1         // Direction output definition
#define IN  0         // Direction input definition
```

GPIO Base Addresses

```
#define GPIO0 ((vu32 *)0xE0028000)  // Base address for GPIO Port 0
#define GPIO1 ((vu32 *)0xE0028010)  // Base address for GPIO Port 1
```

Single Pin Operations

```
// Initialize pin direction (dir = IN/OUT)
#define INITPIN(portAdd,pin,dir) (*(portAdd+2)=(((*(portAdd+2))&~(1<<pin))|(dir<<pin))

// Set pin high
#define SETPIN(portAdd,pin) (*(portAdd+1)= (1<<pin))

// Clear pin low
#define CLRPIN(portAdd,pin) (*(portAdd+3)= (1<<pin))
```

```
// Read pin value (returns 0 or 1)
#define READPIN(portAdd,pin) (((*portAdd)>>pin)&1)

// Toggle pin state
#define TGLPIN(portAdd,pin) ((*portAdd) ^=(1<<pin))

// Write value (0/1) to pin
#define WRITEPIN(portAdd,pin,val) (*portAdd=(((*portAdd)&~(1<<pin))|(val<<pin)))
```

4-Pin Operations (Nibble)

```
// Initialize 4 consecutive pins
#define INIT4PINS(portAdd,pin,dir) (*(portAdd+2)=(((*(portAdd+2))&~(0xF<<pin))|(dir<<pin))

// Read 4 consecutive pins
#define READ4PINS(portAdd,pin) (((*portAdd)>>pin)&0xF)

// Write to 4 consecutive pins
#define WRITE4PINS(portAdd,lpin,data) ((*portAdd)=(((*portAdd)&~(0xF<<lpin))|(data<<lpin)))
```

8-Pin Operations (Byte)

```
// Initialize 8 consecutive pins
#define INIT8PINS(portAdd,pin,dirVal) (*(portAdd+2)=(((*(portAdd+2))&~(0xFF<<pin))|(dirVal<<pin))

// Set 8 consecutive pins high
#define SET8PINS(portAdd,lpin) (*(portAdd+1)= (0xFF<<lpin))

// Clear 8 consecutive pins low
#define CLR8PINS(portAdd,lpin) (*(portAdd+3)= (0xFF<<lpin))

// Read 8 consecutive pins
#define READ8PINS(portAdd,pin) (((*portAdd)>>pin)&0xFF)

// Write to 8 consecutive pins
#define WRITE8PINS(portAdd,lpin,data) (*portAdd=(((*portAdd)&~(0xFF<<lpin))|(data<<lpin)))
```

Header Guard End

```
#endif                // End of header guard
```

Key Technical Details:

1. **Memory Mapping**:
   - portAdd+0 = PIN register (input values)
   - portAdd+1 = SET register (set output high)
   - portAdd+2 = DIR register (direction control)
   - portAdd+3 = CLR register (clear output low)
2. **Bit Manipulation**:
   - All macros use atomic read-modify-write operations
   - Bit masking (&~) clears target bits before setting new values
   - Shifting (<<) positions bits correctly
3. **Volatile Access**:

- Uses vu32 type to prevent compiler optimization of hardware accesses

4. **Multi-pin Operations**:
   - Supports efficient handling of groups of pins (4/8 bits)
   - Useful for parallel interfaces and bus operations

---

lcd.h

Header Guard

```
#ifndef _LCD_H       // Prevents multiple inclusions
#define _LCD_H       // Defines the header guard
```

Includes and Pin Definitions

```
#include "types.h"   // Includes custom type definitions

// Data bus width configurations
#define LCD_PINS_4B 20  // P1.[23-20] for 4-bit mode (D7-D4)
#define LCD_PINS_8B 16  // P1.[23-16] for 8-bit mode (D7-D0)

// Control line pin assignments
#define LCD_RS 16    // Register Select (P0.16)
#define LCD_RW 26    // Read/Write (P1.26)
#define LCD_EN 17    // Enable (P0.17)

// Command/Data modes
#define LCD_DATA 1   // Data mode
#define LCD_CMD 0    // Command mode
#define LCD_INT 2    // Intermediate mode (for initialization)
```

LCD Configuration

```
// Default configuration
#define LCD 32       // 16x2 LCD (32 characters total)
#define LCD_MODE 8   // Default to 8-bit mode
```

Command Set Macros

```
// Initialization commands
#define LCD_INIT8 0x30  // 8-bit init
#define LCD_INIT4 0x20  // 4-bit init

// Basic commands
#define LCD_CLR 0x01    // Clear display
#define LCD_HOMECUR 0x02 // Home cursor

// Cursor/Display control
#define LCD_SETCURINC 0x06      // Cursor increment
#define LCD_SETDISPSHIFT 0x05   // Display shift

// Display state commands
#define LCD_DISPON 0x0C             // Display on
#define LCD_DISPON_BLKON 0x0D       // Display on + blink
#define LCD_DISPON_CURON 0x0E       // Display on + cursor
#define LCD_DISPON_CURON_BLKON 0x0F // Display on + cursor + blink
```

```
// Movement commands
#define LCD_DISPSHIFT_R 0x1C  // Shift display right
#define LCD_DISPSHIFT_L 0x18  // Shift display left
#define LCD_CURSHIFT_R 0x14   // Move cursor right
#define LCD_CURSHIFT_L 0x10   // Move cursor left
```

LCD Dimension Configuration

```
#if LCD_MODE==8
    #define LCD_16x1 0x30  // 8-bit 16x1 config
    #define LCD_16x2 0x3C  // 8-bit 16x2 config
    #define LCD_20x4 0x38  // 8-bit 20x4 config
#elif LCD_MODE==4
    #define LCD_16x1 0x20  // 4-bit 16x1 config
    #define LCD_16x2 0x28  // 4-bit 16x2 config
    #define LCD_20x4 0x28  // 4-bit 20x4 config
#endif

// Active configuration based on LCD size
#if LCD==16
    #define LCD_DIM LCD_16x1
    #define LCD_MAX_ROWS 1
    #define LCD_MAX_COLS 16
#elif LCD==32
    #define LCD_DIM LCD_16x2
    #define LCD_MAX_ROWS 2
    #define LCD_MAX_COLS 16
#elif LCD==80
    #define LCD_DIM LCD_20x4
    #define LCD_MAX_ROWS 4
    #define LCD_MAX_COLS 20
#endif
```

Memory Addressing

```
#define LCD_CGRAM 0x40  // Character Generator RAM address
#define LCD_ROW0 0x80   // DDRAM address row 0
#define LCD_ROW1 0xC0   // DDRAM address row 1
#define LCD_ROW2 0x94   // DDRAM address row 2 (for 4-line displays)
#define LCD_ROW3 0xD4   // DDRAM address row 3 (for 4-line displays)
```

Function Declarations

```
// Initialization
void initLcd(void);

// Core functions
void writeLcd(s8 word, s16 dc);  // Write data/command
void moveLcdCursor(s16 row, s16 col); // Position cursor

// Character generation
void makeNewCharAt(const s8 *pixels, s16 asc); // Create custom character

// String operations
void char2Lcd(s8 ch);            // Write single character
void str2Lcd(const s8 *str);     // Write string

// Display management
void clearLcdDisplay(void);      // Clear entire display
```

```
void clearLcdRow(s16 row);       // Clear specific row

// Special functions
void scrllStr2Lcd(s8 *str, s16 row); // Scrolling text display
```

Header Guard End

```
#endif  // End of header guard
```

Key Technical Details:

1. **Flexible Configuration**:
   - Supports both 4-bit and 8-bit interface modes
   - Configurable for different LCD sizes (16x1, 16x2, 20x4)
2. **Complete Command Set**:
   - Includes all standard HD44780 controller commands
   - Organized by functional groups (initialization, cursor control, etc.)
3. **Memory Addressing**:
   - Provides addresses for all DDRAM rows
   - Includes CGRAM access for custom characters
4. **Hardware Abstraction**:
   - Pin definitions abstract physical connections
   - Functions provide high-level control
5. **Advanced Features**:
   - Custom character generation
   - Scrolling text display
   - Row-specific operations

---

lcd.c

Includes and Custom Character Definition

```
#include "types.h"   // Basic type definitions
#include "delay.h"   // Delay functions
#include "gpio.h"    // GPIO control macros
#include "lcd.h"     // LCD definitions

// Example custom character (commented out)
// const s8 bell[] = {0x04,0x0E,0x0E,0x0E,0x1F,0x00,0x04,0x00};
```

LCD Initialization Function

```
void initLcd() {
    // Configure control pins as outputs
    INITPIN(GPIO0, LCD_RS, 1);  // Register Select
    // INITPIN(GPIO1, LCD_RW, 1); // Read/Write (commented out)
    INITPIN(GPIO0, LCD_EN, 1);  // Enable

    // Configure data bus based on mode
    #if LCD_MODE==4
        INIT4PINS(GPIO1, LCD_PINS_4B, 0xF);  // 4-bit mode
```

```
    #elif LCD_MODE==8
        INIT8PINS(GPIO1, LCD_PINS_8B, 0xFF); // 8-bit mode
    #endif

    // Initialization sequence
    delayMs(15);  // Power-on delay

    // Send initialization commands (3 times for stability)
    writeLcd(LCD_INIT8, LCD_CMD);  // 8-bit mode set
    delayMs(5);
    writeLcd(LCD_INIT8, LCD_CMD);
    delayUs(200);
    writeLcd(LCD_INIT8, LCD_CMD);
    delayUs(200);

    // Switch to 4-bit mode if configured
    #if LCD_MODE==4
        writeLcd(LCD_INIT4, LCD_INT);  // Special intermediate mode
    #endif

    // Final configuration
    writeLcd(LCD_DIM, LCD_CMD);      // Set interface length
    writeLcd(LCD_DISPON, LCD_CMD);   // Turn display on
    writeLcd(LCD_CLR, LCD_CMD);      // Clear display
    writeLcd(LCD_SETCURINC, LCD_CMD);// Set cursor direction
    delayMs(2);
}
```

Core Write Function

```
void writeLcd(s8 word, s16 dc) {
    // Set Register Select pin
    if(dc == 1) SETPIN(GPIO0, LCD_RS);    // Data mode
    else if(dc == 0) CLRPIN(GPIO0, LCD_RS); // Command mode
    else if(dc == 2) {                     // Initialization mode
        CLRPIN(GPIO0, LCD_RS);
        // Write upper nibble only for init
        WRITE4PINS(GPIO1, LCD_PINS_4B, (word >> 4));
        // Generate enable pulse
        SETPIN(GPIO0, LCD_EN);
        delayUs(1);
        CLRPIN(GPIO0, LCD_EN);
        return;
    }

    // Data transfer based on mode
    #if LCD_MODE==8
        // 8-bit mode - write full byte
        WRITE8PINS(GPIO1, LCD_PINS_8B, word);
        // Generate enable pulse
        SETPIN(GPIO0, LCD_EN);
        delayUs(1);
        CLRPIN(GPIO0, LCD_EN);
    #elif LCD_MODE==4
        // 4-bit mode - write upper nibble
        WRITE4PINS(GPIO1, LCD_PINS_4B, (word >> 4));
        SETPIN(GPIO0, LCD_EN);
        delayUs(1);
        CLRPIN(GPIO0, LCD_EN);
```

```
        // Write lower nibble
        WRITE4PINS(GPIO1, LCD_PINS_4B, (word & 0x0F));
        SETPIN(GPIO0, LCD_EN);
        delayUs(1);
        CLRPIN(GPIO0, LCD_EN);
    #endif

    delayUs(60);  // Processing delay
}
```

Character and String Functions

```
// Create custom character in CGRAM
void makeNewCharAt(const s8 *pixels, s16 asc) {
    u16 i;
    if(asc > 8) return;  // Only 8 custom characters supported

    // Set CGRAM address
    writeLcd(LCD_CGRAM + asc*8, LCD_CMD);
    // Write character pattern (8 bytes)
    for(i = 0; i < 8; i++)
        char2Lcd(pixels[i]);
}

// Write single character
void char2Lcd(s8 ch) {
    writeLcd(ch, LCD_DATA);
}

// Write null-terminated string
void str2Lcd(const s8 *str) {
    while(*str) {
        writeLcd(*(str++), LCD_DATA);
    }
}
```

Cursor Control Functions

