Term Project Report

on

G-Yantra : AT89S52 Based Gaming Device

Submitted By:

Tanvi AshishKumar Shah

[EC046]

[22ECUOG045]

Rudra Darshan Joshi

[EC009]

[22ECUOT048]

B. Tech. Sem. VI (2024-25)

Electronics and Communication

under the supervision of

Prof. (Dr.) Nikhil Kothari,

Professor (EC)

Dharmsinh Desai University

Department of Electronics and Communication,

Faculty of Technology,

Dharmsinh Desai University,

Nadiad - 387001

Dharmasinh Desai University
Faculty of Technology
College Road, Nadiad - 387001, Gujarat

Certificate

This is to certify that the Microcontroller and IOT Term Project Report work titled **G-Yantra : AT89S52 Based Gaming Device** is the bonafide work of **Miss. Shah Tanvi Ashishkumar** Roll No. **EC046** Identity No. **22ECUOG045** of B. Tech, semester VI in the branch of Electronics and Communication during the academic year 2024-2025.

Date: _____

Prof. (Dr.) Nikhil Kothari
Faculty Supervisor

Prof. (Dr.) Purvang Dalal
Head of the Department (EC)

"In a world dominated by high-performance processors and hyper-realistic graphics, creating a project that runs basic games on an old processor might seem trivial. However, the true purpose of this project was to deconstruct the complexity hidden beneath modern conveniences—to understand and recreate the foundational elements of gaming at the hardware level. Through this process, we not only gained a deeper appreciation for the ingenuity of early developers but also came to realize the remarkable effort it took to make what feels ordinary today seem effortless. This project stands as a tribute to the pioneers whose work laid the groundwork for the seamless experiences we now take for granted."

– Rudra Joshi & Tanvi Shah

Contents

Abstract

The objective of the G-Yantra project was to develop a replica of Nintendo's iconic 1990s Gameboy, employing the 8051 microcontroller programmed entirely in assembly language. This undertaking aimed to explore advanced applications of the 8051 architecture, providing both an academic perspective and practical insights into low-level programming. The project is designed to engage professionals in the embedded systems industry as well as enthusiasts of computer architecture.

G-Yantra integrates multiple hardware modules, including a 128×64 GLCD and an interrupt-driven keyboard interface, which facilitate user interaction with the microcontroller's software. The system is powered by a rechargeable Li-Ion battery pack.

In addition to hardware replication, the project features a simple graphical user interface and includes two assembly-coded games: Snake Game and 4-In-A-Row.

Through this implementation, the developers have gained valuable experience in programming assembly language for complex applications and have explored the fundamental principles underlying game logic on embedded hardware.

1 Background

1.1 Motivation

During the final weeks of the third phase of Semester 5, the faculty members associated with the Microcontroller Applications (MCA) lab assigned students the task of developing fundamental assembly language programs. These included interfacing keyboards, blinking LEDs, and generating square waves, all to be implemented and verified on an 8051-based microcontroller, specifically the AT89C51. For two weeks, the developers worked meticulously to ensure the successful execution and verification of these programs, preparing them for demonstration to their peers during the concluding lab sessions.

Through this process, the team encountered microcontroller programming at the hardware level for the first time, gaining hands-on experience with burning assembly programs onto microcontroller ICs. Unlike modern development platforms such as Arduino and ESP32, this experience provided valuable insight into the low-level intricacies of embedded system programming. Witnessing their assembly code execute flawlessly on physical hardware reinforced their understanding of theoretical concepts studied in the classroom, bridging the gap between theory and real-world implementation.

This initial experience ignited a deeper interest in assembly language programming. The developers were particularly drawn to the level of control and hardware interaction that assembly provided, allowing for direct manipulation of circuits with clarity and efficiency.

The pivotal moment of inspiration occurred during the testing phase when Rudra Joshi discovered a YouTube video titled ”The Insane Engineering of Gameboy” by Real Engineering. The video detailed the hardware and software architecture of Nintendo’s Gameboy, a revolutionary product of its time. What stood out was that the Gameboy’s processor—either Intel’s 8080 series or Zilog’s Z80 series—operated at just 1 MHz, yet it powered a range of classic games written in assembly language with a relatively simple instruction set.

This realization sparked a challenge: If developers in the past could create fully functional games on slower processors, why couldn’t a similar concept be implemented on the 8051? With this motivation in mind, Rudra proposed the idea of replicating the Gameboy’s core functionality using the 8051 microcontroller as a Semester 6 IoT project. After discussing the feasibility with the faculty, Rudra approached Tanvi Shah, who was leading another batch under the same faculty’s supervision and shared a similar passion for assembly language and computer architecture. With mutual enthusiasm and technical alignment, the developers collaborated to transform this vision into a full-fledged academic project, ultimately leading to the

development of G-Yantra.

1.2 Literature Review

To deepen our understanding of Nintendo's Game Boy functionality, architecture, and programming paradigms, we engaged with a series of instructional videos from the YouTube channel **NES Hacker**, which specializes in dissecting the Nintendo Entertainment System (NES) and related consoles. These videos provided foundational knowledge on various aspects of game console architecture and programming, including:

- **NES Graphics Explained** – Explored the graphical capabilities and limitations of the NES, detailing how sprites and backgrounds are rendered.
- **6502 Assembly Crash Course** – Offered an introduction to programming in 6502 assembly language, the core language for NES development.
- **NES Architecture Explained** – Provided an overview of the NES hardware components and their interactions.
- **NES Hardware Explained** – Delved into the specifics of NES hardware design and functionality.
- **The Code that Makes Mario Move** – Analyzed the programming techniques behind character movement in NES games.
- **Gameboy Development Environment** – Introduced tools and workflows for Game Boy game development.
- **Game Boy Graphics & How To Code Them** – Discussed the graphical subsystem of the Game Boy and methods for programming its visuals.

Through these resources, we gained insights into critical components such as video processors, serial communication interfaces, interrupt-driven keyboards, spirite-based graphics, sound processors, and techniques for enhancing user interactivity through sound and graphics

Understanding these elements was pivotal in conceptualizing our project, especially considering the Game Boy's unique architecture.

The original Game Boy, launched in 1989, featured an 8-bit custom CPU known as the Sharp LR35902, operating at approximately 4.19 MHz. This processor combined elements from both the Intel 8080 and the Zilog Z80 architectures, offering a simplified instruction set optimized for

handheld gaming applications. The system's graphical capabilities were managed by the Pixel Processing Unit (PPU), which handled background tile maps and sprite rendering, enabling efficient graphics performance within the hardware constraints of the time.

In contrast, the NES utilized the Ricoh 2A03 processor, a variant of the MOS Technology 6502 CPU, running at approximately 1.79 MHz. The 6502 assembly language, known for its simplicity and efficiency, was instrumental in NES game development, allowing programmers to write performance-critical code within the limited resources available.

These architectural insights underscored the importance of efficient programming practices and informed our approach to replicating similar functionalities using the 8051 microcontroller. Despite the absence of a dedicated video processor in the 8051, we aimed to emulate essential gaming features through meticulous software design and resource management, embracing the academic challenge of achieving complex functionalities within constrained hardware environments.

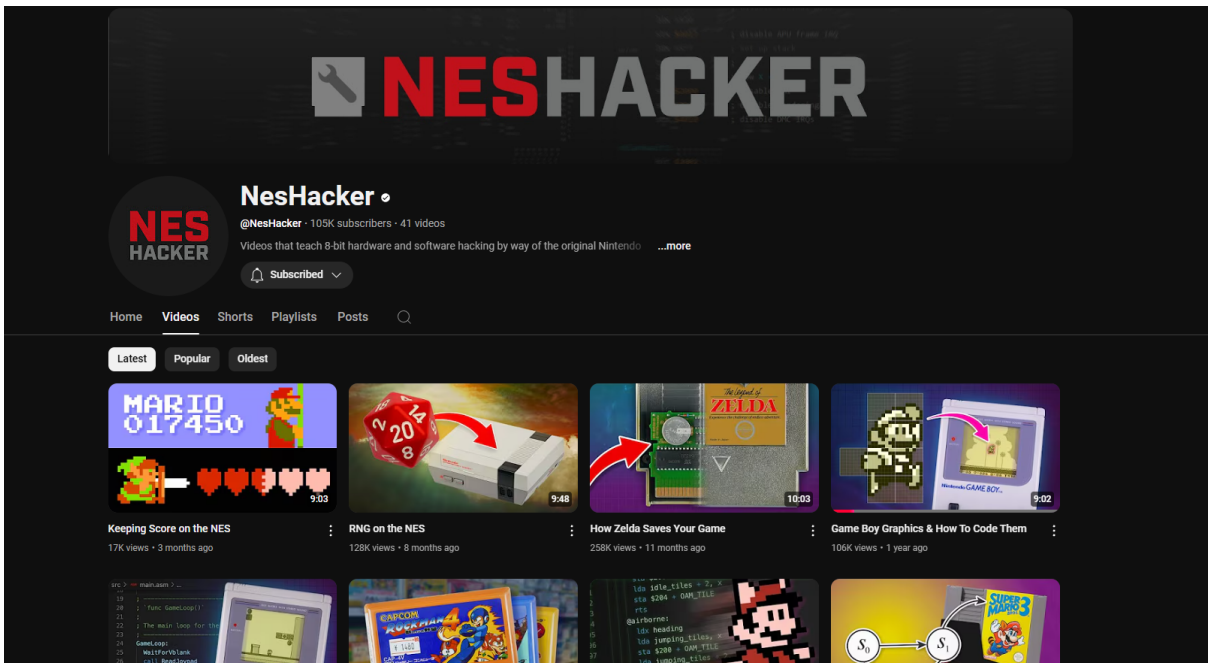


Figure 1: Homepage of NES Hacker YouTube Channel

1.3 Basic Theory

The 8051 microcontroller is a classic 8-bit device that executes instructions sequentially, allowing for straightforward control of various peripherals through its I/O ports. In earlier coursework and lab experiments, the development team interfaced the 8051 with simpler modules, such as a 16×2 character LCD and different keypad configurations (8×1, 4×3, and 4×4). These exercises served to verify basic functionalities, such as displaying text and scanning user input, and to demonstrate how such modules could be integrated into larger, more sophisticated systems.

Building on these foundations, the idea arose to combine a graphical display and a keyboard interface to create a gaming platform reminiscent of Nintendo's original Game Boy. As noted in the Motivation subsection, this concept was partly inspired by the excitement of exploring low-level, assembly-based programming and by the proven feasibility of integrating displays and keypads with the 8051, as documented in various online resources. One such example is a tutorial from Engineers Garage [10], which demonstrates how to connect a 16×2 character LCD and a 4×4 keypad to an 8051 using embedded C. Observing that the 8051 could handle both user input and display operations simultaneously strengthened our confidence in applying these principles to a 128×64 GLCD (for richer graphics) and an 8×1 keyboard (for game controls).

Beyond the display and keypad, the block diagram below highlights the essential components of our system. The 8051 microcontroller lies at the center, orchestrating interactions between the 128×64 GLCD, which renders the game interface, and the 8×1 keyboard, which captures user inputs for navigation and gameplay. An optional speaker module for audio processing may be incorporated if time permits, providing additional depth to the user experience. This design underscores how the fundamental 8051 theory covering I/O interfacing, instruction sequencing, and memory management can be leveraged to implement an end-to-end gaming solution on relatively constrained hardware.

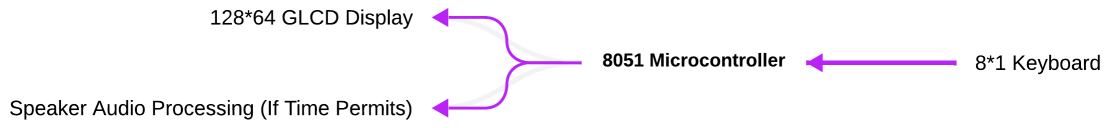


Figure 2: Basic G-Yantra Block-Diagram

2 Problem Definition and Design

2.1 Problem Statement

The G-Yantra project aims to replicate the iconic Gameboy experience using an AT89S52 microcontroller from the 8051 family. The primary challenge lies in integrating diverse peripheral devices—specifically, a 128×64 GLCD module for graphical output and a custom-built 8×1 keyboard for user input—within the constraints of limited processing power and memory. The task is to develop a system that not only executes assembly-coded games efficiently but also offers a responsive and reliable user interface through robust hardware-software integration.