```
// Position cursor at specific row and column
void moveLcdCursor(s16 row, s16 col) {
    if(row == 0) writeLcd(LCD_ROW0 + col, LCD_CMD);  // Row 1
    else if(row == 1) writeLcd(LCD_ROW1 + col, LCD_CMD); // Row 2
}

// Clear entire display
void clearLcdDisplay() {
    writeLcd(0x01, LCD_CMD);  // Clear command
    delayMs(2);
}

// Clear specific row by filling with spaces
void clearLcdRow(s16 row) {
    u8 col;
    // Move to row start
    if(row == 0) writeLcd(LCD_ROW0, LCD_CMD);
    else if(row == 1) writeLcd(LCD_ROW1, LCD_CMD);

    // Fill row with spaces
    for(col = 0; col < LCD_MAX_COLS; col++) {
```

```
        writeLcd(' ', LCD_DATA);
    }

    // Return cursor to row start
    if(row == 0) writeLcd(LCD_ROW0, LCD_CMD);
    else if(row == 1) writeLcd(LCD_ROW1, LCD_CMD);
}
```

Scrolling Text Function

```
void scrllStr2Lcd(s8* str, s16 row) {
    s32 len = 0, i = 0, j, k, l, m;
    s8 *temp = str;

    // Calculate string length
    while(*temp) {
        len++;
        temp++;
    }

    // Infinite scrolling loop
    while(1) {
        j = LCD_MAX_COLS - 1 - ((i++) % LCD_MAX_COLS);

        if(j) {
            // Right-aligned scrolling
            m = 1;
        } else if(j == 0) {
            // Left-aligned scrolling
            m = len;
        }

        for(l = 0; l < m; l++) {
            moveLcdCursor(row, j);
            // Print visible portion of string
            for(k = 0; k < LCD_MAX_COLS - j; k++) {
                if(k + l >= len) break;
                char2Lcd(str[k + l]);
            }
            delayMs(60);  // Scroll speed
            clearLcdRow(row);
        }
    }
}
```

Key Technical Details:

1. **Initialization Sequence**:
    - Follows strict HD44780 timing requirements
    - Includes power-on delay and multiple reset commands
    - Handles both 4-bit and 8-bit modes
2. **Data Transfer**:
    - Atomic operations for reliable communication
    - Proper timing with enable pulses
    - Support for both command and data modes

3. **Memory Management**:
    - DDRAM addressing for display positions
    - CGRAM access for custom characters
    - Automatic cursor positioning
4. **Optimizations**:
    - Minimal delays for maximum performance
    - Efficient string handling
    - Hardware-aware bit manipulation

---

kpm.h

Header Guard

```
#ifndef _KPM_H       // Prevents multiple inclusions
#define _KPM_H       // Defines the header guard
```

Includes

```
#include "types.h"   // Includes custom type definitions (u8, s32, etc.)
```

Keypad Configuration Macros

```
#define ROWS 4       // Number of rows in keypad (output lines)
#define COLS 4       // Number of columns in keypad (input lines)

// Row pin assignments (outputs)
#define KPM_R0 24    // Row 0 connected to P1.24
#define KPM_R1 25    // Row 1 connected to P1.25
#define KPM_R2 26    // Row 2 connected to P1.26
#define KPM_R3 27    // Row 3 connected to P1.27

// Column pin assignments (inputs)
#define KPM_C0 28    // Column 0 connected to P1.28
#define KPM_C1 29    // Column 1 connected to P1.29
#define KPM_C2 30    // Column 2 connected to P1.30
#define KPM_C3 31    // Column 3 connected to P1.31
```

Key Label Declarations

```
// 2D key labels (for single keypad layer)
extern s32 label2D[ROWS][COLS];

// 3D key labels (for multiple keypad layers)
extern s32 label3D[3][COLS][COLS];
```

Function Prototypes

```
// Initializes the keypad GPIO pins
void initKpm4x4(void);

// Reads a key press from standard 4x4 keypad
// Returns the label from label2D array
s32 readKpm4x4(void);
```

```
// Reads from a 4x4x3 multilayer keypad
// Returns the label from label3D array
s32 readKpm4x4x4(void);
```

Header Guard End

```
#endif              // End of header guard
```

Key Technical Details:

1. **Pin Configuration**:
    - Rows are outputs (drive high during scanning)
    - Columns are inputs (with pull-down resistors)
    - Pins are on Port 1 (P1.24-P1.31)
2. **Key Labeling**:
    - label2D maps physical keys to logical values
    - label3D supports multilayer keypads (e.g., with shift keys)
    - Uses s32 type to allow special values for non-key presses
3. **Scanning Method**:
    - Implements standard matrix scanning:
        1. Drive one row high
        2. Read all columns
        3. Check for high signal (indicating pressed key)
        4. Repeat for each row
4. **Multilayer Support**:
    - The 4x4x3 declaration suggests support for:
        - 4 rows
        - 4 columns
        - 3 layers (e.g., normal, shift, alt)

---

kpm.c

Header Includes

```
#include <lpc214x.h>  // LPC2148 microcontroller registers
#include "types.h"    // Custom type definitions
#include "gpio.h"     // GPIO control macros
#include "kpm.h"      // Keypad matrix definitions
#include "delay.h"    // Delay functions
```

Key Label Mapping

```
// 4x4 keypad label matrix
s32 label2D[4][4] = {
    {'1', '2', '3', 'A'},  // Row 0: 1, 2, 3, A
    {'4', '5', '6', 'B'},  // Row 1: 4, 5, 6, B
    {'7', '8', '9', 'C'},  // Row 2: 7, 8, 9, C
    {'\b','0','\n','D'}    // Row 3: Backspace, 0, Enter, D
};
```

Initialization Function

```
void initKpm4x4(void) {
    // Configure row pins (P1.24-P1.27) as outputs
    INIT4PINS(GPIO1, KPM_R0, 0xF);

    // Note: Columns are inputs by default (commented initialization)
    // INIT4PINS(GPIO1, KPM_C0, 0x0);
}
```

Key Scanning Function

```
s32 readKpm4x4(void) {
    s16 i, j, stat = 0;

    // Scan each row
    for(i = 0; i < ROWS; i++) {
        // Activate current row (active low)
        WRITE4PINS(GPIO1, KPM_R0, ((1<<i) ^ 0xF));

        // Read column status (default high)
        stat = READ4PINS(GPIO1, KPM_C0);

        // Check if any column is low (key pressed)
        if(stat != 0xF) {
            // Wait for key release (debouncing)
            while(READ4PINS(GPIO1, KPM_C0) != 0xF);

            // Determine which column was pressed
            for(j = 0; j < COLS; j++) {
                if(!((stat >> j) & 1)) {
                    delayMs(300);  // Debounce delay
                    return label2D[i][j];  // Return mapped key value
                }
            }
        }
    }
    return 0;  // No key pressed
}
```

Key Technical Details:

1. **Matrix Scanning**:
    - Uses row-by-row activation (active low)
    - Reads all columns simultaneously
    - Implements full debouncing with:
        - Wait-for-release
        - 300ms delay after release
2. **Electrical Considerations**:
    - Rows are actively driven outputs
    - Columns have pull-up resistors (default high)
    - Active-low logic (0 = pressed)
3. **Key Identification**:

- Combines row and column position
- Returns ASCII/mapped values from label2D
- Special characters:
  - \b = Backspace (0x08)
  - \n = Enter (0x0A)

4. **Performance**:
   - Non-blocking when no key pressed
   - Blocking during key press/release
   - ~300ms debounce period

---

rfid.h

```
/* RFID Header File */
/* Prevents multiple inclusions of this header */
#ifndef _RFID_H
#define _RFID_H

/* Include custom type definitions (s8, s32, etc.) */
#include "types.h"

/* Function Prototypes */

/* Reads an RFID card number into the provided buffer
 * Parameters:
 *   buf - Pointer to a buffer where the card number will be stored
 * Notes:
 *   - The function will block until a complete card number is received
 *   - Buffer must have enough space for the card number + null terminator
 */
void getCardNo(s8 *buf);

/* Validates the format of an RFID card number
 * Parameters:
 *   buf - Pointer to buffer containing the card number to validate
 * Returns:
 *   s32 - 1 if card number is valid, 0 if invalid
 * Validation Rules:
 *   - Checks for proper start/end markers
 *   - Verifies correct length
 *   - May include checksum verification (implementation dependent)
 */
s32 isCardNoOk(s8 *buf);

/* End of header guard */
#endif
```

---

rfid.c

```
/* Include necessary header files */
#include <string.h>      // Standard string operations (strlen, etc.)
#include "types.h"       // Custom type definitions (likely contains s8, u32, etc.)
#include "uart.h"        // UART communication functions (rx_UART, etc.)
```

```c
/* Function: getCardNo
 * Purpose: Reads a card number from UART1 until ETX (0x03) is received
 * Parameter: buf - Pointer to buffer where card number will be stored
 * Note: Assumes UART1 is already initialized before calling this function
 */

void getCardNo(s8 *buf) {
    u32 i = 0;           // Buffer index counter, initialized to 0
    s8 ch;               // Temporary variable to store received character

    do {
        // Read a character from UART1
        ch = rx_UART(U1);

        // If character is not null (0), store it in buffer and increment index
        if (ch) {
            buf[i++] = ch;
        }
    } while (ch != 0x03); // Continue until End of Text (ETX) character is received

    buf[i] = '\0';       // Null-terminate the string (standard C string convention)
}

/* Function: isCardNoOk
 * Purpose: Validates the format of a received card number
 * Parameter: buf - Pointer to buffer containing the card number
 * Returns: 1 if valid, 0 if invalid
 * Validation Rules:
 *   1. Total length must be 10 characters
 *   2. Must start with STX (0x02) and end with ETX (0x03)
 */
s32 isCardNoOk(s8 *buf) {
    // Check if total length is exactly 10 characters
    if (strlen(buf) != 10) {
        return 0;  // Invalid length
    }

    // Check if first character is STX (Start of Text) and last is ETX (End of Text)
    if ((buf[0] != 0x02) || (buf[9] != 0x03)) {
        return 0;  // Invalid start/end markers
    }

    return 1;  // All checks passed, card number is valid
}
```

Key Notes:

1. **Protocol Assumptions**:
    - Card numbers are expected to be in a specific format: [STX][DATA][ETX]
    - STX (Start of Text) = 0x02
    - ETX (End of Text) = 0x03
    - Total length must be exactly 10 characters
2. **Buffer Handling**:
    - getCardNo reads until ETX is received and ensures null-termination
    - isCardNoOk validates the structure of the received data

3. **UART Communication**:
    - Uses rx_UART(U1) to read from UART1
    - Assumes UART1 is properly initialized before these functions are called
4. **Return Values**:
    - isCardNoOk returns 1 (true) for valid cards, 0 (false) for invalid ones

---

rtc.h

```
/* RTC (Real-Time Clock) Header File */
/* Prevents multiple inclusions of this header */
#ifndef _RTC_H_
#define _RTC_H_

/* Includes custom type definitions (u8, u16, s8, etc.) */
#include "types.h"

/* Clock Configuration Defines */
#define FOSC 12000000        // Main oscillator frequency: 12 MHz
#define CCLK (5*FOSC)        // CPU clock: 60 MHz (5 × FOSC using PLL)
#define PCLK (CCLK/4)        // Peripheral clock: 15 MHz (CCLK divided by 4)
#define FRTC 32768           // RTC oscillator frequency: 32.768 kHz (standard RTC crystal)

/* RTC Prescaler Configuration */
#define INT ((s32)(PCLK/FRTC -1))    // Integer part of prescaler
#define FRAC (PCLK-((INT+1)*FRTC))   // Fractional part of prescaler

/* Time Format Options */
#define F_12HRS 1    // 12-hour format flag
#define F_24HRS 0    // 24-hour format flag

/* CCR (Clock Control Register) Bit Definitions */
#define CLKEN  0     // Clock enable bit
#define CTCRST 1     // CTC reset bit
#define CLKSRC 4     // Clock source select bit

/* ILR (Interrupt Location Register) Bit Definitions */
#define RTCCIF 0     // Counter increment interrupt flag
#define RTCALF 1     // Alarm interrupt flag