2.2 Project Objectives

The following are the project objectives associated with the given problem definition:

- **Develop a Functional Prototype:** Create a working replica of a classic Gameboy that demonstrates both display and input functionalities using the AT89S52 microcontroller.
- **Effective Hardware Interfacing:** Successfully interface the 128×64 GLCD and the self-assembled 8×1 keyboard with the microcontroller, ensuring seamless communication between hardware components.
- **Assembly Language Programming:** Delve into assembly language programming to gain an understanding of the inner workings of the microcontroller. By writing code at the assembly level, we directly manipulate registers, memory, and I/O ports, thereby experiencing first-hand the precise control offered over hardware components. This approach not only allows for performance-critical optimizations in resource-constrained environments but also reinforces our theoretical knowledge of microcontroller architectures. Ultimately, this deep dive into low-level programming bridges the gap between conceptual design and practical implementation, providing valuable insights into how hardware devices can be efficiently managed and orchestrated within a combined system.
- **Educational Insights:** Provide a platform that deepens understanding of basic system design, low-level programming, and digital interfacing techniques.

2.3 Design Approach

2.3.1 Hardware Design Approach

The hardware design is centered around the AT89S52 microcontroller and focuses on two primary interfaces:

Display Interface: The system uses a 128×64 GLCD module to deliver the graphical output needed for game visualization.

Keyboard Interface: A custom-built 8×1 keyboard was constructed by soldering together components. It is integrated through an AND gate-based, interrupt-driven driver circuit that utilizes 8-input NAND and NOT gate ICs. This configuration connects to the microcontroller's external interrupt pin-0, ensuring rapid and reliable user input detection.

Reasons for the Hardware Interfacing Design Approach:

- **Real-Time Responsiveness:** An interrupt-driven approach ensures prompt processing of user inputs, essential for interactive gaming.
- **Legacy Compatibility:** The hardware choices align well with the AT89S52's capabilities, facilitating straightforward integration with peripheral modules.

Figure 3 shows a diagram of the system hardware interfacing, illustrating the interconnections between the microcontroller, GLCD, and keyboard.

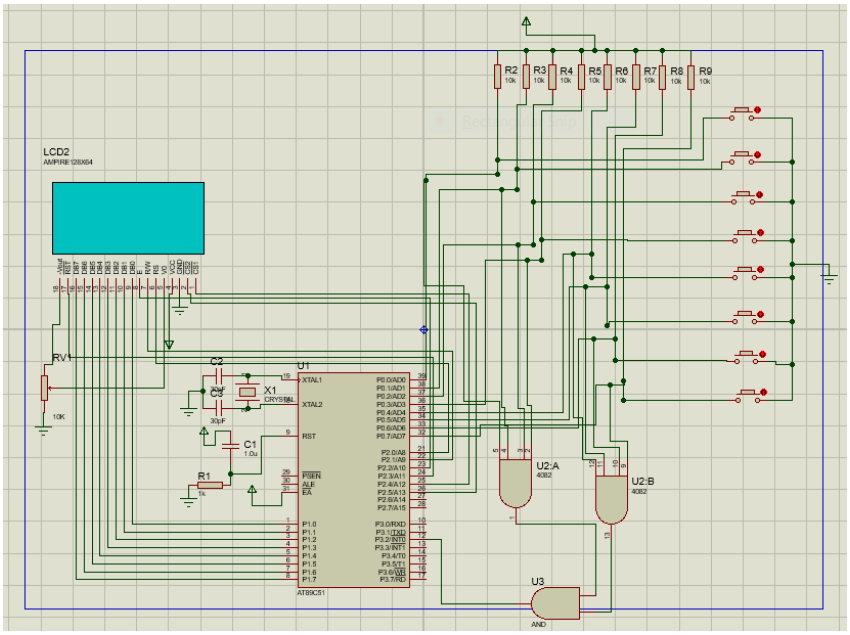


Figure 3: System Hardware Interfacing Diagram

2.3.2 Software Design Approach

The software design approach for the G-Yantra project is divided into four key areas:

1. GLCD Related Operations
2. Keyboard Based Operations
3. Graphical User Interface (GUI) Interactions
4. Game Operations

GLCD Related Operations: To effectively design software for GLCD (Graphical LCD) related operations, it's essential to understand the fundamental workings of the 128×64 GLCD module. This comprehension ensures efficient integration and functionality within the system.

Overview of the 128×64 Graphical LCD: Graphical LCDs (GLCDs) differ significantly from standard character LCDs such as 16×1, 16×2, 16×4, and 20×2 modules. Character LCDs are designed to display only alphanumeric characters or user-defined custom symbols, typically arranged within a fixed-size matrix—usually 5×7 or 5×8 pixels per character. These LCDs are limited to predefined font sizes and cannot render images or arbitrary graphical content.

In contrast, a 128×64 graphical LCD provides full control over 128 columns and 64 rows of pixels, resulting in a total of 8192 individual dots. Since each pixel can be independently manipulated, the display is highly flexible, allowing users to render characters of any size, display images, and even create animations. Unlike character LCDs, where text formatting is restricted to specific locations, a graphical LCD enables precise positioning of text, symbols, and graphical elements anywhere on the screen.

For example, in a 128×64 GLCD, an image occupying the entire screen would use 1024 bytes of memory (since each byte represents 8 vertical pixels). This ability to control individual pixels makes GLCDs an excellent choice for projects that require dynamic visual content, such as custom fonts, waveforms, gaming interfaces, and animated graphics.

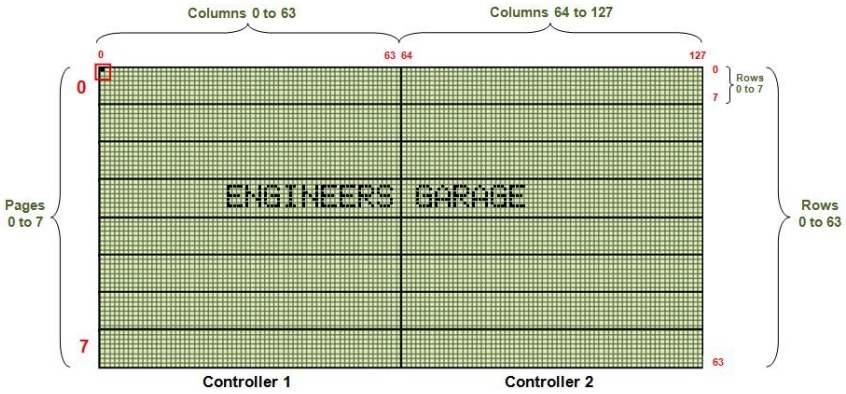


Figure 4: Pixel Arrangement of a 128×64 GLCD

Internal Structure and Controllers The 128×64 graphical LCD (GLCD) is controlled by two KS0108 controllers, each responsible for managing one half of the display. A single KS0108 controller can handle up to 4096 dots (512 bytes of memory), which means that to control 8192 dots (1024 bytes) in a 128×64 GLCD, two controllers are required.

To efficiently manage the display, the GLCD is divided into two equal halves:

- The left half of the display (columns 0 to 63) is controlled by the first KS0108 controller.
- The right half of the display (columns 64 to 127) is managed by the second KS0108 controller.

This division necessitates precise control when updating the display, as commands and data must be sent to the appropriate controller depending on the pixel location. Since each controller operates independently, careful software design is required to synchronize operations across both halves of the screen.

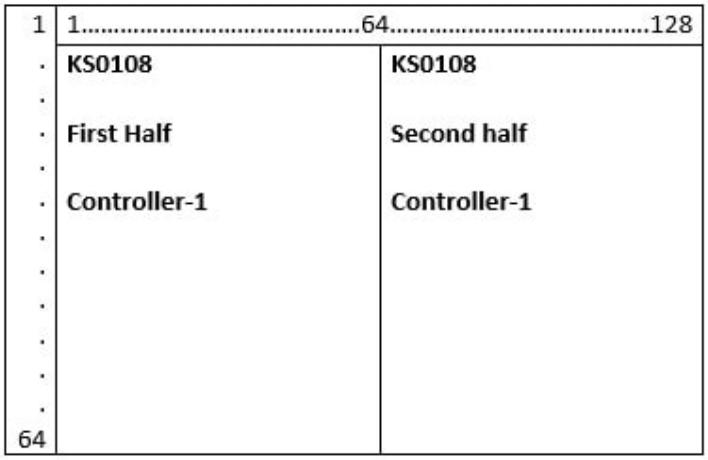


Figure 5: GLCD Division and KS0108 Controllers

Page and Memory Organization To efficiently manage pixel data, each half of the 128×64 GLCD is further divided into 8 pages of equal size. This segmentation allows for structured data storage and retrieval, making it easier to update specific regions of the display without affecting other sections.

Each page consists of:

- 8 rows and 64 columns, totaling 512 dots (pixels) per page.
- Since each pixel is stored as one bit, a single page requires 64 bytes of memory (as each byte represents 8 vertical pixels).
- The entire display (both halves) consists of 1024 bytes (8 pages × 64 bytes per page × 2 controllers).

Pixel Representation and Control:

- A single pixel in the GLCD is composed of 8 vertically stacked dots, meaning that 1 column and 8 rows form a pixel unit.
- Each pixel is turned ON when set to 0 (logic low) and turned OFF when set to 1 (logic high).
- By modifying these bits, users can generate custom graphics, fonts, and animations dynamically.

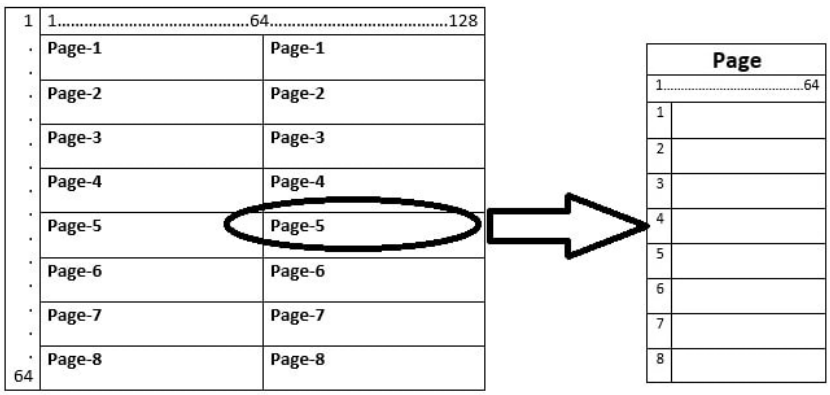


Figure 6: GLCD Page Structure Distribution

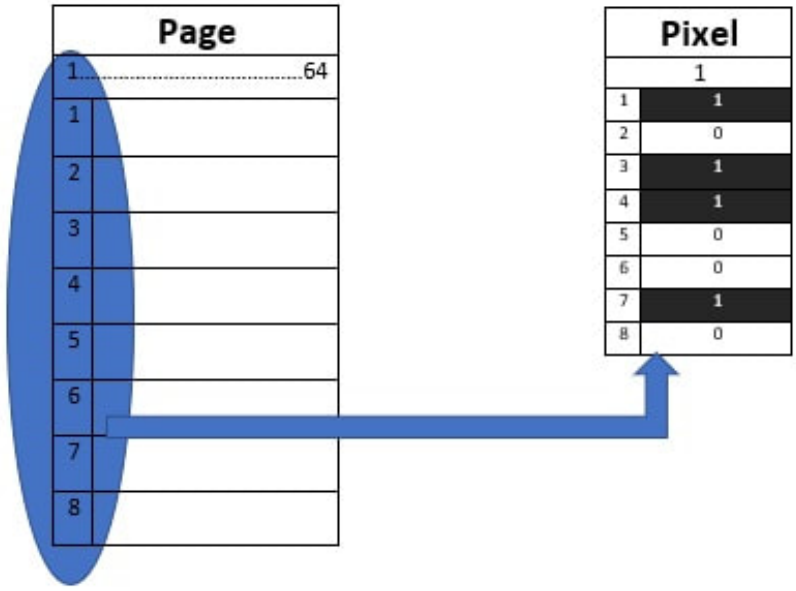


Figure 7: GLCD Pixel Structure Distribution

Pinout of 128×64 GLCD Using KS0108 Controllers: The GLCD communicates with a microcontroller using 20 pins, categorized into power, control, and data signals.