/* CIIR (Counter Increment Interrupt Register) Bit Definitions */
#define IMSEC  0     // Enable seconds increment interrupt
#define IMMIN  1     // Enable minutes increment interrupt
#define IMHOUR 2     // Enable hours increment interrupt
#define IMDOM  3     // Enable day of month increment interrupt
#define IMDOW  4     // Enable day of week increment interrupt
#define IMDOY  5     // Enable day of year increment interrupt
#define IMMON  6     // Enable month increment interrupt
#define IMYEAR 7     // Enable year increment interrupt

/* AMR (Alarm Mask Register) Bit Definitions */
#define AMRSEC  0    // Alarm seconds mask
#define AMRMIN  1    // Alarm minutes mask
#define AMRHOUR 2    // Alarm hours mask
#define AMRDOM  3    // Alarm day of month mask
#define AMRDOW  4    // Alarm day of week mask
#define AMRDOY  5    // Alarm day of year mask
```

```
#define AMRMON  6    // Alarm month mask
#define AMRYEAR 7    // Alarm year mask

/* External Variables */
extern s8 day[][4];  // External declaration for day names array (e.g., "MON", "TUE")

/* Function Prototypes */

/* Initializes the RTC peripheral */
void init_RTC(void);

/* Sets the RTC date and time */
/* Parameters: year, month, day, hour, minute, second */
void set_RTC(u16 y, u8 m, u8 d, u8 h, u8 min, u8 s);

/* Gets formatted time string */
/* Parameters: buffer pointer, format (12/24 hour) */
void getTime(s8*, u8);  // Output format: "HH:MM:SS AM/PM" or "HH:MM:SS"

/* Gets formatted date string */
/* Parameters: buffer pointer */
void getDate(s8*);      // Output format: "DD/MM/YYYY"

/* Sets an alarm time */
/* Parameters: hour, minute */
void setAlarm(u8 h, u8 m);

/* End of header guard */
#endif
```

Key Features Explained:

1. **Clock Configuration**:
   - The defines establish the clock hierarchy from oscillator to peripheral clocks
   - Shows a common ARM architecture setup with PLL multiplication and clock division
2. **RTC Prescaler**:
   - INT and FRAC are used to configure the RTC prescaler for accurate timekeeping
   - Ensures the 32.768 kHz RTC clock works with the peripheral bus clock
3. **Register Bit Definitions**:
   - Clearly labels all control register bits for easy configuration
   - Follows standard RTC peripheral register organization
4. **Time Formats**:
   - Provides flags for both 12-hour and 24-hour time formats
5. **Functionality**:
   - Complete RTC management (init, set, get time/date)
   - Alarm functionality
   - Supports multiple time/date formats

---

rtc.c

```c
/* Include necessary headers */
#include <lpc214x.h>   // LPC2148 microcontroller specific registers
#include "types.h"     // Custom type definitions
#include "rtc.h"      // RTC header file with declarations

/* Day names array (3-letter abbreviations) */
s8 day[][4] = {"MON","TUE","WED","THR","FRI","SAT","SUN"};

/* Function: init_RTC
 * Purpose: Initialize the Real-Time Clock peripheral
 */
void init_RTC(void) {
    CCR = (1<<CTCRST);    // Reset the RTC counter
    PREINT = INT;         // Set integer part of prescaler
    PREFRAC = FRAC;       // Set fractional part of prescaler
    CCR = (1<<CLKEN);     // Enable the RTC
}

/* Function: set_RTC
 * Purpose: Set the current date and time in RTC
 * Parameters:
 *   y - year
 *   m - month
 *   d - day
 *   h - hour
 *   min - minute
 *   s - second
 */
void set_RTC(u16 y, u8 m, u8 d, u8 h, u8 min, u8 s) {
    // Set date components
    YEAR = y;            // Set year register
    MONTH = m;           // Set month register
    DOM = d;             // Set day of month register
    DOW = 0, DOY = 0;    // Day of week and day of year (not calculated here)

    // Set time components
    HOUR = h;            // Set hour register
    MIN = min;           // Set minute register
    SEC = s;             // Set second register
}

/* Function: getTime
 * Purpose: Get formatted time string
 * Parameters:
 *   str - pointer to output buffer
 *   format - 0 for 24-hour, 1 for 12-hour format
 */
void getTime(s8 *str, u8 format) {
    // Handle hours based on format
    if(format) {  // 12-hour format
        if(HOUR > 12) {
            str[0] = '0' + ((HOUR-12)/10);  // Convert 24-hour to 12-hour format
            str[1] = '0' + ((HOUR-12)%10);
        } else if(!HOUR) {  // Handle midnight (0 hours)
            str[0] = '1', str[1] = '2';
        } else {
            str[0] = '0' + (HOUR/10);
```

```
            str[1] = '0' + (HOUR%10);
        }
    } else {  // 24-hour format
        str[0] = '0' + (HOUR/10);
        str[1] = '0' + (HOUR%10);
    }

    // Add minutes and seconds (same for both formats)
    str[2] = ':';
    str[3] = '0' + (MIN/10);
    str[4] = '0' + (MIN%10);
    str[5] = ':';
    str[6] = '0' + (SEC/10);
    str[7] = '0' + (SEC%10);

    // Add AM/PM for 12-hour format
    if(format) {
        str[8] = ' ';
        str[9] = (HOUR < 12) ? 'A' : 'P';  // AM or PM
        str[10] = 'M';
        str[11] = '\0';  // Null-terminate
    } else {
        str[8] = '\0';   // Null-terminate for 24-hour format
    }
}

/* Function: getDate
 * Purpose: Get formatted date string (DD/MM/YYYY)
 * Parameter: str - pointer to output buffer
 */
void getDate(s8 *str) {
    // Day of month
    str[0] = '0' + (DOM/10);
    str[1] = '0' + (DOM%10);
    str[2] = '/';

    // Month
    str[3] = '0' + (MONTH/10);
    str[4] = '0' + (MONTH%10);
    str[5] = '/';

    // Year (4 digits)
    str[6] = '0' + (YEAR/1000)%10;
    str[7] = '0' + (YEAR/100)%10;
    str[8] = '0' + (YEAR/10)%10;
    str[9] = '0' + (YEAR/1)%10;
    str[10] = '\0';  // Null-terminate
}

/* Function: setAlarm
 * Purpose: Set RTC alarm time
 * Parameters:
 *   h - alarm hour
 *   m - alarm minute
 */
void setAlarm(u8 h, u8 m) {
    ALMIN = m;       // Set alarm minutes
    ALHOUR = h;      // Set alarm hours
    ALSEC = 0;       // Set alarm seconds to 0
```

```
    // Configure alarm mask register (enable hour, minute, second matching)
    AMR = ~((1<<AMRHOUR) | (1<<AMRMIN) | (1<<AMRSEC));

    // Clear any pending alarm interrupt
    ILR = 1<<RTCALF;
}

/* Function prototype for getting complete RTC information */
void getAll_RTC(s8*);  // Not implemented in this file

/* Helper function: int2str
 * Purpose: Convert integer to string (stored in reverse order)
 * Parameters:
 *   str - pointer to output buffer
 *   n - number to convert
 */
void int2str(s8 *str, u32 n) {
    do {
        *(str++) = (n % 10) + '0';  // Convert digit to ASCII
        n /= 10;                     // Move to next digit
    } while(n);                      // Continue until all digits processed
}
```

Key Implementation Details:

1. **RTC Initialization**:
   - First resets the counter, then configures the prescaler for accurate timekeeping
   - Finally enables the RTC with (1<<CLKEN)
2. **Time Formatting**:
   - Handles both 12-hour and 24-hour formats
   - Properly converts hours for AM/PM display
   - Always shows leading zeros for single-digit values
3. **Date Formatting**:
   - Formats date as DD/MM/YYYY with leading zeros
   - Handles all four digits of the year
4. **Alarm Configuration**:
   - Sets specific alarm time (hour and minute)
   - Configures which time components should trigger the alarm
   - Clears any pending alarm interrupt
5. **Helper Functions**:
   - int2str converts numbers to ASCII strings (note: stores digits in reverse order)
6. **Register Usage**:
   - Uses LPC214x hardware registers directly (YEAR, MONTH, DOM, etc.)
   - Follows the LPC2148 RTC peripheral specification

---

uart.h

```
/* UART (Universal Asynchronous Receiver/Transmitter) Header File */
/* Prevents multiple inclusions of this header */
#ifndef _UART_H_
#define _UART_H_

/* Include custom type definitions (u8, s8, etc.) */
#include "types.h"

/* System Clock Configuration */
#define FOSC 12000000        // Main oscillator frequency: 12 MHz
#define CCLK (FOSC*5)        // CPU clock: 60 MHz (5 × FOSC using PLL)
#define PCLK (CCLK/4)        // Peripheral clock: 15 MHz (CCLK divided by 4)

/* UART Baud Rate Configuration */
#define BAUD 9600            // Desired baud rate: 9600 bits per second
#define DVSR (PCLK/(16*BAUD)) // Divisor value for baud rate generation

/* UART Channel Identifiers */
#define U0 0                 // UART0 identifier
#define U1 1                 // UART1 identifier

/* UART Pin Definitions (for LPC214x) */
#define U0TX 0               // UART0 TXD pin number (P0.0)
#define U0RX 1               // UART0 RXD pin number (P0.1)
#define U1TX 8               // UART1 TXD pin number (P0.8)
#define U1RX 9               // UART1 RXD pin number (P0.9)

/* Special Function Register (SFR) Bit Definitions */

/* UxLCR (Line Control Register) Bits */
#define UART_DLAB  7         // Divisor Latch Access Bit (DLAB)
#define UART_WRDLEN 3        // Word Length Select bits (set for 8-bit data)

/* UxLSR (Line Status Register) Bits */
#define UART_TEMT 6          // Transmitter Empty indicator
#define UART_DR   0          // Data Ready indicator (received data available)

/* Function Prototypes */

/* Initializes UART peripheral(s) */
void initUART(void);

/* Transmits a byte on specified UART channel */
/* Parameters: UART channel number (U0/U1), data byte */
void txUART(u8, u8);

/* Receives a byte from specified UART channel */
/* Parameters: UART channel number (U0/U1) */
/* Returns: received data byte */
u8 rxUART(u8);

/* Alternative receive function (same as rxUART) */
u8 rx_UART(u8 un);

/* Transmits a single character on specified UART channel */
/* Parameters: UART channel number (U0/U1), character to send */
void charTxUART(u8, s8);
```

```
/* Transmits a null-terminated string on specified UART channel */
/* Parameters: UART channel number (U0/U1), string pointer */
void strTxUART(u8, s8*);

/* Receives a single character from specified UART channel */
/* Parameters: UART channel number (U0/U1) */
/* Returns: received character */
s8 charRxUART(u8);

/* Receives a string until newline on specified UART channel */
/* Parameters: UART channel number (U0/U1), buffer pointer */
void strRxUART(u8, s8*);

/* Alternative string receive function */
/* Parameters: UART channel number (U0/U1), buffer pointer */
void strRx_UART(u8 un, s8* buf);

/* End of header guard */
#endif
```

Key Components Explained:

1. **Clock Configuration**:
   - Defines the clock hierarchy from oscillator (FOSC) to peripheral clock (PCLK)
   - Shows typical PLL configuration for LPC214x (5× multiplier)
2. **Baud Rate Calculation**:
   - DVSR is calculated using the standard UART baud rate formula:
     PCLK / (16 × desired baud rate)
3. **Hardware Mapping**:
   - Clearly defines which physical pins correspond to which UART channels
   - Follows LPC214x default UART pin assignments
4. **Register Bit Definitions**:
   - Documents critical bits in UART registers:
     - UART_DLAB for accessing baud rate divisors
     - UART_WRDLEN for 8-bit data format
     - UART_TEMT and UART_DR for status checking
5. **Function Interface**:
   - Provides complete UART functionality:
     - Initialization (initUART)
     - Byte-level I/O (txUART, rxUART)
     - Character I/O (charTxUART, charRxUART)
     - String I/O (strTxUART, strRxUART)
   - Includes alternative functions (rx_UART, strRx_UART) for compatibility
6. **Design Notes**:
   - Supports both UART0 and UART1 through channel parameter
   - Uses standard baud rate (9600) as default
   - Follows common embedded practice of direct register bit manipulation

uart.c

```
#include <lpc214x.h>   // LPC2148 microcontroller register definitions
#include "types.h"     // Custom data type definitions
#include "uart.h"      // UART function declarations
#include "atm.h"       // ATM application specific definitions

/* Initialize UART0 and UART1 peripherals */
void initUART(void) {
    // Configure UART1 pins (P0.8 as TXD1, P0.9 as RXD1)
    PINSEL0 = ((PINSEL0 & ~(0xF << U1TX*2)) | (5 << U1TX*2);

    // UART1 Line Control Register setup:
    U1LCR = ((1 << UART_DLAB) | (UART_WRDLEN));  // Enable DLAB, 8-bit word length

    // Set UART1 baud rate divisors
    U1DLL = DVSR;       // Lower byte of divisor
    U1DLM = DVSR >> 8;  // Upper byte of divisor

    // Lock baud rate registers
    U1LCR &= ~(1 << UART_DLAB);

    // Configure UART0 pins (P0.0 as TXD0, P0.1 as RXD0)
    PINSEL0 = ((PINSEL0 & ~0xF) | 5);

    // UART0 Line Control Register setup (same as UART1)
    U0LCR = ((1 << UART_DLAB) | (UART_WRDLEN));

    // Set UART0 baud rate divisors
    U0DLL = DVSR;       // Lower byte of divisor
    U0DLM = DVSR >> 8;  // Upper byte of divisor

    // Lock baud rate registers
    U0LCR &= ~(1 << UART_DLAB);

#ifdef UART_INT
    // UART0 interrupt configuration (if enabled)
    VICIntSelect = 0;                     // Set all interrupts as IRQ
    VICVectAddr0 = (unsigned)UART0_isr;   // Set interrupt handler address
    VICVectCntl0 = 0x20 | 6;              // Assign UART0 to vector 0 (priority)
    VICIntEnable = 1 << 6;                // Enable UART0 interrupt
    U0IER = 0x01;                         // Enable Receive Data Available interrupt
#endif
}

/* Transmit a byte on specified UART */
void txUART(u8 un, u8 word) {
    if(un) {  // UART1
        U1THR = word;  // Write to transmit holding register
        while(((U1LSR >> UART_TEMT) & 1) == 0);  // Wait until transmitter empty
    } else {   // UART0
        U0THR = word;
        while(((U0LSR >> UART_TEMT) & 1) == 0);
    }
}

/* Receive a byte from specified UART */
```

```
u8 rx_UART(u8 un) {
    static u8 word;  // Stores received character
    word = 0;        // Default return value

    if(un) {  // UART1
        if((U1LSR >> UART_DR) & 1)  // Check if data ready
            word = U1RBR;           // Read from receive buffer
    } else {   // UART0
        if((U0LSR >> UART_DR) & 1)
            word = U0RBR;
    }
    return word;
}

/* Receive a string until newline character */
void strRx_UART(u8 un, s8* buf) {
    u32 i = 0;
    s8 ch;

    do {
        ch = rx_UART(U0);  // Always uses UART0 (parameter unused)
        if(ch) buf[i++] = ch;  // Store if not null
    } while(ch != '\n');       // Continue until newline

    buf[i-1] = '\0';  // Replace newline with null terminator
}

/* Transmit a null-terminated string */
void strTxUART(u8 un, s8 *str) {
    while(*str)  // Loop until null terminator
        txUART(un, *(str++));  // Transmit each character
}
```

Key Implementation Details:

1. **Initialization**:
    - Configures both UART0 and UART1 with identical settings
    - Sets baud rate using pre-calculated DVSR value
    - Uses 8-bit word length, 1 stop bit, no parity (default)
    - Optional interrupt configuration for UART0
2. **Core Functions**:
    - txUART(): Blocking transmit with hardware register wait
    - rx_UART(): Non-blocking receive (returns 0 if no data)
    - strTxUART(): String transmitter using txUART()
    - strRx_UART(): Line-based receiver (note: always uses UART0)
3. **Hardware Interaction**:
    - Direct register access (UxTHR, UxRBR, UxLSR)
    - Proper baud rate divisor handling (DLL/DLM registers)
    - Pin function selection via PINSEL0
4. **Design Notes**:
    - Mixed blocking (tx) and non-blocking (rx) approaches

- String functions work with null-terminated C strings
- Interrupt support is conditionally compiled
- UART1 implementation is incomplete (missing some functionality)
- strRx_UART() ignores the channel parameter (always uses UART0)

5. **Potential Improvements**:
   - Consistent channel parameter usage
   - Timeout mechanisms for receive functions
   - Error handling for invalid baud rates
   - More complete UART1 support
   - Buffer overflow protection in string functions

---

atm.h

```
/* ATM Controller Header File */
#ifndef _ATM_H       // Header guard to prevent multiple inclusions
#define _ATM_H

/* Standard C Library Headers */
#include <lpc214x.h>   // LPC2148 microcontroller registers
#include <string.h>    // String manipulation functions
#include <stdio.h>     // Standard I/O functions
#include <stdlib.h>    // Standard library functions

/* Custom Project Headers */
#include "types.h"     // Custom type definitions (u8, s8, u32, etc.)
#include "kpm.h"       // Keypad matrix module
#include "lcd.h"       // LCD display module
#include "uart.h"      // UART communication module
#include "delay.h"     // Delay functions
#include "rfid.h"      // RFID card reader module
#include "atm.h"       // Self-inclusion (typically not needed)

/* Configuration Defines */
#define UART_INT       // Enable UART interrupts (comment to disable)
#define CC (const char*)  // Convenience macro for string literals

/* ATM Transaction Limits */
#define MAX_TRYS       3       // Maximum PIN attempts allowed
#define MAX_DEPOSIT    30000   // Maximum single deposit amount (₹30,000)
#define MAX_WITHDRAW   30000   // Maximum single withdrawal amount (₹30,000)
#define MAX_TRANSFER   100000  // Maximum transfer amount (₹100,000)

/* String/Keypad Matrix Definitions */
#define BUF_MAX       50      // Maximum buffer size for strings
#define PIN           1       // Flag for PIN entry mode
#define STR           0       // Flag for string entry mode

/* Timing Constants */
#define DISP_TIME     1000    // Display time in ms (1 second)
#define ATM_TIME      (30*120000) // ATM timeout period (30 seconds)

/* Global Variables */
extern volatile s8 buf[], dummy;  // Communication buffers
extern volatile u32 r_flag, time, buf_index;  // Status flags and counters
```

```
/* Function Prototypes */

/* System Initialization */
void sys_init(void);  // Initializes all ATM hardware components

/* UART Interrupt Handler */
void UART0_isr(void) __irq;  // __irq indicates ARM interrupt service routine

/* PC Communication Functions */
void checkPC(void);          // Checks for messages from PC
void sendMsg(s8 *str);       // Sends message to PC
void getMsg(s8 *str);        // Receives message from PC
s32 isMsgOk(s8 *str);        // Validates received message format
u32 getPin(s8 *str);         // Extracts PIN from message

/* Keypad Matrix Input */
int str_KPM(s8 *str, u8 row);  // Gets string/PIN input via keypad

/* ATM Transaction Functions */
void atm_wtd(s8 *rfid, s8 *buf);  // Withdrawal transaction
void atm_dep(s8 *rfid, s8 *buf);  // Deposit transaction
void atm_bal(s8 *rfid, s8 *buf);  // Balance inquiry
void atm_mst(s8 *rfid, s8 *buf);  // Master menu (admin functions)
int atm_pin(s8 *rfid, s8 *pin, s8 *buf);  // PIN verification

#endif  // End of header guard
```

Key Components Explained:

1. **System Architecture**:
   - Integrates all hardware modules (LCD, UART, RFID, Keypad)
   - Defines transaction limits and timing constants
   - Uses both interrupt-driven and polling-based approaches
2. **PC Communication**:
   - Implements message-based protocol with validation
   - Uses UART interrupts for efficient communication
   - Provides functions for both sending and receiving messages
3. **Security Features**:
   - PIN entry with attempt limiting (MAX_TRYS)
   - RFID card authentication
   - Timeout protection (ATM_TIME)
4. **Transaction Processing**:
   - Supports all standard ATM operations:
     - Withdrawals (with amount limit)
     - Deposits
     - Balance inquiries
     - Administrative functions
   - Amount limits prevent fraudulent transactions
5. **User Interface**:

- Combines LCD output with keypad input
- Specialized input function (str_KPM) handles both strings and PINs
- Visual feedback timing (DISP_TIME)

6. **Design Notes**:
   - Uses volatile variables for interrupt-shared data
   - Implements a state machine architecture
   - Follows embedded systems best practices for resource-constrained environments
   - Provides admin/master functions for system maintenance

---

atm.c

Header Includes and Definitions

```
#include <string.h>   // Standard string functions
#include "atm.h"      // Main ATM header with all declarations

#define ISNUM(ch) ((ch>='0')&&(ch<='9'))  // Macro to check if character is a digit

// Global buffers and flags
volatile s8 buf[BUF_MAX]="";  // Communication buffer (50 bytes)
volatile s8 dummy;            // Dummy variable for interrupt clearing
volatile u32 r_flag;          // Received data flag (interrupt signaling)
volatile u32 time;            // Timeout counter
volatile u32 buf_index;       // Buffer index for UART reception
```

System Initialization

```
void sys_init(void){
    initUART();    // Initialize UART0 and UART1
    initLcd();     // Initialize LCD display
    initKpm4x4();  // Initialize 4x4 keypad matrix
                   // Buzzer initialization would go here
}
```

UART Interrupt Handler

```
void UART0_isr(void) __irq {
    if ((U0IIR & 0x0E) == 0x04) {  // Check for Receive Data Available interrupt
        char ch = U0RBR;  // Read received byte

        if (!r_flag){  // Only process if not already processing a message
            if(ch=='\n'){  // End of message
                buf[buf_index-1] = '\0';  // Replace \r with null terminator

                if(!strcmp((const char*)buf,"@Y:LINEOK$")){  // Line test message
                    sendMsg("#Y:LINEOK$");  // Respond with acknowledgment
                } else {
                    r_flag=1;  // Set flag for main program to process message
                }
                buf_index=0;  // Reset buffer index
            }
            else if(buf_index < BUF_MAX-1) {  // Buffer not full
                buf[buf_index++]=ch;  // Store character
            }
        }
```

```
    }

    dummy = U0IIR;  // Clear interrupt (read IIR)
    VICVectAddr = 0;  // Acknowledge interrupt in vector controller
}
```

Communication Functions

```
void sendMsg(s8 *str){
    strTxUART(U0,str);  // Send string
    strTxUART(U0,"\r\n");  // Send terminator

    #ifdef DBG  // Debug output to LCD
    moveLcdCursor(1,0);
    str2Lcd("Tx:");
    str2Lcd(str);
    str2Lcd(".");
    delayS(1);
    #endif
}

void getMsg(s8 *str){
    strRx_UART(U0,str);  // Receive string (includes \r)
    str[strlen(str)-1]='\0';  // Remove \r

    #ifdef DBG  // Debug output
    moveLcdCursor(1,0);
    str2Lcd("Rx:");
    str2Lcd(str);
    str2Lcd(".");
    delayS(1);
    #endif
}

s32 isMsgOk(s8 *str){
    if((str[0]!='@')||(str[strlen(str)-1]!='$')) return 0;  // Check message framing
    return 1;
}
```

PIN Entry Function

```
u32 getPin(s8 *str){
    s32 i=0;
    u32 time=ATM_TIME;  // 30 second timeout
    s8 ch;

    clearLcdRow(1);  // Clear LCD bottom row

    while(i<4 && time){  // Need 4 digits or timeout
        while(!(ch=readKpm4x4())){  // Wait for keypress
            if(time) --time;
            else return 0;  // Timed out
        }

        if(ISNUM(ch)){  // Numeric key
            char2Lcd(ch);  // Show digit briefly
            delayMs(300);
            moveLcdCursor(1,i);
            char2Lcd('*');  // Replace with asterisk
```