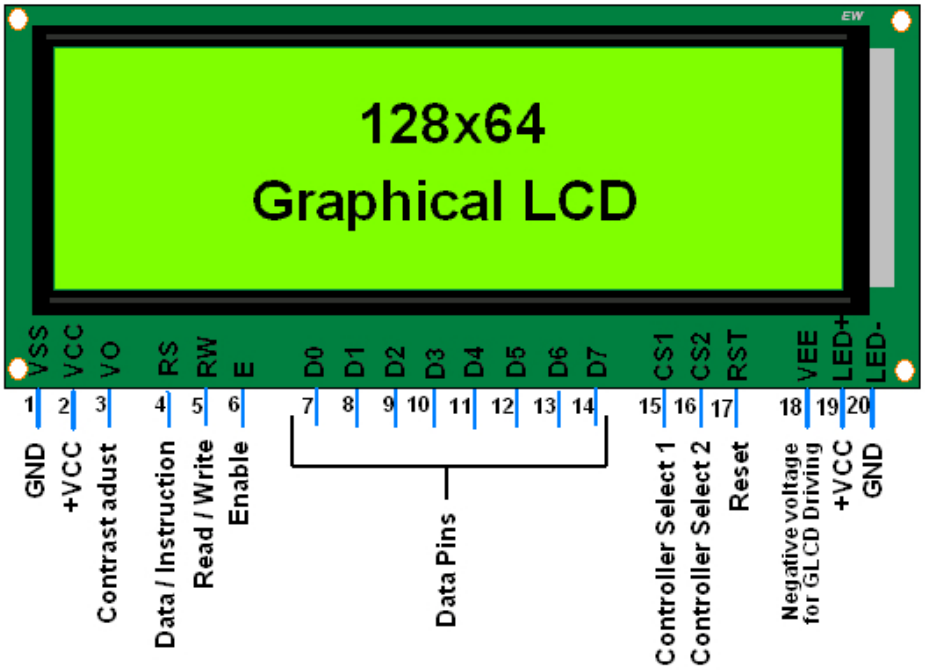


Figure 8: Pinout Diagram of 128×64 GLCD (JHD12864E).

Table 1 describes the function of each pin in detail.

Pin No.	Symbol	Description
1	VSS	Ground (0V reference)
2	VDD	Power supply for logic circuit (+5V)
3	V0	LCD contrast adjustment
4	RS	Register Selection: RS = 0 (Command), RS = 1 (Data)
5	R/W	Read/Write Selection: R/W = 0 (Write), R/W = 1 (Read)
6	E	Enable signal for data latching
7-14	DB0-DB7	8-bit data bus for parallel communication
15	CS1	Chip Select 1: Controls the left half of the display
16	CS2	Chip Select 2: Controls the right half of the display
17	RST	Reset signal: RST = 0 (Display Off, resets display memory)
18	VEE	Negative voltage (-10V) for LCD driving
19	LED+	LED backlight power (+5V)
20	LED-	LED backlight ground (0V)

Table 1: Pin Description of 128×64 GLCD (JHD12864E)

The control pins RS, R/W, E, CS1, CS2, and RST play a crucial role in selecting and configuring the display. By correctly setting these pins, the microcontroller can send commands and data to illuminate specific pixels on the screen.

Making Display on 128×64 GLCD: Once the GLCD has been initialized and the appropriate half and page selected, the next step is to display text and images. Each page of the GLCD is structured in an 8×64 matrix, meaning that each page consists of:

- 8 rows (vertical pixel alignment)
- 64 columns (horizontal data storage)
- Each column contains 8 dots (arranged vertically)

Since the GLCD operates in an 8-bit parallel mode, we send 8-bit data to control the ON/OFF state of each column's vertical dots.

- A pixel is ON when set to '1' (logic high).
- A pixel is OFF when set to '0' (logic low).

For example:

- Sending `FF` (11111111 in binary) will turn ON all 8 dots in a column.
- Sending `F0` (11110000 in binary) will turn OFF the first four dots and turn ON the last four dots.
- By manipulating these binary values, we can generate text, symbols, and graphical patterns.

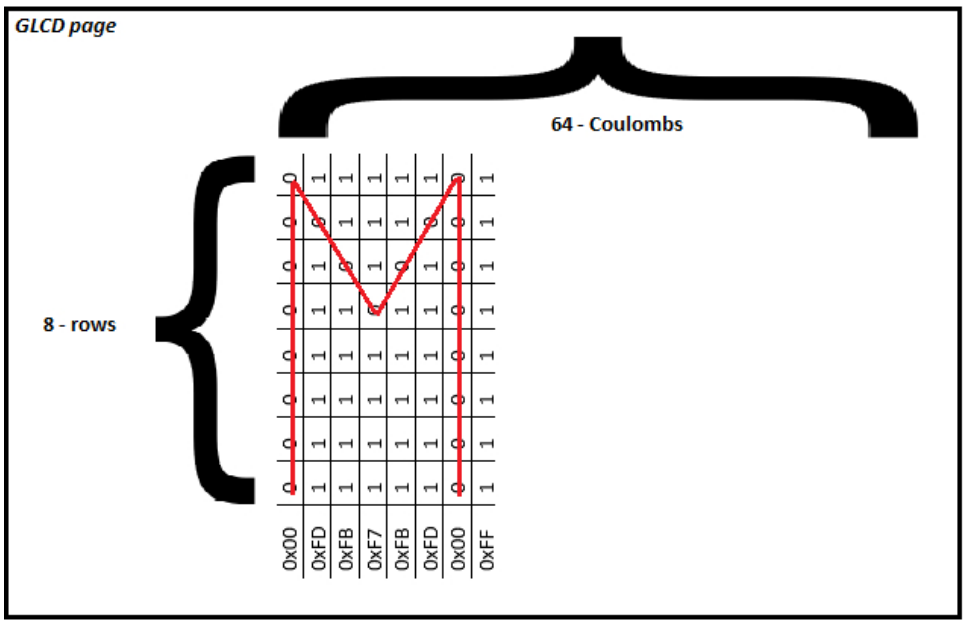


Figure 9: Mapping of Binary Data to GLCD Pixels for Character Display

Table 2 shows an example of how the letter **'M'** is stored in memory and sent to the GLCD.

Column	Hex Data	Binary Representation
1	0x00	0000 0000
2	0xFD	1111 1101
3	0xFB	1111 1011
4	0xF7	1111 0111
5	0xFD	1111 1101
6	0x00	0000 0000
7	0xFF	1111 1111

Table 2: Binary Data Representation for Displaying 'M' on GLCD

Each 8-bit column represents a vertical slice of the character. When these columns are combined sequentially, they form the shape of the **'M'** character. This process is repeated to display full words and graphical elements on the screen.

Implementation of GLCD Subroutines: The GLCD subroutines for the G-Yantra project are designed to facilitate the interaction between the 8051 microcontroller (AT89S52) and the 128×64 GLCD module. These subroutines are organized to handle initialization, command and data writing, column and page selection, and keyboard operations. The following is an overview of the implementation:

Subroutine Overview: The main firmware follows these steps:

1. **Initialize GLCD:** Configure ports, initialize stack pointer, and allow for stabilization through delay routines.
2. **Select GLCD Half:** Use Chip Select signals (CS1 and CS2) to choose between the left and right halves of the display.
3. **Select Page:** Choose one of the 8 vertical pages (each page representing 8 rows) in the active half.
4. **Display Text:** Write 8-bit data to the GLCD to form characters or images.

Command and Data Write Subroutines: The **cmdwrt** subroutine writes commands to the GLCD by:

- Waiting for the GLCD to be ready (using a delay).
- Sending the command byte to the data port.
- Clearing RS to indicate command mode.
- Clearing R/W to set write mode.
- Generating an enable pulse by toggling the E pin.

Similarly, the **datawrt** subroutine writes data to the display by setting RS to indicate data mode.

Column and Page Selection: The **set_column** subroutine computes the starting column address based on the pixel grid and selects the appropriate half of the GLCD. If the target column exceeds 64, the routine automatically selects the right half of the display.

Design Considerations in the set_column Subroutine: The **set_column** subroutine is implemented to send an 8-bit command that modifies an entire block of 8 columns in the GLCD. This design choice means that:

- Instead of addressing each of the 128 columns individually, the display is virtually divided into 16 groups (since $128 \div 8 = 16$).
- Combined with the organization of the GLCD into 8 rows per page, this results in a virtual grid of **8 rows** by **16 columns**.
- While this approach simplifies addressing by aligning with the 8-bit data width of the GLCD, it also makes it difficult to update a single column independently, as each instruction updates an entire 8-column block.

This trade-off reduces the complexity of writing display routines by decreasing the number of commands needed to update large sections of the screen. However, it also imposes the constraint that any modifications must be made to these pre-defined 8-column blocks, which can complicate the development of routines that require pixel-level or column-level precision.

set_pg_cntrl subroutine selects one of the 8 vertical pages in the active half, allowing precise control over which section of the display is updated.

Delay and Debounce Routines: To ensure reliable operation, a 1 ms delay subroutine is used for timing control, ensuring that the GLCD has sufficient time to process commands. A separate debounce subroutine (**dboun**) is implemented for handling keyboard inputs, ensuring stable key detection.

Block Diagrams of GLCD Subroutines: The following block diagrams illustrate the key GLCD subroutines and their algorithmic flow.

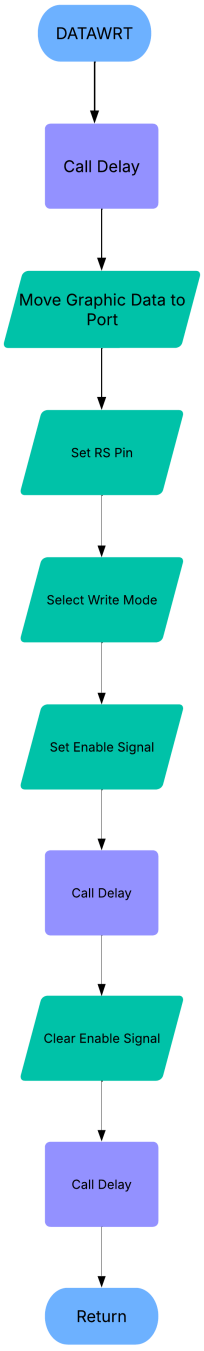


Figure 10: Block Diagram of DATAWRT Subroutine

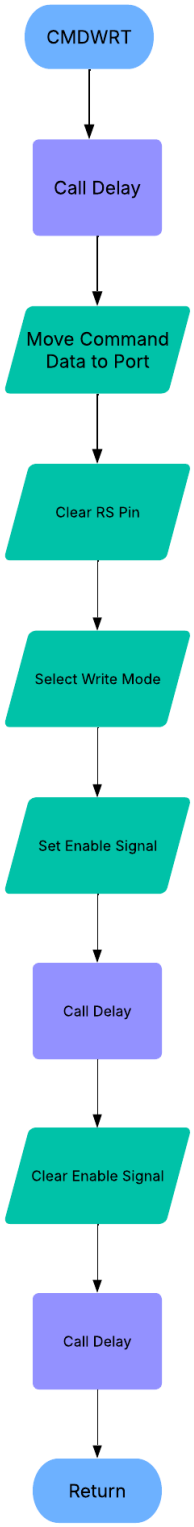


Figure 11: Block Diagram of CMDWRT Subroutine

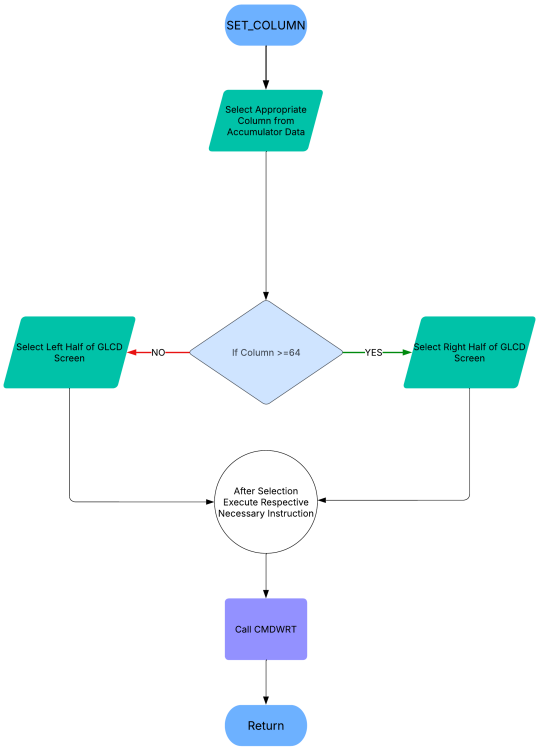


Figure 12: Block Diagram of set_column Subroutine

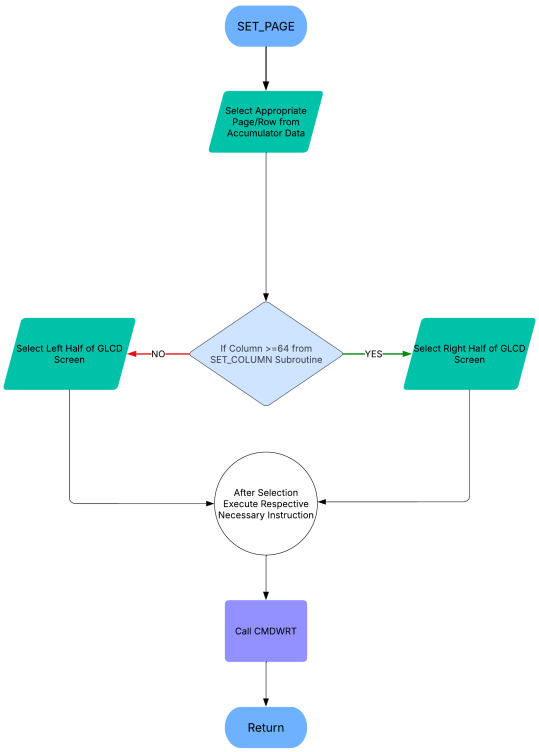


Figure 13: Block Diagram of set_pg Subroutine

Keyboard-Based Operations: In this project, an 8×1 custom keyboard is interfaced with the 8051 microcontroller through **External Interrupt 0 (INT0)**. Whenever a key is pressed, the microcontroller triggers the keyboard ISR, which debounces the input, identifies the pressed key, and updates the relevant game or GUI states.