```
            str[i++]=ch;  // Store digit
            time=ATM_TIME;  // Reset timeout
        }
        else if(ch=='\b'){  // Backspace
            if(i){
                --i;
                moveLcdCursor(1,i);
                char2Lcd(' ');
                moveLcdCursor(1,i);
            }
            time=ATM_TIME;
        }
        else if(ch=='C'){  // Cancel
            return -1;
        }
    }
    str[i]='\0';  // Null terminate

    return time?1:0;  // 1=success, 0=timeout
}
```

String Input Function

```
int str_KPM(s8 *str,u8 row){
    u32 i=0;
    s8 ch='\0';
    moveLcdCursor(row,0);

    while(i<BUF_MAX-1 && time){  // Buffer not full and not timed out
        ch=readKpm4x4();
        if(ch) time=ATM_TIME;  // Reset timeout on keypress
        else {
            time--;
            continue;
        }

        if(ch=='\b'){  // Backspace
            str[i]='\0';
            moveLcdCursor(row,i);
            char2Lcd(' ');
            moveLcdCursor(row,i);
            if(i) --i;
        }
        else if(i && ch=='\n'){  // Enter (only allowed after some input)
            str[i]='\0';
            return 0;  // Valid string
        }
        else if(ISNUM(ch)){  // Numeric input
            if(!i && (ch=='0')) continue;  // No leading zeros
            str[i]=ch;
            char2Lcd(ch);
            i++;
        }
        else if(ch=='C'){  // Cancel
            return -1;
        }
    }
    return 1;  // Timeout
}
```

ATM Transaction Functions

Withdrawal

```
void atm_wtd(s8 *rfid,s8 *buf){
    s8 amt[20];
    clearLcdDisplay();
    str2Lcd("Withdraw amt:");

    // Get amount input
    if(str_KPM(amt,1)==-1){  // Cancelled
        moveLcdCursor(0,0);
        str2Lcd(" Cancelled !!!! ");
        clearLcdRow(1);
        delayS(2);
        return;
    }

    // Validate amount (must be multiple of 100)
    if((amt[strlen(amt)-2]!='0')||(amt[strlen(amt)-1]!='0')){
        moveLcdCursor(0,0);
        str2Lcd(" Amount Must be ");
        moveLcdCursor(1,0);
        str2Lcd("100Rs multiples!");
        delayS(2);
        return;
    }

    // Send withdrawal request
    checkPC();  // Verify PC connection
    sprintf(buf,"#%c:WTD:%s:%s$",'A',rfid,amt);
    sendMsg(buf);

    #ifdef UART_INT
    while(!r_flag);  // Wait for response (interrupt)
    r_flag=0;
    #else
    getMsg(buf);  // Poll for response
    #endif

    // Handle response
    if(!strcmp(buf,"@OK:DONE$")){
        moveLcdCursor(0,0);
        str2Lcd("Amount Withdrawn");
        moveLcdCursor(1,0);
        str2Lcd("  Succesfully!! ");
    }
    else if(!strcmp(buf,"@ERR:LOWBAL$")){
        moveLcdCursor(0,0);
        str2Lcd("Withdraw Failed ");
        moveLcdCursor(1,0);
        str2Lcd("  Low-Balance!  ");
    }
    else if(!strcmp(buf,"@ERR:MAXAMT$")){
        moveLcdCursor(0,0);
        str2Lcd("Withdraw Failed ");
        moveLcdCursor(1,0);
```

```
        str2Lcd("Exceeds MaxLimit");
    }
    delayS(2);
}
```

Deposit (similar structure to withdrawal)

```
void atm_dep(s8 *rfid,s8 *buf){
    // Similar flow to withdrawal but without 100Rs multiple check
    // Sends #A:DEP:<rfid>:<amt>$ message
    // Handles @OK:DONE$, @ERR:NEGAMT$, @ERR:MAXAMT$ responses
}
```

Balance Inquiry

```
void atm_bal(s8 *rfid,s8 *buf){
    clearLcdDisplay();
    str2Lcd("Balance:");

    checkPC();
    while(1){
        sprintf(buf,"#%c:BAL:%s$",'A',rfid);
        sendMsg(buf);

        #ifdef UART_INT
        while(!r_flag);
        r_flag=0;
        #else
        getMsg(buf);
        #endif

        if(isMsgOk(buf)) break;  // Valid message format
    }

    // Extract balance from @OK:BAL=<amt>$ response
    if(!strncmp(buf,"@OK:BAL=",7)){
        s32 i=0;
        while(1){
            if(buf[i+8]=='$'){
                buf[i]='\0';
                break;
            }
            buf[i]=buf[i+8];  // Shift balance to start of buffer
            i++;
        }
        clearLcdRow(1);
        str2Lcd(buf);
        str2Lcd(" Rs");
    }
    delayS(2);
}
```

PIN Change Function

```
int atm_pin(s8 *rfid,s8 *pin,s8 *buf){
    s8 dum[5]={0};
    s32 ret;

    // Old PIN verification
```

```
    clearLcdDisplay();
    str2Lcd("Enter Old pin:");
    ret=getPin(buf);
    if(!ret) return 0;  // Timeout
    if(ret==-1) return 0;  // Cancelled
    if(strcmp(pin,buf)){  // Verify old PIN
        clearLcdDisplay();
        str2Lcd("Incorrect pin!!");
        return 0;
    }

    // New PIN entry
    clearLcdDisplay();
    str2Lcd("Enter New pin:");
    ret=getPin(buf);
    if(!ret) return 0;  // Timeout
    if(ret==-1) return 0;  // Cancelled

    // New PIN confirmation
    clearLcdDisplay();
    str2Lcd("Re-enter New pin");
    ret=getPin(dum);
    if(!ret) return 0;  // Timeout
    if(ret==-1) return 0;  // Cancelled

    if(strcmp(buf,dum)){  // Verify match
        clearLcdDisplay();
        str2Lcd("New pin mismatch");
        delayS(1);
        return 0;
    }

    strcpy(pin,dum);  // Update PIN

    // Send PIN change to server
    checkPC();
    while(1){
        sprintf(buf,"#%c:PIN:%s:%s$",'A',rfid,pin);
        sendMsg(buf);

        #ifdef UART_INT
        while(!r_flag);
        r_flag=0;
        #else
        getMsg(buf);
        #endif

        if(isMsgOk(buf)) break;
    }

    if(!strcmp(buf,"@OK:DONE$")) return 1;  // Success
    else {
        clearLcdDisplay();
        str2Lcd("Unknown Error");
        return 0;
    }
}
```

Master Menu (Transaction History)

```
void atm_mst(s8 *rfid,s8 *buf){
    s32 i=0,j=0;
    s8 t[][4]={"WTD","DEP","TIN","TOT"};  // Transaction type labels

    checkPC();
    while(j<3){  // Get up to 3 transactions
        sprintf(buf,"#%c:MST:%s:%d$",'A',rfid,j+1);
        sendMsg(buf);

        #ifdef UART_INT
        while(!r_flag);
        r_flag=0;
        #else
        getMsg(buf);
        #endif

        if(isMsgOk(buf)){
            if(buf[5]=='7') break;  // No more transactions

            // Display transaction details
            moveLcdCursor(0,0);
            for(i=0;i<16;i++){  // Date/time portion
                char2Lcd(buf[i+7]);
            }
            clearLcdRow(1);
            moveLcdCursor(1,0);
            for(i=0;buf[i+24]!='$';i++){  // Amount portion
                char2Lcd(buf[i+24]);
            }
            moveLcdCursor(1,13);
            str2Lcd(t[buf[5]-'1']);  // Transaction type
            delayS(2);
            ++j;
        }
    }
}
```

PC Connection Check

```
void checkPC(void){
    while(1){
        sendMsg("#X:LINEOK$");  // Send line test

        #ifdef UART_INT
        while(!r_flag);
        r_flag=0;
        #else
        getMsg(buf);
        #endif

        if(!strcmp((const char*)buf,"@X:LINEOK$")){
            break;  // PC responded OK
        }
        else{
            moveLcdCursor(1,0);
```

```
            str2Lcd("waiting for PC.");
        }
        delayS(2);
    }
}
```

Key Features Summary:

1. **Robust Communication Protocol**:
    - Uses framed messages with '@' prefix and '$' suffix
    - Implements message validation (isMsgOk)
    - Supports both interrupt and polling modes
2. **Security Features**:
    - PIN entry with masking (asterisks)
    - Timeout protection for all user inputs
    - RFID card authentication
3. **Transaction Processing**:
    - Handles withdrawals, deposits, balance inquiries
    - Validates amounts (e.g., multiples of 100 for withdrawals)
    - Provides clear user feedback via LCD
4. **Error Handling**:
    - Comprehensive response code handling
    - Timeout management
    - User cancellation support
5. **Modular Design**:
    - Separates UI, communication, and business logic
    - Reusable input functions (getPin, str_KPM)
    - Clear state management via flags

---

atmMain.c

Initialization Section

```
#include "atm.h"  // Main header containing all dependencies and prototypes

// Menu items displayed on 16x2 LCD (6 options)
const s8 menu[6][16] = {
    "1.WITHDRAW CASH",  // Index 0
    "2.DEPOSIT CASH",   // Index 1
    "3.VEIW BALANCE",   // Index 2 (Note: Typo in "VIEW")
    "4.MINI STATMENT",  // Index 3 (Note: Typo in "STATEMENT")
    "5.PIN CHANGE",     // Index 4
    "6.EXIT ATM"        // Index 5
};

s32 main() {
    // User authentication variables
    s8 rfid[9]="11111111";  // Default RFID (8 digits + null terminator)
    s8 pin[5]="1111";       // Default PIN (4 digits + null terminator)
```

```
    s8 ch;                  // Keypad input character

    // System state variables
    s32 trys;               // Remaining PIN attempts
    s32 curRow=0;           // Current menu row (0-4)
    s32 prevRow=-1;         // Previous menu row (for refresh detection)
    s32 errNo;              // Error/status code

    // Hardware initialization
    sys_init();  // Initialize UART, LCD, Keypad

    // Welcome message
    moveLcdCursor(0,0);         // Move to first line
    str2Lcd(" Welcome To ATM "); // Display welcome
    sendMsg(" Welcome To ATM "); // Also send to backend
    delayS(1);                  // Show for 1 second
```

Main Card Detection Loop

```
    while(1) {  // Infinite service loop
        // Card insertion prompt
        moveLcdCursor(0,0);
        str2Lcd("Place the card  ");  // Line 1
        moveLcdCursor(1,0);
        str2Lcd("on the reader.. ");  // Line 2

        // Read RFID card
        getCardNo(buf);  // Reads into global buffer

        // Validate card format
        if(!isCardNoOk(buf)) {  // Checks length, start/end markers
            delayS(1);  // Brief delay before retry
            continue;   // Skip to next iteration
        }

        // Extract 8-digit RFID (skip start byte)
        strncpy(rfid, (const char*)buf+1, 8);
        rfid[8] = '\0';  // Ensure null-termination

        // Display RFID temporarily
        clearLcdDisplay();
        str2Lcd("RFID:");
        str2Lcd(rfid);
```

Backend Card Verification

```
        // Verify card with backend
        checkPC();  // Ensure connection to server

        // Send card verification request
        while(1) {
            sprintf(buf,"#%c:%s$",'C',rfid);  // Format: #C:RFID$
            sendMsg(buf);

            // Wait for response (interrupt or polling)
            #ifdef UART_INT
            while(!r_flag);  // Wait for interrupt flag
            r_flag=0;        // Clear flag
            #else
```

```
            getMsg(buf);     // Poll for response
            #endif

            if(isMsgOk(buf)) break;  // Valid message format
        }

        // Process response
        if(!strncmp(buf,"@OK:ACTIVE:",11)) {
            // Valid card - display welcome with name
            moveLcdCursor(0,0);
            str2Lcd("Welcome customer");
            moveLcdCursor(1,0);
            // Extract and display name (between : and $)
            for(time=0; buf[time+11]!='$' && time<16; time++) {
                char2Lcd(buf[time+11]);
            }
            // Clear remaining spaces
            for(;time<16;time++) char2Lcd(' ');
            delayS(2);
        }
        else if(!strcmp(buf,"@ERR:BLOCK$")) {
            // Blocked card
            moveLcdCursor(0,0);
            str2Lcd("Card is Blocked!");
            moveLcdCursor(1,0);
            str2Lcd("   Visit Bank.  ");
            delayS(2);
            continue;  // Restart card loop
        }
        else if(!strcmp(buf,"@ERR:INVALID$")) {
            // Unknown card
            moveLcdCursor(0,0);
            str2Lcd(" Card not Found!");
            moveLcdCursor(1,0);
            str2Lcd("Register at bank");
            delayS(2);
            continue;
        }
```

PIN Entry and Verification

```
        // PIN entry with MAX_TRYS attempts
        trys = MAX_TRYS;  // Reset attempts (typically 3)

        while(trys) {
            // Show remaining attempts after first failure
            if(trys != MAX_TRYS) {
                moveLcdCursor(0,0);
                str2Lcd("  Wrong Pin!!!  ");
                moveLcdCursor(1,0);
                char2Lcd('0'+trys);  // Show remaining tries
                str2Lcd(" - Tries Left  ");
                delayS(2);
            }

            // PIN entry prompt
            moveLcdCursor(0,0);
            str2Lcd("Enter pin:      ");
            clearLcdRow(1);
```

```
        // Get PIN input (masked with asterisks)
        errNo = getPin(pin);

        // Handle input result
        if(!errNo) {  // Timeout
            moveLcdCursor(0,0);
            str2Lcd("Session Time-Out");
            moveLcdCursor(1,0);
            str2Lcd("   Thank you.   ");
            delayS(2);
            break;
        }
        else if(errNo==-1) {  // Cancelled
            moveLcdCursor(0,0);
            str2Lcd("Session Canceled");
            moveLcdCursor(1,0);
            str2Lcd("   Thank you.   ");
            delayS(2);
            break;
        }

        // Verify PIN with backend
        checkPC();  // Ensure connection

        // Send verification request
        while(1) {
            sprintf(buf,"#%c:%s:%s$",'V',rfid,pin);  // #V:RFID:PIN$
            sendMsg(buf);

            // Wait for response
            #ifdef UART_INT
            while(!r_flag);
            r_flag=0;
            #else
            getMsg(buf);
            #endif

            if(isMsgOk(buf)) break;
        }

        // Check response
        if(!strcmp(buf,"@OK:MATCHED$")) {
            break;  // Correct PIN - exit retry loop
        }
        else if(!strcmp(buf,"@ERR:WRONG$")) {
            trys--;  // Decrement attempts
        }
    }

    // Handle PIN attempts exhaustion
    if(!trys) {
        // Block card in backend
        while(1) {
            sprintf(buf,"#%c:BLK:%s$",'A',rfid);  // #A:BLK:RFID$
            sendMsg(buf);

            // Wait for acknowledgment
            #ifdef UART_INT
```

```
            while(!r_flag);
            r_flag=0;
            #else
            getMsg(buf);
            #endif

            if(isMsgOk(buf) && !strcmp(buf,"@OK:DONE$")) break;
        }

        // Show blocked message
        clearLcdDisplay();
        str2Lcd(" Card Blocked!! ");
        moveLcdCursor(1,0);
        str2Lcd("   Visit Bank.  ");
        delayS(2);
        continue;  // Restart card loop
    }
```

ATM Main Menu System

```
    // Menu navigation state
    prevRow=-1;  // Force initial refresh
    curRow=0;    // Start at first menu item
    time=ATM_TIME;  // Reset inactivity timer

    while(time) {  // Menu loop with timeout
        // Refresh menu when row changes
        if(curRow != prevRow) {
            clearLcdDisplay();
            str2Lcd(menu[curRow]);    // Current option
            moveLcdCursor(1,0);
            str2Lcd(menu[curRow+1]);  // Next option
            prevRow = curRow;
        }

        // Check keypad with timeout
        while(!(ch=readKpm4x4())) {
            if(time) time--;  // Decrement timeout
            else break;       // Exit on timeout
        }

        if(ch) {  // Key pressed
            // Navigation handling
            if(ch=='B') {  // Down arrow
                if(curRow < 4) curRow++;  // Scroll down
            }
            else if(ch=='A') {  // Up arrow
                if(curRow > 0) curRow--;  // Scroll up
            }
            // Option selection
            else if(ch==('0'+curRow+1) || ch==('0'+curRow+2)) {
                switch(ch) {
                    case '1':  // Withdraw
                        atm_wtd(rfid,buf);
                        prevRow=-1;  // Force menu refresh
                        break;

                    case '2':  // Deposit
                        atm_dep(rfid,buf);
```

```
                        prevRow=-1;
                        break;

                    case '3':  // Balance
                        atm_bal(rfid,buf);
                        delayS(3);  // Show balance longer
                        prevRow=-1;
                        break;

                    case '4':  // Mini-statement
                        atm_mst(rfid,buf);
                        prevRow=-1;
                        break;

                    case '5':  // PIN Change
                        if(trys) {  // Only if attempts remain
                            if(atm_pin(rfid,pin,buf)) {
                                // Successful change
                                clearLcdDisplay();
                                str2Lcd("Pin changed.");
                                trys=MAX_TRYS;  // Reset attempts
                            }
                            else {
                                // Failed change
                                moveLcdCursor(1,0);
                                str2Lcd("Trys left: ");
                                char2Lcd(--trys+'0');
                            }
                            delayMs(DISP_TIME);

                            if(!trys) {  // Block if attempts exhausted
                                // ... (blocking code as before)
                                goto exit_menu;  // Exit menu loop
                            }
                        }
                        prevRow=-1;
                        break;

                    case '6':  // Exit
                        checkPC();
                        sendMsg("#Q:SAVE$");  // Notify backend
                        goto exit_menu;  // Exit menu loop
                }
            }

            time = ATM_TIME;  // Reset inactivity timer
        }

        if(time) time--;  // Decrement timeout
    }
```

Session Termination

```
exit_menu:
    // Session end messages
    if(time) {  // User-initiated exit
        moveLcdCursor(0,0);
        str2Lcd("  Thank You !!  ");
        moveLcdCursor(1,0);
```

```
            str2Lcd("Have a nice Day.");
        }
        else {  // Timeout
            moveLcdCursor(0,0);
            str2Lcd("Session Time-Out");
            moveLcdCursor(1,0);
            str2Lcd("   Thank you.   ");
        }
        delayS(2);  // Show message before restarting
    }
}
```

Key Technical Aspects:

1. **State Management**:
    - curRow/prevRow track menu navigation
    - trys manages PIN attempt counter
    - time handles session timeout
2. **Communication Protocol**:
    - Uses framed messages (# prefix, $ suffix)
    - Message types:
        - #C:RFID$ - Card verification
        - #V:RFID:PIN$ - PIN verification
        - #A:BLK:RFID$ - Block card
        - #Q:SAVE$ - Session end
3. **Error Handling**:
    - Card blocked/invalid cases
    - PIN attempt tracking
    - Timeout detection
4. **User Interface**:
    - 16x2 LCD menu system
    - Keypad navigation (up/down/select)
    - Visual feedback for all operations
5. **Security Features**:
    - Masked PIN entry
    - Automatic card blocking
    - Session timeout

ATM Communication Protocol: MCU ↔ PC Commands

Sr. No.	Transaction Type	Direction	Command Format	Response Format	Description
1	**Card Verification**	MCU → PC	#C:<RFID>$	@OK:ACTIVE:<Name>$	Valid card with account name
				@ERR:BLOCK$	Card is blocked
				@ERR:INVALID$	Card not registered
2	**PIN Verification**	MCU → PC	#V:<RFID>:<PIN>$	@OK:MATCHED$	Correct PIN
				@ERR:WRONG$	Incorrect PIN
3	**Withdrawal**	MCU → PC	#A:WTD:<RFID>:<Amount>$	@OK:DONE$	Withdrawal successful
				@ERR:LOWBAL$	Insufficient balance
				@ERR:NEGAMT$	Negative amount invalid
				@ERR:MAXAMT$	Exceeds ₹30,000 limit
4	**Deposit**	MCU → PC	#A:DEP:<RFID>:<Amount>$	@OK:DONE$	Deposit successful
				@ERR:NEGAMT$	Negative amount invalid
				@ERR:MAXAMT$	Exceeds ₹30,000 limit
5	**Balance Inquiry**	MCU → PC	#A:BAL:<RFID>$	@OK:BAL=<Amount>$	Returns current balance (e.g., @OK:BAL=25000$)
6	**Mini-Statement**	MCU → PC	#A:MST:<RFID>:<TxNo>$	@TXN:<Type>:<Date>:<Amount>$	Transaction details (type, date, amount)
				@TXN:7$	No more transactions to show
7	**PIN Change**	MCU → PC	#A:PIN:<RFID>:<NewPIN>$	@OK:DONE$	PIN updated successfully
8	**Card Blocking**	MCU → PC	#A:BLK:<RFID>$	@OK:DONE$	Card blocked (after 3 failed PIN attempts)
9	**System**	MCU → PC	#X:LINEOK$	@X:LINEOK$	Connection test (keep-alive)
10	**Session End**	MCU → PC	#Q:SAVE$	(None)	Graceful termination (no response expected)

Field Definitions:

- <RFID>: 8-digit card number (e.g., 12345678)
- <PIN>: 4-digit PIN (e.g., 1111)
- <Amount>: Transaction amount in rupees (e.g., 5000 = ₹5,000)
- <Name>: Account holder name (e.g., JohnDoe)
- <TxNo>: Transaction index (1, 2, 3...)
- <Type>: Transaction type (1=Withdraw, 2=Deposit, etc.)
- <Date>: Format DD/MM/YYYY (e.g., 01/06/2023)

Key Protocol Rules:

1. **Framing**:
   - All commands start with # (MCU→PC) or @ (PC→MCU) and end with $.
   - Example: #A:WTD:12345678:5000$ → Request to withdraw ₹5,000.
2. **Error Handling**:
   - @ERR:<Reason>$ indicates failures (e.g., @ERR:BLOCK$).
   - MCU retries or blocks card after 3 failed PIN attempts.
3. **Timeouts**:

- MCU waits for responses using r_flag (interrupt) or polling (getMsg()).

4. **Amount Limits**:
    - Max withdrawal/deposit: ₹30,000 per transaction.

ATM Project's Working Flow

1. System Initialization

[MCU BOOT] → [Peripheral Initialization] → [Welcome Screen]

- **Actions**:
  - MCU powers on and initializes:
    - UART (for PC communication)
    - LCD (16x2 display)
    - Keypad (4x4 matrix)
    - RFID reader
  - Displays: "Welcome To ATM"
  - Sends test command: #X:LINEOK$ to verify PC connection.

---

2. Card Authentication

[RFID Scan] → [Card Validation] → [Block/Proceed]

- **Steps**:
  1. User places card → RFID reader detects card and extracts 8-digit ID.
  2. MCU sends to PC: #C:12345678$
  3. PC responds with:
     - Valid: @OK:ACTIVE:JohnDoe$ (shows name on LCD)
     - Blocked: @ERR:BLOCK$ → Display "Card Blocked"
     - Invalid: @ERR:INVALID$ → Display "Card Not Found"

---

3. PIN Verification

[PIN Entry] → [Server Check] → [Access Granted/Block]

- **Process**:
  - User enters 4-digit PIN (masked as **** on LCD).
  - MCU sends: #V:12345678:1111$
  - PC checks and replies:
    - Correct: @OK:MATCHED$ → Proceeds to menu.
    - Wrong: @ERR:WRONG$ → Retry (3 attempts → blocks card).
  - **Timeout**: 30-second inactivity → auto-exit.

---

4. Main Menu Navigation

[Menu Display] → [Keypad Input] → [Transaction Execution]

- **Menu Options**:

1. WITHDRAW CASH 2. DEPOSIT CASH

```
3. VIEW BALANCE    4. MINI STATEMENT
5. PIN CHANGE      6. EXIT ATM
```

- **Navigation**:
  - A key: Scroll up
  - B key: Scroll down
  - 1-6 keys: Select option

---

5. Transaction Processing

A. Withdrawal

```
[Amount Entry] → [MCU: #A:WTD:12345678:5000$] → [PC Checks] → [Dispense Cash]
```

- **Responses**:
  - @OK:DONE$ → Dispenses cash + shows success.
  - @ERR:LOWBAL$ → Shows "Low Balance".
  - @ERR:MAXAMT$ → Shows "Exceeds Limit (₹30,000)".

B. Deposit

```
[Amount Entry] → [MCU: #A:DEP:12345678:10000$] → [PC Updates Balance] → [Confirmation]
```

- **Responses**: Same as withdrawal (success/error codes).

C. Balance Inquiry

```
[MCU: #A:BAL:12345678$] → [PC: @OK:BAL=25000$] → [LCD Shows Balance]
```

D. Mini-Statement