ISR Logic for Integrated Snake/GUI: In the integrated code (Snake + GUI), the ISR checks whether the key press pertains to:

- **GUI Navigation:** Moves the cursor up/down in the game selection menu or toggles game selection bits.
- **Game Start:** Sets a start bit to begin the Snake game loop.
- **Direction Control Snake Game:** To handle Snake's Head movement directly in the ISR based on Current Direction Key Pressed.

Also the reason for Explaining the Code Snake plus GUI Integrated ISR and 4 for In a Row ISR Differently is Because while preparing this document the complete Integration of 4 in a Row game is yet to be done but in the code provided in the Annexure both the ISR's will have been unified into one as suggested in paragraphs below

ISR Logic for 4 in a Row: When a key press is detected:

- **Debounce & Identify Key:** The `dboun` routine ensures stable input, then the accumulator is shifted to find which bit is active.
- **Left/Right Movement:** The coin's previous position is cleared, the new position is calculated, and the coin is redrawn.
- **Select Key:** Finalizes the coin's placement on the grid (8x16 Virtual Grid Implemented with Help of Developed Subroutines) and toggles the current player bit.

In both cases, the ISR begins by calling a *debounce* subroutine (`dboun`) and reading `P0` to identify which key is pressed. A rotation (`rrc a`) and comparison loop pinpoints the exact bit representing the pressed key. Status bits (e.g., `08h`, `09h`, `0Ah`, `0Bh`) determine whether the firmware is in GUI mode, waiting to start a game, or already running a particular game.

Merging the ISRs: Both ISR's are combined as follows:

1. **Trigger Interrupt:** A key press invokes the INT0 ISR.
2. **Debounce:** The `dboun` subroutine ensures stable key detection.
3. **Key Decode:** Using `rrc` and looping we identify which key (0–7) was pressed.
4. **Check Active Mode:** Read a status bit or variable indicating "GUI," "Snake Game", or "4 in a Row,"
5. **Execute Logic:**
   - For GUI, move the cursor or select an item.
   - For "Snake Game" determine new Head Direction.
   - For "4 in a Row," move the coin, handle wraparound, toggle player bits.
6. **Update Display & Return:** Redraw any necessary elements on the GLCD and `RETI`.

This modular design ensures that new games or GUI screens can be added simply by introducing additional flags or bits to differentiate states within the same interrupt routine.

The following block diagram display the program flow associated with Keyboard's ISR

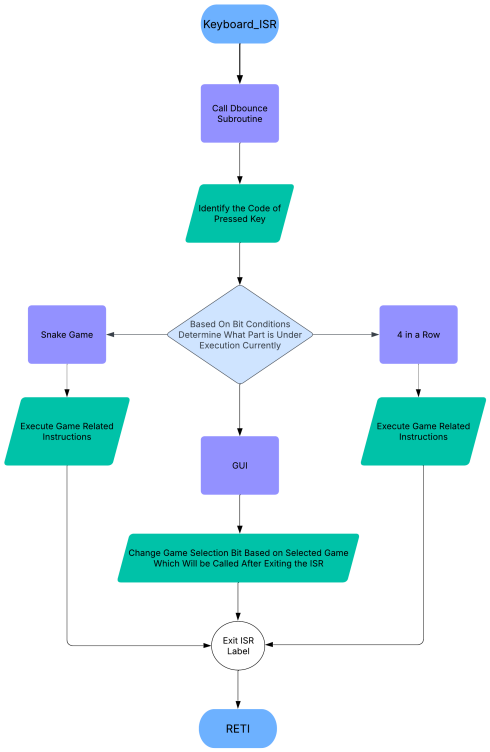


Figure 14: Block Diagram of Keyboard ISR Logic

This approach allows both games (and any GUI elements) to coexist seamlessly, simplifying user interactions via a single 8×1 keyboard.

Game Based Operations: In this project two games have been developed namely the Snake Game and the 4 in row Game. We'll Discuss the basic theory and operations associated with these two games.

Snake Game: The **Snake Game** is a classic arcade game where the player controls a growing snake that moves around the screen, collecting food while avoiding collisions with itself and the screen boundaries. The game is simple but requires strategic movement to maximize the score.

History and Background:

- The first known snake-type game, **Blockade**, was released in 1976 by Gremlin.
- The game became widely popular in the late 1990s when it was included in Nokia mobile phones.
- Variants of the game exist on various platforms, from early arcade systems to modern gaming consoles and mobile applications.
- The game mechanics involve a continuous movement system where the snake grows longer each time it eats food.

Figure 15: Nokia's Famous Snake Game

Game Rules:

- The player controls the snake's movement using directional inputs (up, down, left, right).
- The snake moves in a fixed direction and cannot stop.
- Eating food increases the snake's length and score.
- The game ends if the snake collides with itself or the boundaries of the screen.

Representing Game Visually on GLCD:

- The screen is divided into a grid where each cell is **8×8 pixels**.
- The snake is represented as a series of connected blocks.

Handling Input (Snake Movement):

- Push buttons for controlling the direction (left, right, up, down).
- The snake moves in a continuous direction until a new direction is chosen.

Collision Detection and Game Over Conditions:

- The game checks for collisions with:
  - The screen boundary.
  - The snake's own body.
- If a collision occurs, the game ends and displays a "Game Over" message.

Key Routines Associated with Snake Game:

- Snake Movement Routine
- Food Generation Routine
- Collision Detection Routine
- Update Screen Routine

These Routines are further explained visually in the form of block diagrams below:

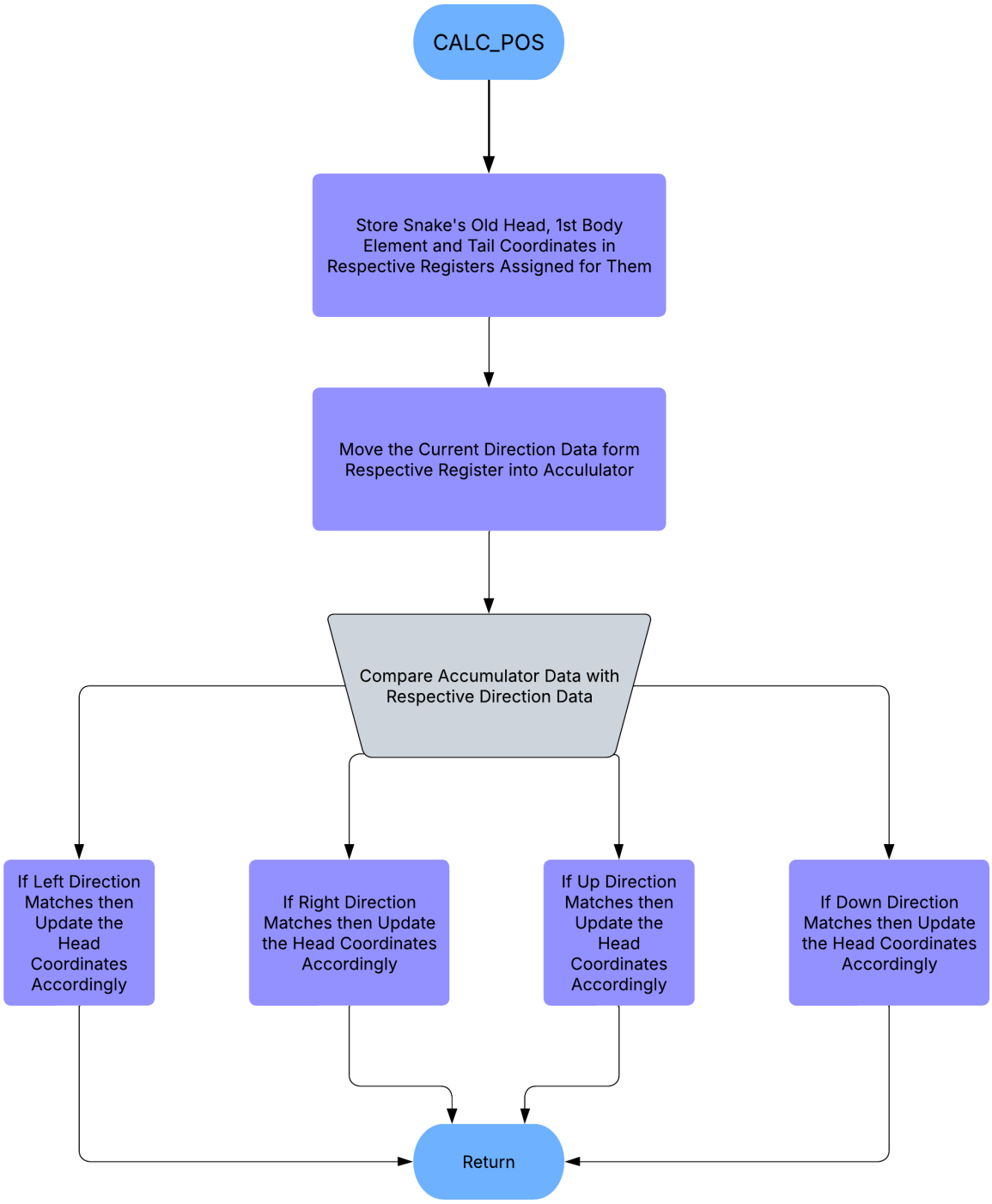


Figure 16: Block Diagram of CALC_POS subrotine

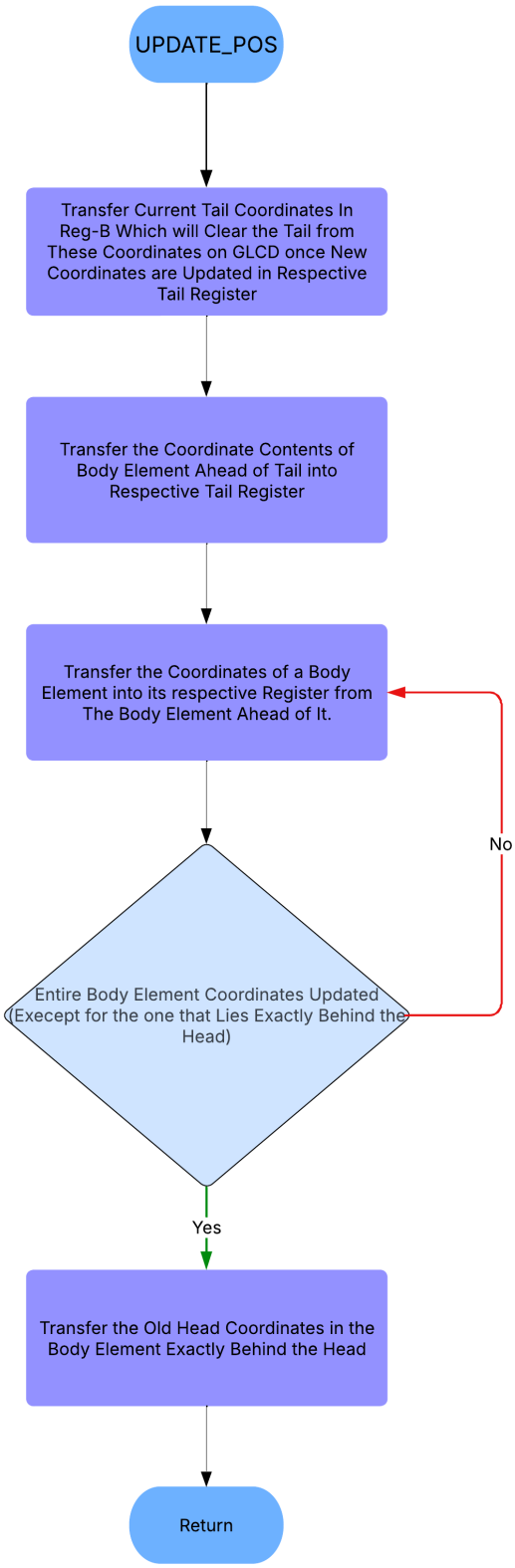


Figure 17: Block Diagram of UPDATE_POS Subroutine

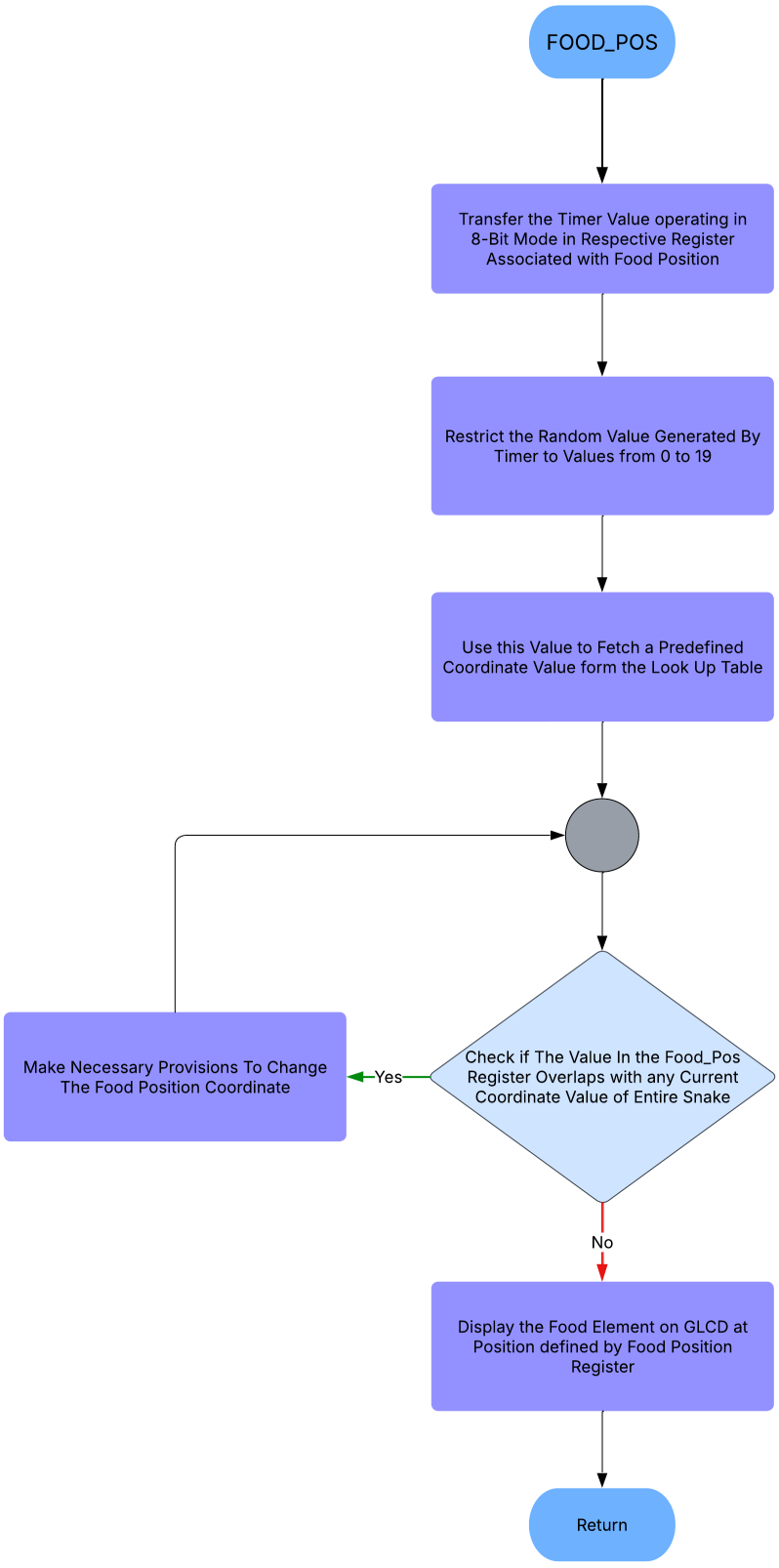


Figure 18: Block Diagram of FOOD_POS Subroutine

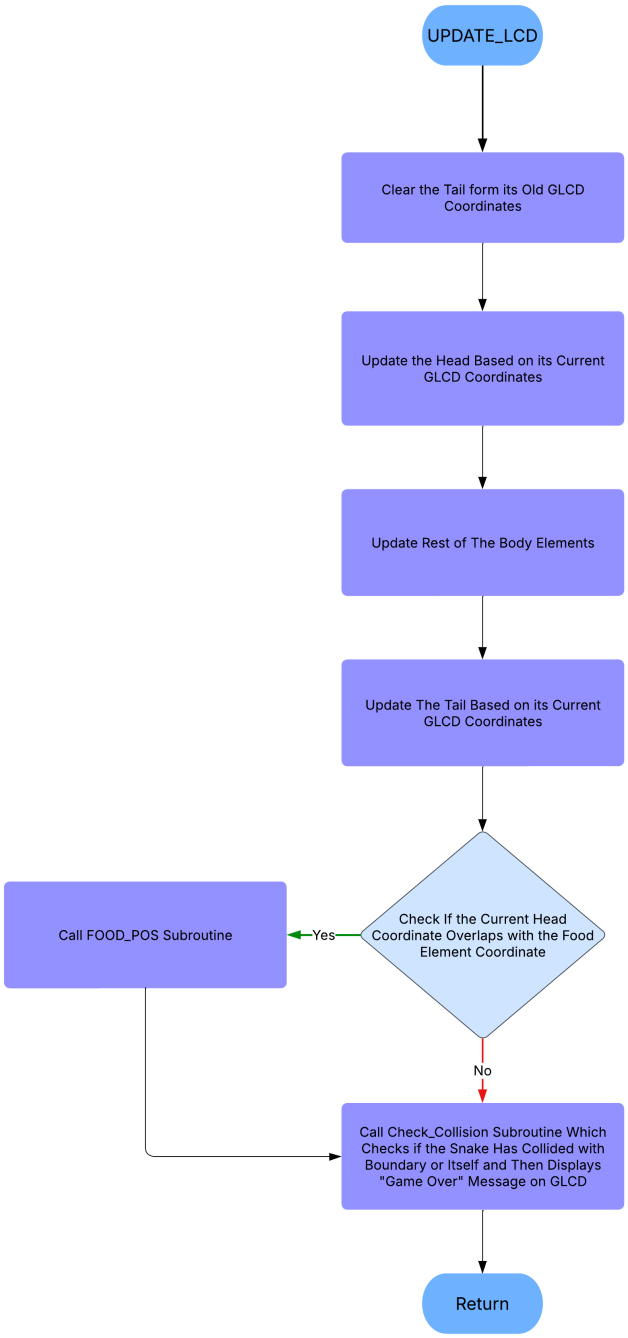


Figure 19: Block Diagram of UPDATE_LCD Subroutine

4 in a Row Based Operations: '4 in a Row' (also known as *Connect Four*) is a classic two-player strategy game where players take turns dropping pieces into a grid. The objective is to align four of their pieces consecutively in a row, column, or diagonal.

History and Background:

- The game was first conceptualized as a connection-based game and later patented by **Howard Wexler** and **Ned Strongin** in 1974.
- It was commercialized by **Milton Bradley** (now owned by Hasbro) and quickly became a widely recognized board game.
- **Mathematical Analysis**: The game was solved by **James D. Allen** in 1988, proving that the first player can always win with optimal strategy. The same was independently verified by **Victor Allis**.
- The standard game board consists of **7 columns** and **6 rows**.
- The total number of possible board positions is approximately:

$$4,531,985,219,092 \tag{1}$$

- The best opening move in the standard version is placing a piece in the center column.
- The game has inspired digital adaptations, AI research, and various rule variations such as **PopOut**, **Mega Connect Four**, and 3D versions.

Figure 20: 4 in a Row Game

Game Rules

- Players alternate turns dropping a piece into a column.
- The piece falls to the **lowest available row** in that column.

- A player wins if they align **four pieces in a row, column, or diagonal**.
- If the grid is full and no player has won, the game results in a **draw**.

Representing Game Visually on GLCD:

- Each cell (block) is **8×8 pixels**.
- Showing Grid and Coins for both players On Screen

Internal Logic for Mapping Screen to Memory:

- A 1D array stores the state of each cell:
  - 0: Empty cell
  - 1: Player 1 piece
  - 2: Player 2 piece

Handling Input (Move Selection):

- Push buttons for left/right movement .
- and Select for dropping a piece

Win Condition Checking:

- Check for four consecutive pieces in:
  - Horizontal direction
  - Vertical direction
  - Diagonal directions (left and right slopes)

Key Routines Associated with 4 in a Row Game:

- Input Handling Routine
- Coin Drop Routine

- Win Check Routine

These Routines are further explained visually in the form of block diagrams below:

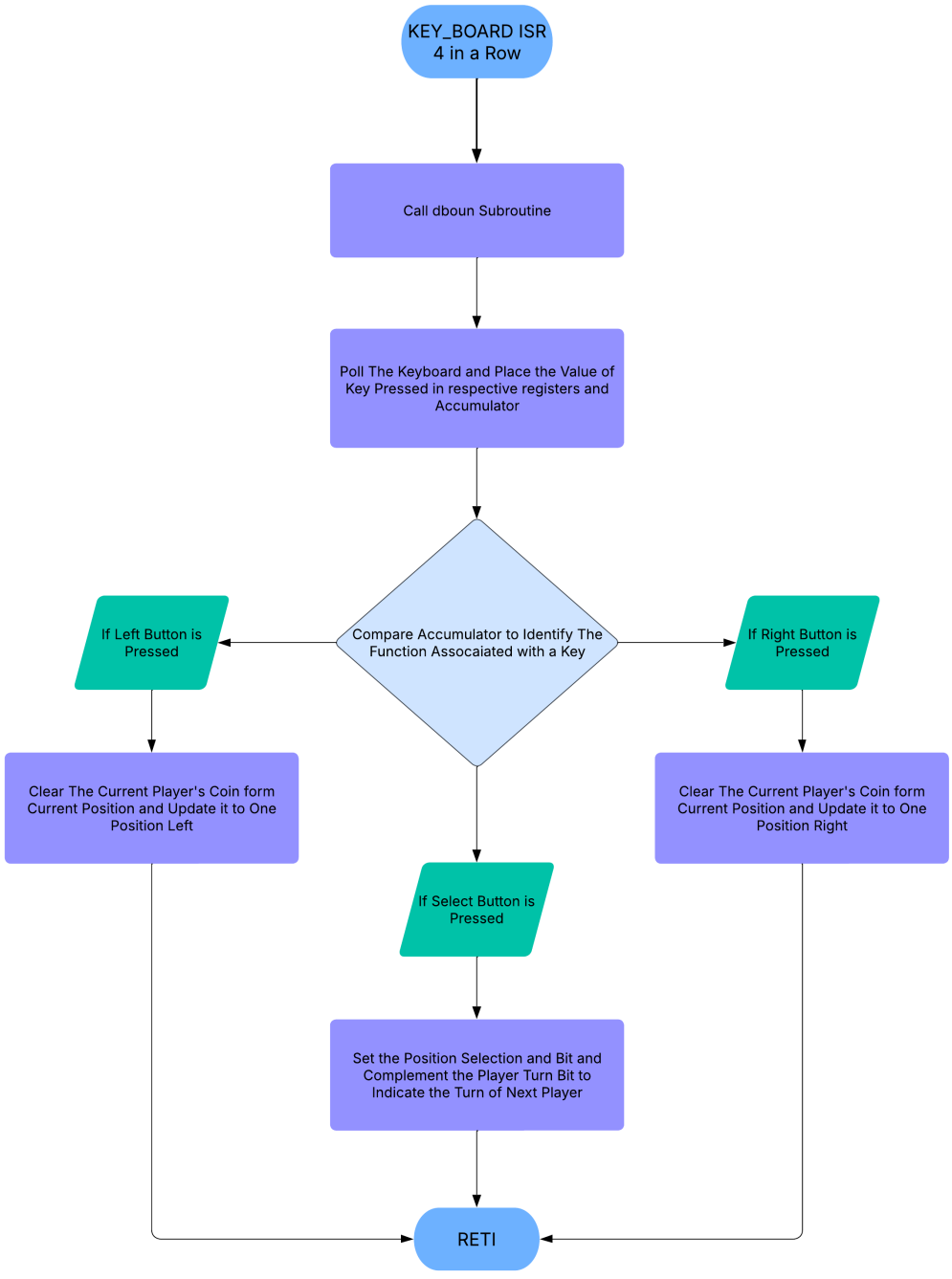


Figure 21: Block Diagram of KEYBOARD ISR for 4 in a Row

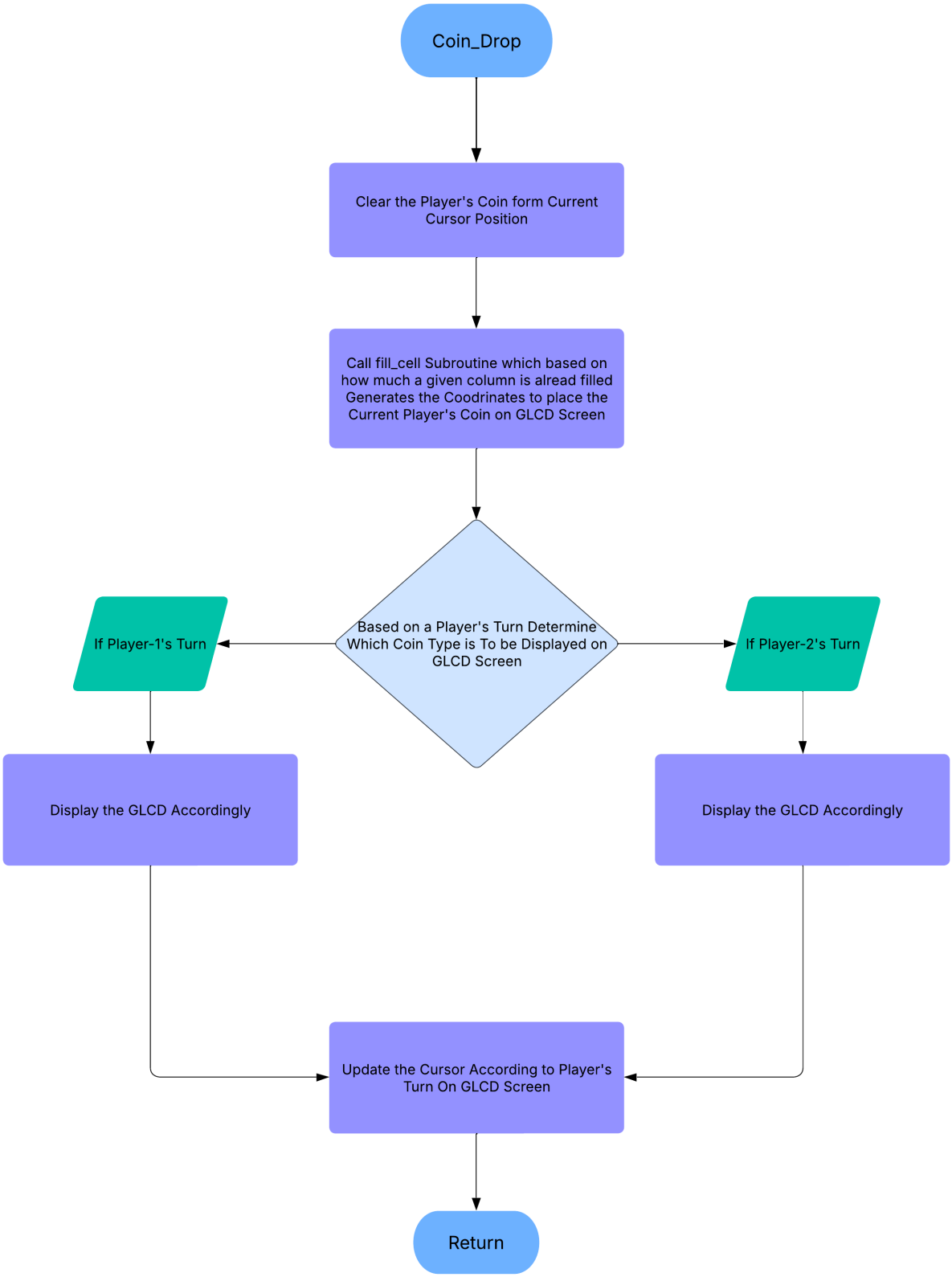


Figure 22: Block Diagram of COIN_DROP

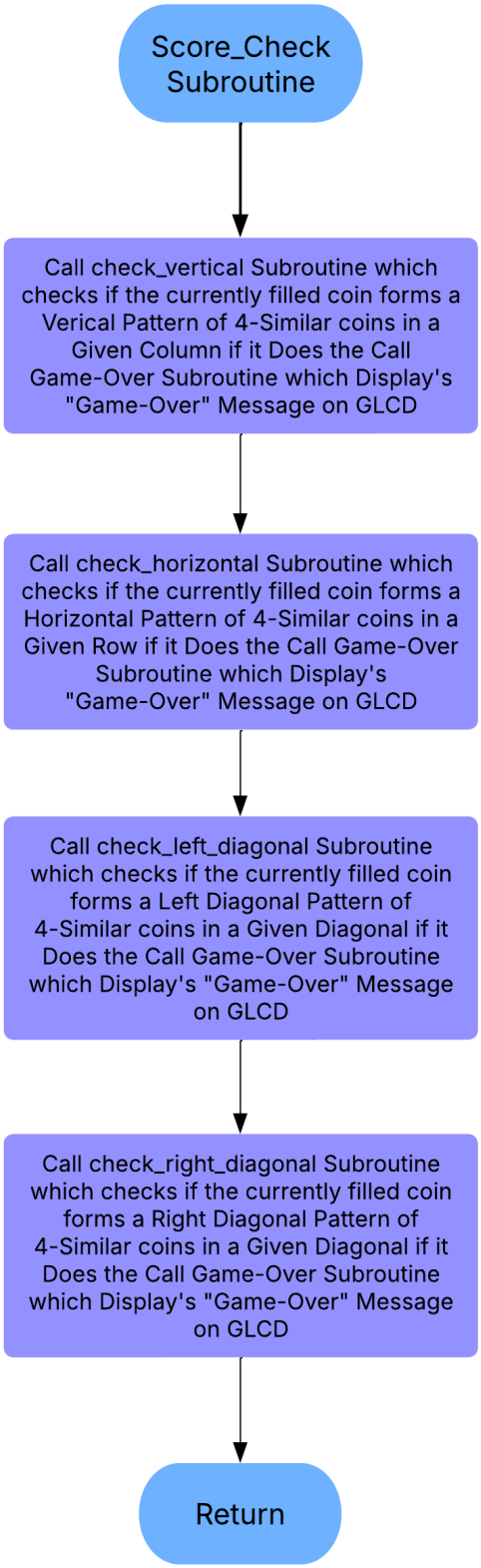


Figure 23: Block Diagram of SCORE_CHECK

3 Test Setup and Methodology

3.1 Initial Software Development

The development process began with setting up basic communication between the AT89S52 and the 128x64 graphical LCD (GLCD). Since the GLCD operates differently from a standard character LCD, it was crucial to understand its internal architecture, including how data is stored and how individual pixels are addressed. The first objective was to display simple text on the screen, which served as a fundamental test to verify whether the microcontroller was correctly interfacing with the GLCD.

To achieve this, the first step involved configuring the control signals of the GLCD, such as ‘CS1‘, ‘CS2‘, ‘RS‘, ‘RW‘, and ‘E‘, ensuring that data could be written to the display’s memory. The initialization sequence was carefully implemented to properly set up the GLCD, including resetting the display, configuring the contrast, and enabling both halves of the screen.

Once the GLCD was initialized, the next step was to display text on the screen. Since the GLCD operates on a pixel-based system, text characters are not directly supported as they are in standard alphanumeric LCDs. Instead, each character was represented as a predefined pixel pattern stored in a lookup table. Thus, along with the basic ‘datawrt‘ and ‘cmdwrt‘ subroutines, another subroutine, ‘display_char‘, was designed to display characters whose hex codes were stored in the lookup table.

To confirm successful text rendering, the first text displayed was ”Microcontroller-Project.com.” Additionally, all the letters were displayed one by one to ensure that every basic element required for further use was functioning correctly.

After multiple tests and refinements, the first successful display of text on the GLCD was achieved, confirming that the microcontroller was properly interfacing with the display. This milestone validated the correct functioning of pixel addressing, data writing, and character rendering, setting the foundation for further graphical developments.

Since we were using a GLCD, we aimed to utilize its full capabilities. Therefore, we also attempted to display images on it. After successfully displaying images, we were confident that we had fully mastered the GLCD and could proceed with further developments.

The images below show the results and testing conducted in Proteus.

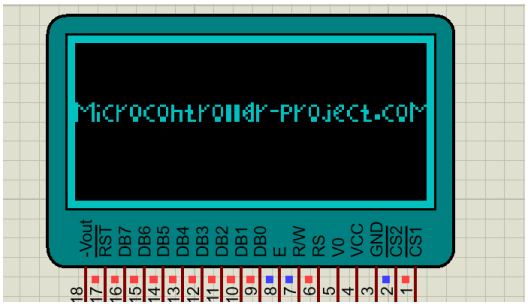


Figure 24: First text display on GLCD

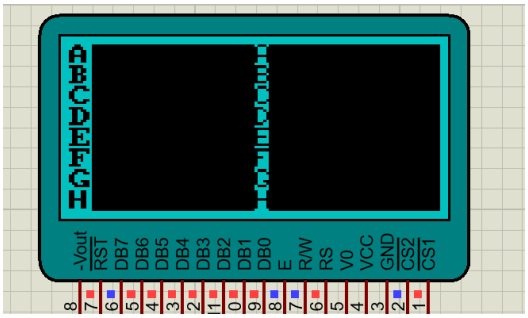


Figure 25: Character display verification

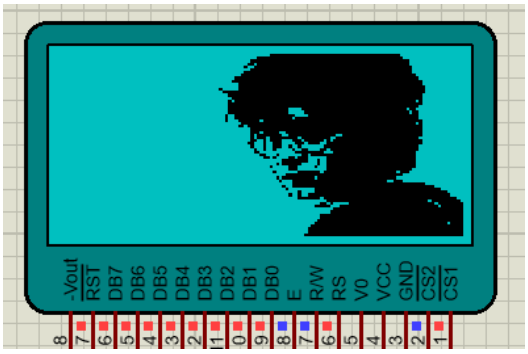


Figure 26: Image displayed on GLCD

Figure 27: G-Yantra logo

3.2 Controlling a Single Element

Once basic text rendering was achieved, the next step involved controlling a single graphical element. This step was crucial, as it required integrating key inputs with graphical display, which is essential for our project. In this process, a small graphical square block was programmed to be controlled using keys, simulating movement in response to user input.

To ensure precise functionality, the movement logic was carefully implemented to allow controlled navigation of the block across the screen. Each keystroke was mapped to a specific direction, modifying the block's position accordingly. This step played a significant role in understanding how dynamic graphics could be manipulated on the GLCD.

Again, Proteus simulations were used to verify each stage. Multiple test cases were conducted to validate smooth and responsive movement. Screenshots were captured as proof of successful execution, demonstrating that keypresses were effectively translated into graphical movements on the display.

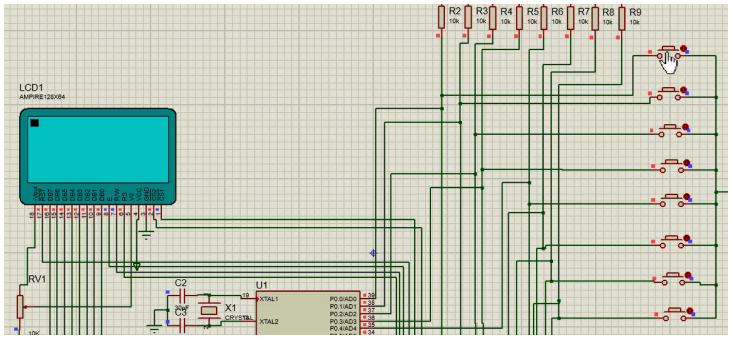


Figure 28: Initial square block display

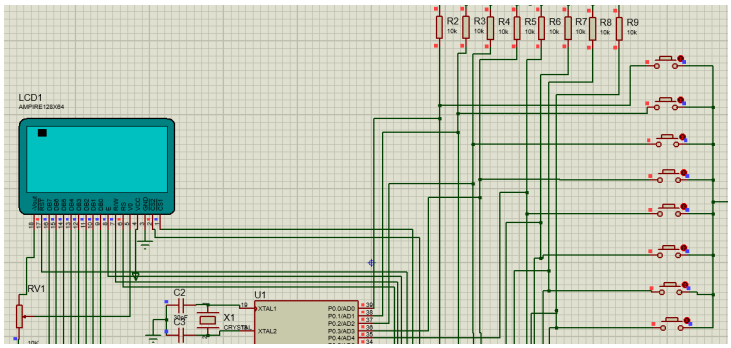


Figure 29: Square block moved right upon key press

3.3 Developing Basic Graphics and Interaction

After gaining control over individual pixels, the next focus was on creating structured graphical elements for both games, as well as GUI components such as the snake body, food element, cursor, grids, and numeric digits (0 to 9). This phase was essential for developing the core visual and interactive elements of the games.

One of the major challenges in this stage was designing pixel-based graphics within the constraints of the 8x8 block as we have virtually divided everything into block of 8x8. Unlike modern displays, where advanced libraries handle rendering, here, every graphical element had to be manually mapped to pixels. This required careful bitwise operations to turn on and off specific pixels in memory.

Developing graphics was a time-consuming process, requiring multiple iterations and testing. Proteus simulations played a crucial role in debugging and refining the visuals, ensuring that each element appeared correctly and responded as intended.

Below are the basic graphics of both games:

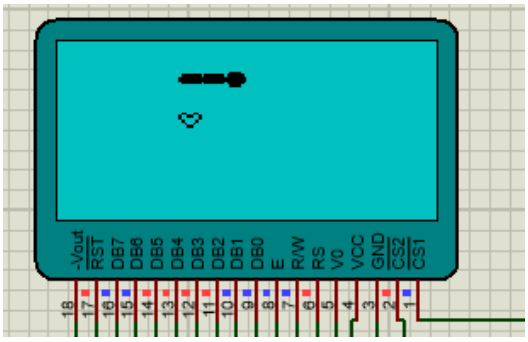


Figure 30: Snake Game Graphics

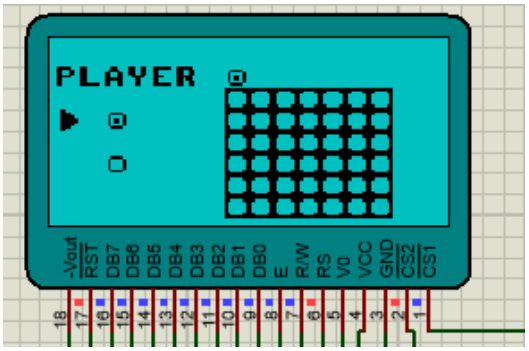


Figure 31: 4-in-a row Graphics

3.4 Simulation Testing in Proteus

Before transitioning to actual hardware implementation, all features were rigorously tested in Proteus. This software simulation approach was essential in verifying the correctness of code, ensuring proper communication between the microcontroller and the GLCD, and refining graphical functions before deploying them on physical components.

Once the fundamental interfacing was validated, testing proceeded to advanced graphical elements and game mechanics, including:

- Verifying the rendering of game elements such as the snake body, food element, and 4-in-a-row grid on the GLCD.
- Testing user interaction by simulating push button inputs to control movement and game actions.
- Ensuring smooth gameplay by evaluating real-time responses, including the snake's movement, food consumption, and coin placement.
- Debugging the graphical user interface (GUI), including cursor movement and selection options.
- Observing frame updates to detect flickering or delays in Proteus.

Each step, from simple text display to dynamic graphical updates, was validated through software simulation. Screenshots were taken at each milestone to document the progression, serving as proof of concept and aiding in debugging. Proteus allowed real-time visualization of how each function executed, helping to detect errors before moving to hardware.

By rigorously testing in Proteus before moving to hardware, the development process became

more efficient and structured. This ensured that when features were finally implemented on physical components, the likelihood of encountering major issues was significantly reduced.

3.5 Transitioning to Hardware

After successful software-based validation, the implementation was gradually transferred to actual hardware. This transition was carried out in a structured, step-by-step manner to ensure that each component functioned correctly before integrating the complete system. The testing process began with a breadboard setup and a development board, allowing for easy modifications and debugging.

In transitioning our project from software simulation to physical hardware implementation, we employed the Silicon TechnoLabs ATMEL 8051 Development Board. This development board facilitated a structured and efficient migration, ensuring each component functioned correctly before integrating into the complete system.

The Follwoing are some Features of the Silicon TechnoLabs ATMEL 8051 Development Board:

- **Microcontroller Compatibility:** Supports AT89S51/52 and P89V51RD2 microcontrollers, providing flexibility for various project requirements.
- **I/O Accessibility:** All pins of the microcontroller are easily accessible via headers, simplifying the process of connecting peripherals and external modules.
- **Power Supply Options:** Offers multiple power input options, including USB and external adapters, ensuring adaptability to different working environments.
- **Communication Interface:** Features an onboard MAX232 IC, facilitating seamless RS232 serial communication.
- **Programming Support:** Compatible with AT89SXX and P89V51R series microcontrollers, providing versatility for various application.
- **Additional Peripherals:** Offers connectors for external modules and peripherals, enhancing the board's adaptability for diverse project requirements.

The Following are the Images Associated with the Development Board Used:

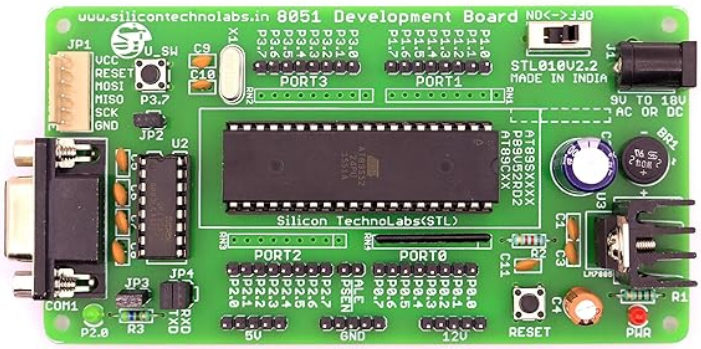


Figure 32: Top view of the Silicon TechnoLabs ATMEL 8051 Development Board

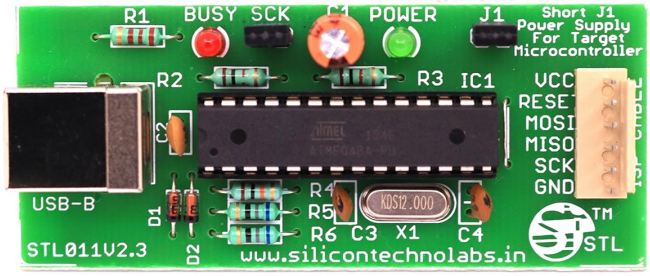


Figure 33: Top view of the Silicon TechnoLabs ATMEL Communication Module

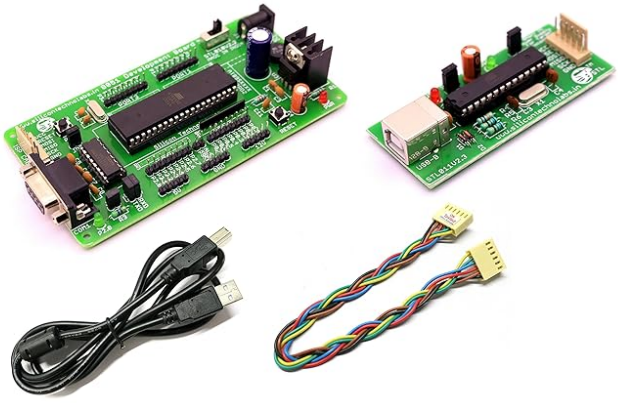


Figure 34: Side View of The Entire Kit

The process began with the basic hardware verification, where individual components were tested separately before being combined. Debugging tools such as LED indicators identify discrepancies between the simulated and real-world behavior. Each stage of hardware testing is documented with images to showcase progress.

3.5.1 Step 1: Displaying Snake Graphics on GLCD

The first step in the hardware implementation was rendering the snake graphics on the GLCD. This phase focused on verifying the accuracy of pixel rendering and ensuring that the display updates correctly when the snake moves. Initially, a static representation of the snake was displayed to confirm pixel mapping, followed by incremental movement logic to simulate motion.

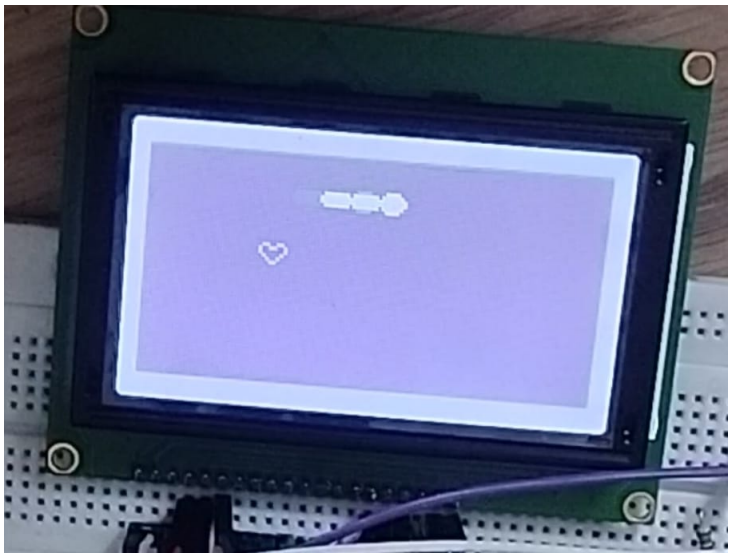


Figure 35: Snake graphics successfully displayed on GLCD

3.5.2 Step 2: Implementing Game Along with Keyboard Controls

Once the snake's graphical elements were displayed correctly, the next step was to integrate real-time movement through keyboard inputs. The microcontroller was programmed to detect directional keypresses and update the snake's position accordingly. This required efficient interrupt-based handling to ensure responsive controls.

Key aspects tested in this phase included:

- Smooth movement of the snake in response to keyboard inputs.
- Correct detection of food consumption, ensuring the snake's length extends automatically

and is displayed properly on the GLCD.

- Accurate collision detection with screen boundaries or the snake's own body, along with proper score updates.

Testing in hardware involved observing real-time gameplay, ensuring no delay in movement, and verifying that the microcontroller correctly processed inputs from the keyboard interface.

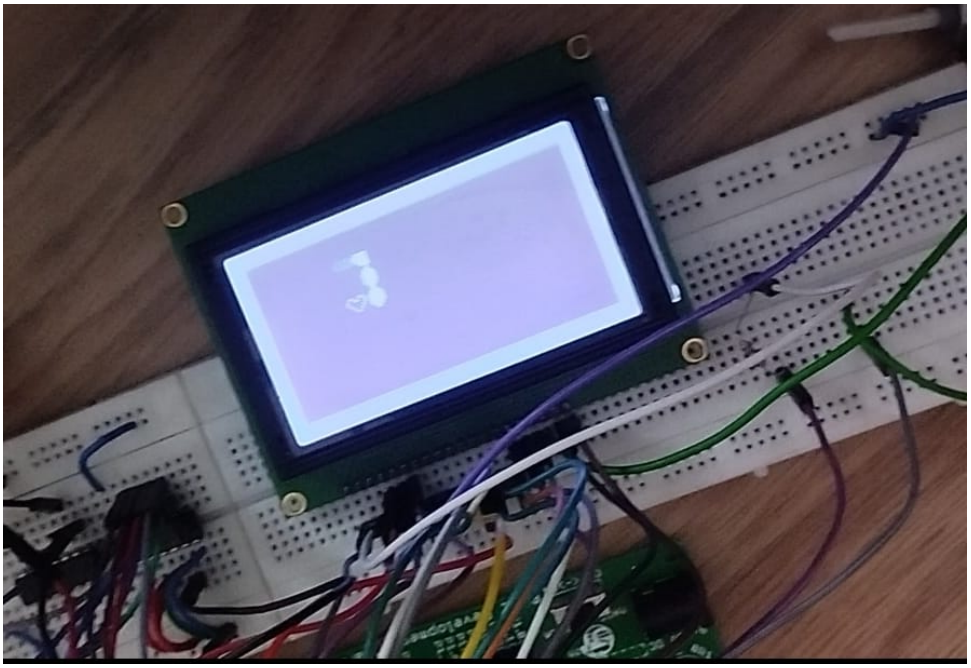


Figure 36: Snake movement controlled via keyboard input

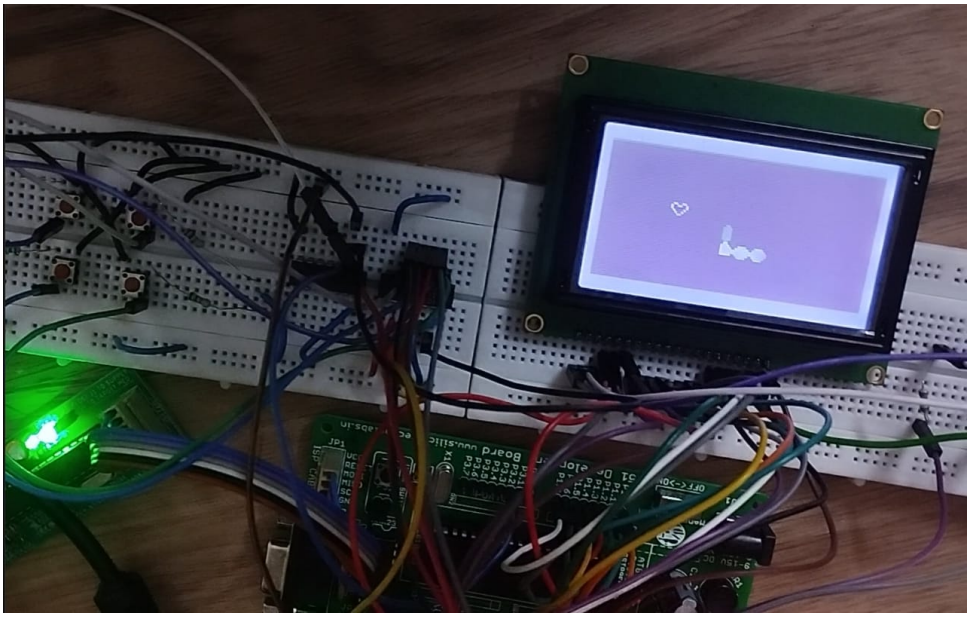


Figure 37: Breadboard,Keyboard and GLCD setup

3.5.3 Step 3: Graphical User Interface with Keyboard

With the snake game mechanics successfully implemented, the next step involved designing the Graphical User Interface (GUI). The GUI was responsible for navigating between different game options, selecting modes, and displaying messages. The keyboard interface was used to control cursor movement within the menu.