```
[MCU: #A:MST:12345678:1$] → [PC: @TXN:1:01/06/2023:5000$] → [LCD Shows Last 3 Transactions]
```

E. PIN Change

```
[Old PIN → New PIN → Confirm] → [MCU: #A:PIN:12345678:2222$] → [PC: @OK:DONE$]
```

- **Failure**: Mismatch → decrement tries.
- **Success**: Resets try counter.

---

6. Session Termination

```
[Exit Selected] → [MCU: #Q:SAVE$] → [LCD: "Thank You"] → [Return to Card Scan]
```

- **Timeout**: 30s inactivity → auto-logout with message: "Session Time-Out".

Error Handling & Security

- **Card Blocking**: After 3 failed PIN attempts → MCU sends #A:BLK:12345678$.
- **Protocol Security**:
  - All commands use #/@ framing to prevent corruption.

- Timeouts prevent infinite hangs.

Visual Flowchart Summary

```
START → Card Scan → Valid? → PIN Entry → Correct? → Menu → Transactions → Exit/Timeout
                  │                    │
                  └─Blocked Card       └─3 Fails → Block Card
```

Advantages

1. Security Benefits

- ⇒ **Two-Factor Authentication** – Combines RFID card + PIN for secure access
- ⇒ **Auto-Block Feature** – Locks card after 3 failed PIN attempts (#A:BLK:<RFID>$)
- ⇒ **No Sensitive Data Storage** – MCU doesn't store PINs/balances (validated by PC)

2. User Experience

- ⇒ **Simple Menu Navigation** – Scroll with ▲/▼ keys, select with 1-6
- ⇒ **Masked PIN Entry** – Shows **** instead of digits
- ⇒ **Session Timeout** – Auto-logout after 30s inactivity

3. Technical Strengths

- ⇒ **Structured Protocol** – Commands like #A:WTD:<RFID>:<AMT>$ ensure clarity
- ⇒ **Modular Code** – Each function (e.g., atm_wtd()) handles one task
- ⇒ **Efficient Resource Use** – No dynamic memory, fixed-size buffers

4. Error Handling

- ⇒ **Clear Error Codes** – e.g., @ERR:LOWBAL$, @ERR:BLOCK$
- ⇒ **Fail-Safe Design** – Transactions complete fully or roll back

5. Scalability

- ⇒ **Easy to Extend** – Add new commands (e.g., #A:TRANSFER) without redesign
- ⇒ **PC Backend Control** – Business logic (limits, fees) managed server-side

6. Cost-Effectiveness

- ⇒ **Low-Cost Hardware** – Uses basic MCU (LPC2148), 4x4 keypad, 16x2 LCD
- ⇒ **Power-Efficient** – No complex OS or background processes

7. Compliance & Standards

- ⇒ **Audit-Ready** – All transactions logged server-side
- ⇒ **Tamper-Resistant** – No user-accessible firmware/data on MCU

Applications

1. Banking & Financial Services

- **Traditional ATMs** - Cash withdrawals, deposits, and balance checks
- **Microfinance Solutions** - Rural banking for low-income users
- **Mobile Banking Kiosks** - Extend banking services to remote areas

2. Educational Institutions

- **University Campuses** - Student ID-based cashless transactions
- **Library Systems** - Pay fines or fees using RFID cards
- **Cafeteria Payments** - Food purchases with prepaid balances

3. Retail & Commercial Use

- **Self-Service Kiosks** - Gift card reloads and cashback services
- **Petty Cash Management** - Secure cash access for employees
- **Vending Machines** - RFID-based payment systems

4. Transportation Systems

- **Metro/Toll Cards** - Recharge travel cards and view usage history
- **Taxi Fleets** - Driver cash withdrawal systems

5. Healthcare Facilities

- **Hospital Payments** - Patient billing and insurance co-pays
- **Pharmacy Kiosks** - Prescription payments via health cards

6. Government Services

- **Welfare Distribution** - Subsidy withdrawals using citizen cards
- **Utility Payments** - Electricity/water bill payments

7. Hospitality Industry

- **Hotel Safes** - RFID-based locker access for guests
- **Resort Cashless Systems** - Payments via wristbands/cards

8. Customizable Extensions

- **Gaming Arcades** - Prepaid credit systems
- **Membership Clubs** - Secure access and payments
- **Event Management** - Cashless food/merchandise purchases

Outputs:

Files structures (MYFILE)

```
v24he6g2@srvr2:~/MYATM$
v24he6g2@srvr2:~/MYATM$ ls
atmConnected  bank  data  Hardware
v24he6g2@srvr2:~/MYATM$ cd atmConnected/
v24he6g2@srvr2:~/MYATM/atmConnected$ ls
atm  atm.c  atm.h  atmMain.c  atmMain.o  atm.o  makefile
v24he6g2@srvr2:~/MYATM/atmConnected$ cd ../bank/
v24he6g2@srvr2:~/MYATM/bank$ ls
bank  bank.c  bank.h  bankMain.c  bankMain.o  bank.o  makefile
v24he6g2@srvr2:~/MYATM/bank$ cd ../data/
v24he6g2@srvr2:~/MYATM/data$ ls
transactions.txt  users.txt
v24he6g2@srvr2:~/MYATM/data$ cd ../Hardware/
v24he6g2@srvr2:~/MYATM/Hardware$ ls
 atm.axf            atm.lnp        atm.plg            atm_uvproj.bak   kpm.crf   lcd.o      rtc.d         uart.c
 atm.build_log.htm  atm_main.c     atm.sct            delay.c          kpm.d     rfid.c     rtc.h         uart.crf
 atm.c              atm_main.crf   'atm_Target 1.dep' delay.crf        kpm.h     rfid.crf   rtc.o         uart.d
 atm.crf            atm_main.d     atm.tra            delay.d          kpm.o     rfid.d     startup.d     uart.h
 atm.d              atm_main.__i   atm.uvgui.DELL     delay.h          lcd.c     rfid.h     startup.lst   uart.o
 atm.h              atm_main.o     atm.uvopt          delay.o          lcd.crf   rfid.o     startup.o
 atm.hex            atm.map        atm_uvopt.bak      gpio.h           lcd.d     rtc.c      Startup.s
 atm.htm            atm.o          atm.uvproj         kpm.c            lcd.h     rtc.crf    types.h
v24he6g2@srvr2:~/MYATM/Hardware$
```

1. atmConnected (PC ↔ MCU)
   - ⇒ atm.h
   - ⇒ atm.c
   - ⇒ atmMain.c
   - ⇒ makefile

2. bank(Bank Databased)
   - ⇒ bank.h
   - ⇒ bank.c
   - ⇒ bankMain.c

3. data (Infromation and Histrory)
   - ⇒ transcation.txt
   - ⇒ user.txt

4. Hardware(MCU Side files)
   - ⇒ gpio.h
   - ⇒ types.h
   - ⇒ delay.h
   - ⇒ delay.c
   - ⇒ lcd.h
   - ⇒ lcd.c
   - ⇒ kpm.h
   - ⇒ kpm.c

⇒ rfid.h
⇒ rfid.c
⇒ rtc.h
⇒ rtc.c
⇒ uart.h
⇒ uart.c
⇒ atm.h
⇒ atm.c
⇒ atmMain.c

LPC2148 Interfacing Connections Table

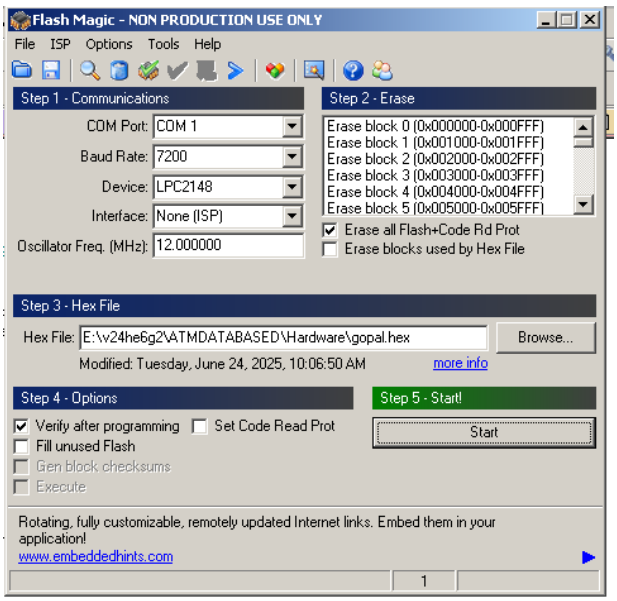
Peripheral	Signal/Pin Name	LPC2148 Pin	Connected To	Description
LCD 16x2	D0	P1.16	D0	Data Line 0
	D1	P1.17	D1	Data Line 1
	D2	P1.18	D2	Data Line 2
	D3	P1.19	D3	Data Line 3
	D4	P1.20	D4	Data Line 4
	D5	P1.21	D5	Data Line 5
	D6	P1.22	D6	Data Line 6
	D7	P1.23	D7	Data Line 7
	RS	P0.16	RS	Register Select
	RW	P0.17	RW	Read/Write
	EN	P0.26	EN	Enable Signal
	VCC	-	5V	Power Supply
	GND	-	GND	Ground
4x4 Keypad	Row 0 (R0)	P1.24	R0	Row Line
	Row 1 (R1)	P1.25	R1	Row Line
	Row 2 (R2)	P1.26	R2	Row Line
	Row 3 (R3)	P1.27	R3	Row Line
	Column 0 (C0)	P1.28	C0	Column Line
	Column 1 (C1)	P1.29	C1	Column Line
	Column 2 (C2)	P1.30	C2	Column Line
	Column 3 (C3)	P1.31	C3	Column Line
RFID Reader	UART1 RX (output from RFID)	P0.9	RFID TX	Serial Communication
	VCC	-	5V	Power Supply
	GND	-	GND	Ground

PC (via DB9 Cable)	UART0 TX	P0.0	DB-9 RX (Pin 2)	Serial Output to PC
	UART0 RX	P0.1	DB-9 TX (Pin 3)	Serial Input from PC

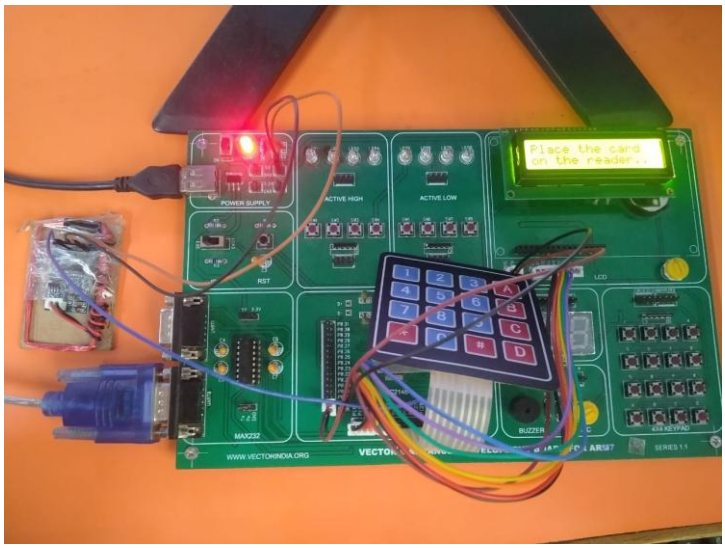
OUTPUT PHOTOS

Main Projects

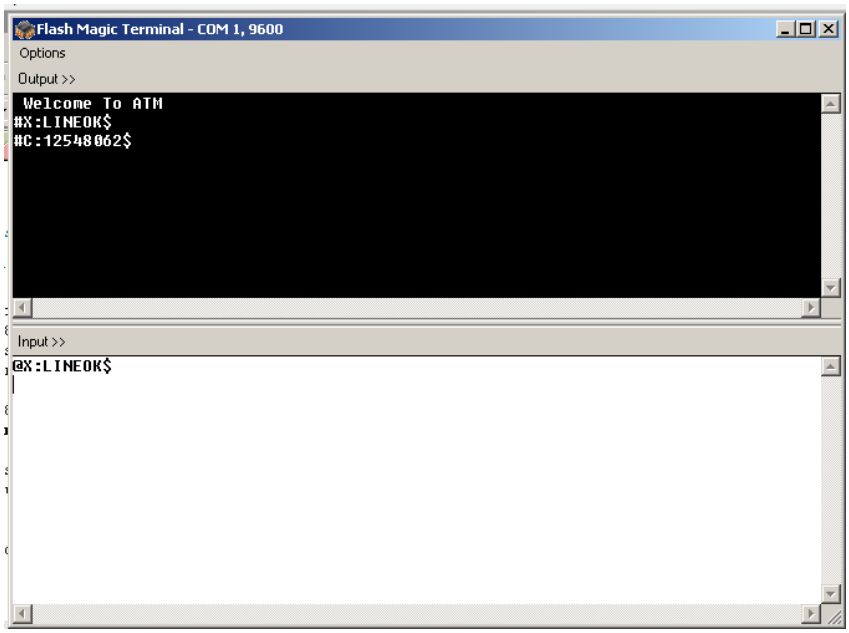
1. *Code Uploading*



2. *System Initialization*



3. *Test without PC*



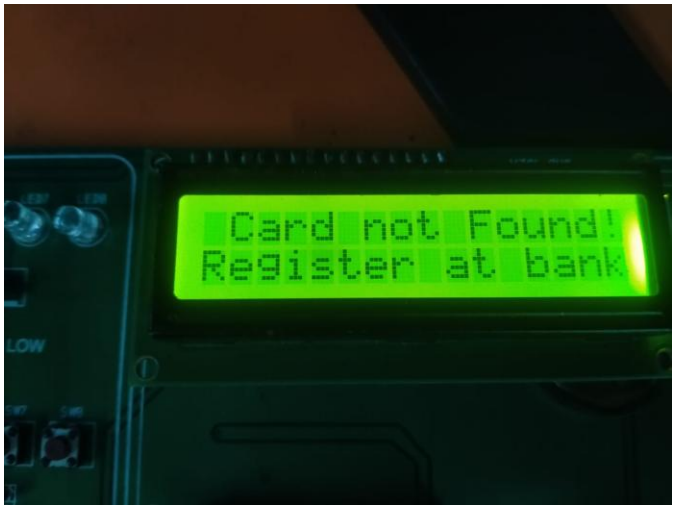
4. Admin (Bank side Information Fill)

```
Enter choice:c

Enter Name:gopal
Enter Mobile No.:9112301175
Enter Login Username:gopal22
Enter Login Password:1234
Re-enter Login Password:1234
Enter RFID card number:12529332
Enter ATM Pin:1234
Re-enter ATM Pin:1234
Enter Opening Amount:5000

==:Account Details:==
AccNo.:202506241015103660
Name  :Gopal
Ph.No.:9112301175
Balanc:5000.000000
Usrnam:gopal22
Passwd:1234
RFID  :12529332
Pin   :1234
Card  :ACTIVE
TranNo:1
Account Created.
```

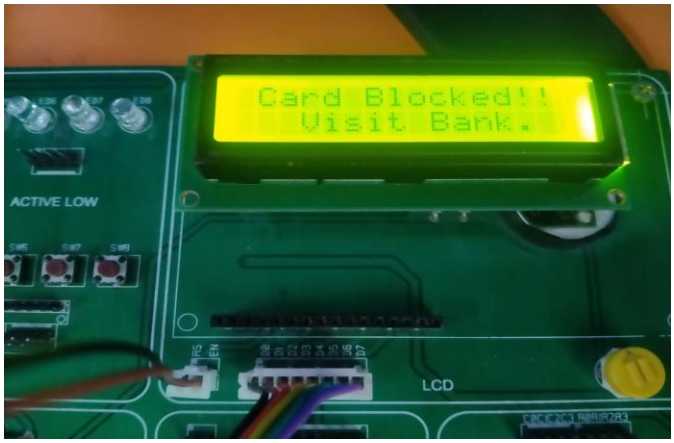
```
HI ADMIN:
[KEY]  : ACTION
c/C    : Create New account.
u/U    : Update Existing account.
h/H    : Transaction history.
w/W    : Withdraw amount.
d/D    : Deposit amount.
b/B    : Balance enquery.
t/T    : Transfer money.
x/X    : Activate card.
e/E    : Display all accounts details.
f/F    : Finding/searching for specific account.
q/Q    : Quit from app.
Enter choice:e


Account ID          |Holder Name                          |Mobile(+91)   |Transactions
202506241018034865  |India                                |+91-7412589630|1
202506241017054024  |Vector                               |+91-9876543210|1
202506241015103660  |Gopal                                |+91-9112301175|1
```

5. *Card Authentication*

a. Card Invalid



b. Card Block

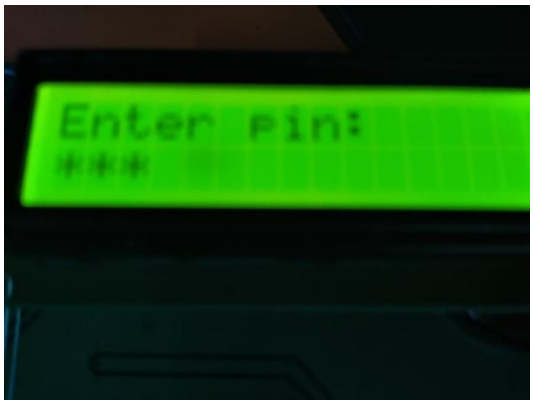


c. Valid Card

6. *Pin Varification*

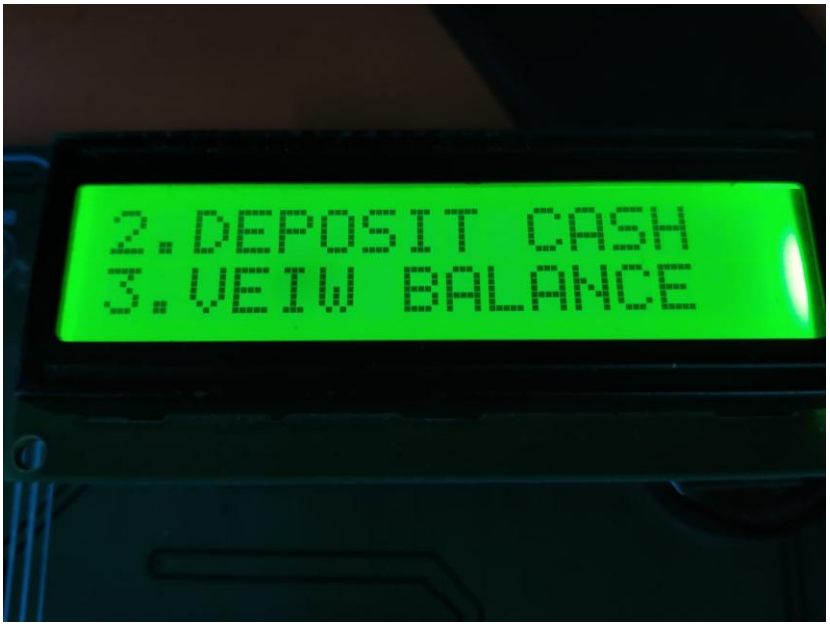
a. Wrong Pin

b. Correct or Matched Pin

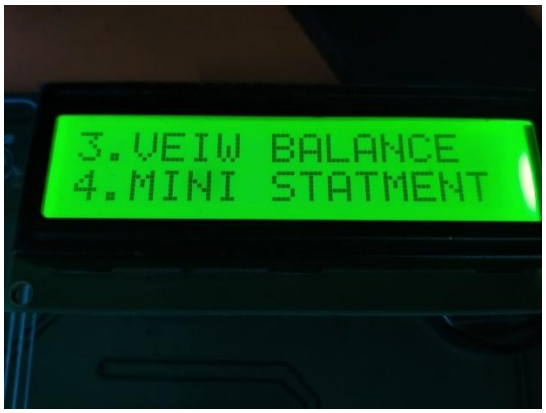


7. *Menu*
    1. Withdrawal

    2. Deposit



3. Balance Inquiry

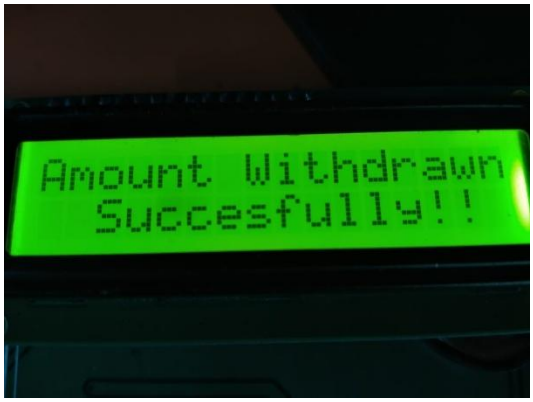


4. Mini Statement

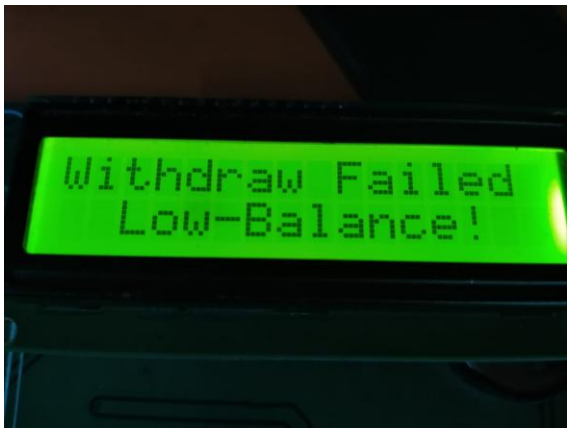
5. Pin change
6. Exit

7. *Withdrawal*

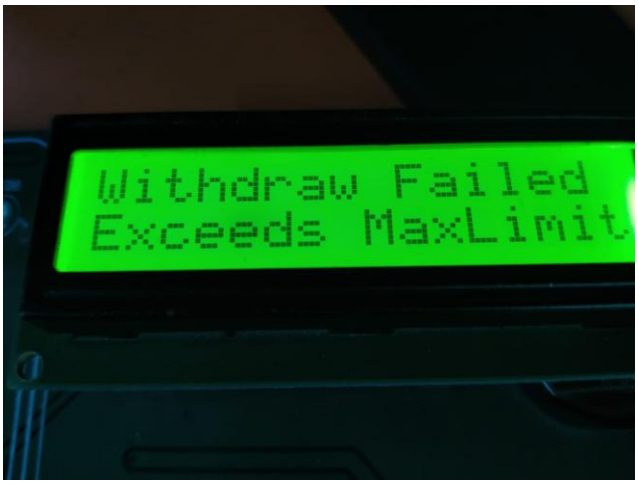
a. Success



b. Low Balance



c. Max Limit



8. *Deposit Successfully*

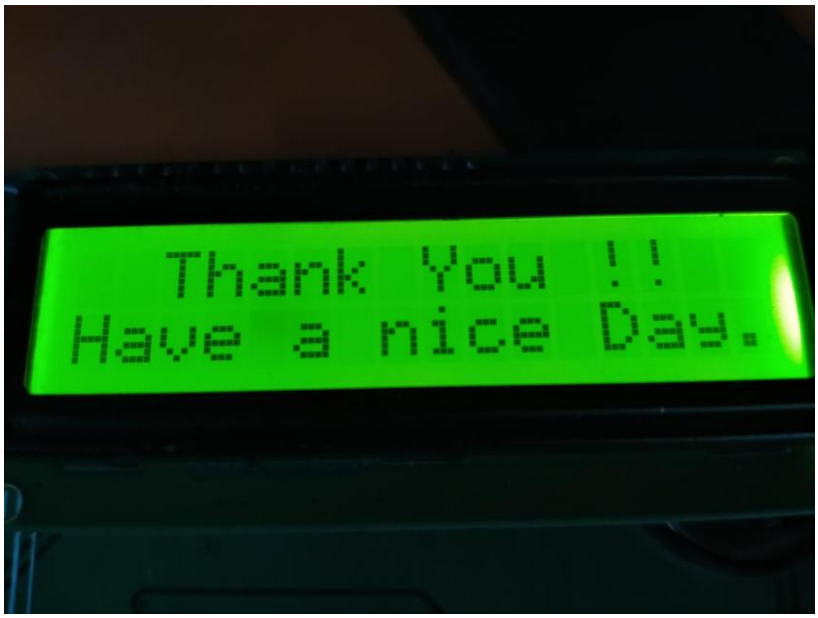
9. *Balance Statements*

10. *Mini Statement*

11. *Pin Change*

a. old pin

b. New pin

c. Successfully change pin

12. *Exit*