Key features of the GUI included:

- An intro screen and game selection menu.
- Cursor navigation using arrow keys and selection via a dedicated key.
- Proper visual feedback to indicate the currently selected option.

Figure 38: Game selection menu displayed on GLCD

Figure 39: Game successfully selected and waiting for user to press start

3.6 Final Testing and Optimization

The complete system, including graphical elements and user inputs, was rigorously tested in real-time conditions. Various test scenarios, such as rapid button presses, simultaneous inputs, and prolonged operation, were evaluated to ensure stability and responsiveness.

At this stage, all functionalities were transitioned from the breadboard and development board to the final PCB implementation. The PCB design was carefully verified to ensure proper routing of signals, minimal noise interference, and reliable power distribution.

This structured approach, progressing from software simulation to hardware prototyping and final PCB integration, ensured a well-refined system with documented improvements at every stage.

The Following are the Images of The Various PCB's that are Associated with our system:

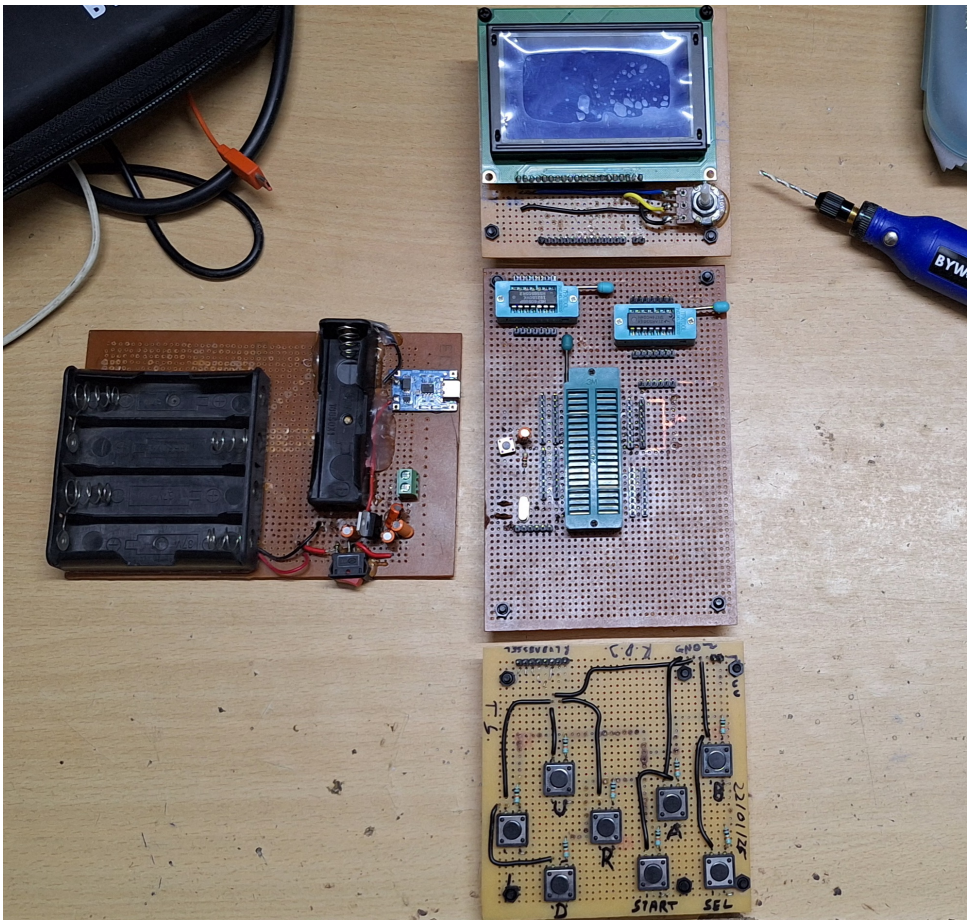


Figure 40: G-Yantra PCB Components Top View

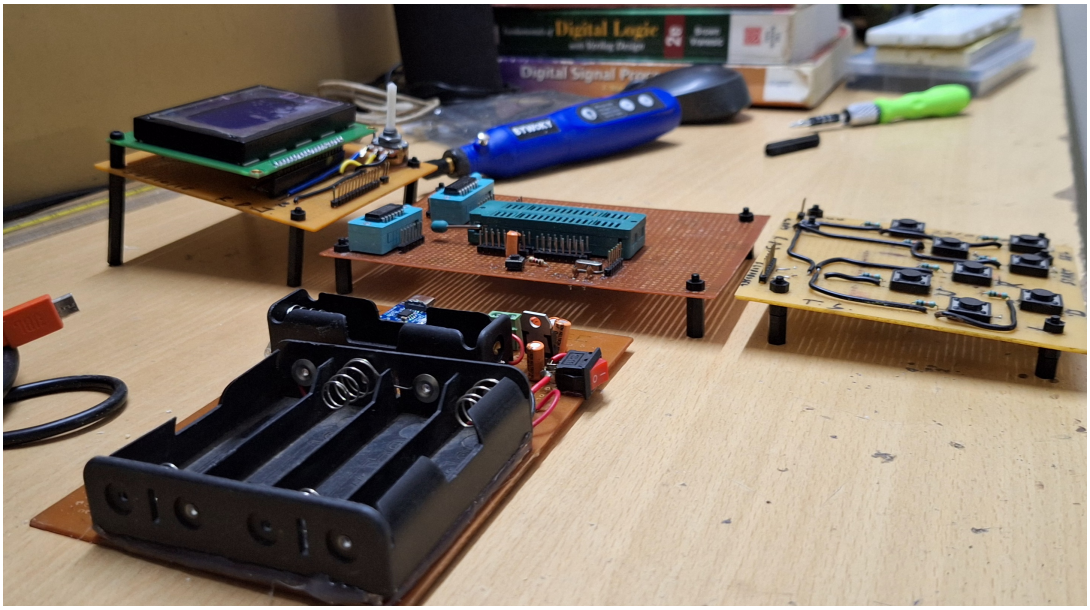


Figure 41: G-Yantra PCB Components Side View

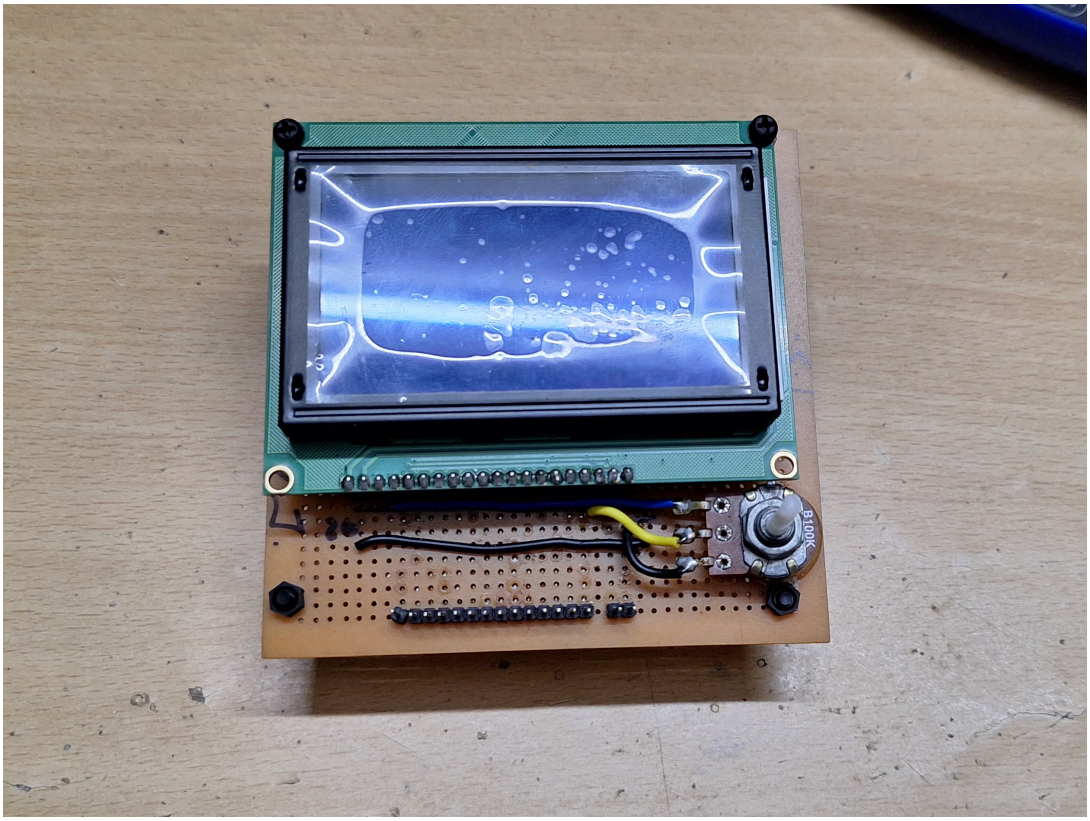


Figure 42: GCLD Module PCB Top View

Figure 43: GCLD Module PCB Side View

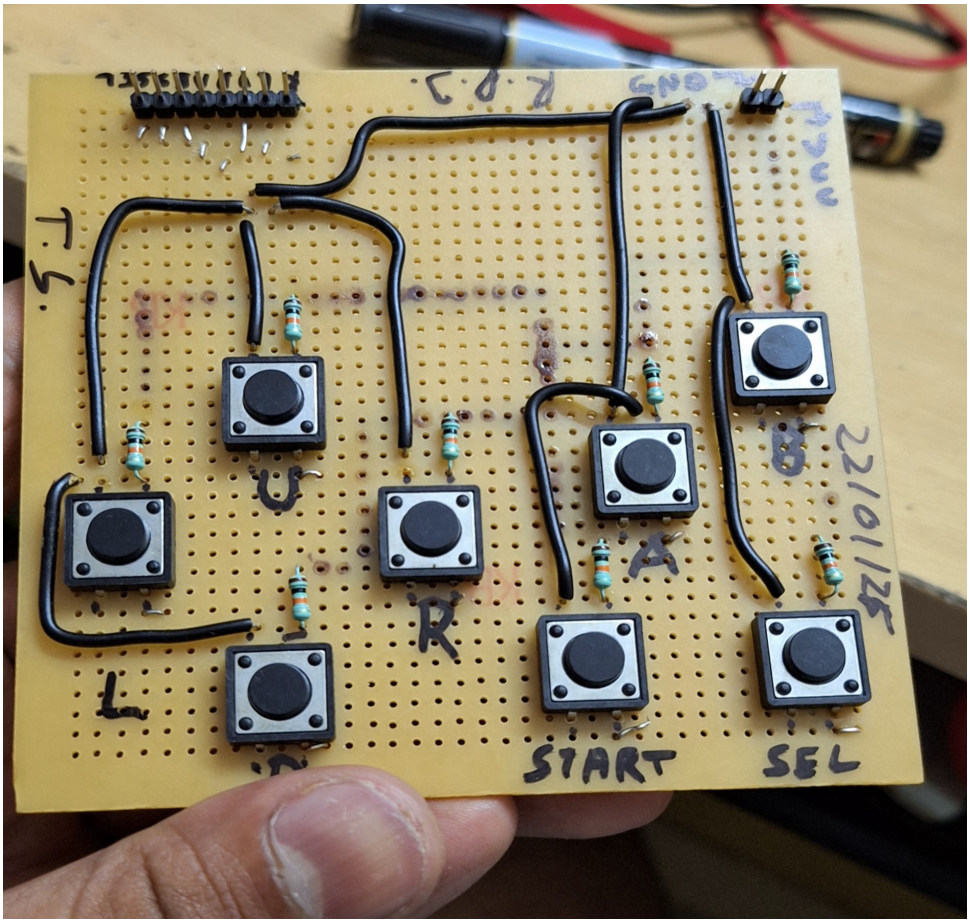


Figure 44: Key-Board PCB Top View

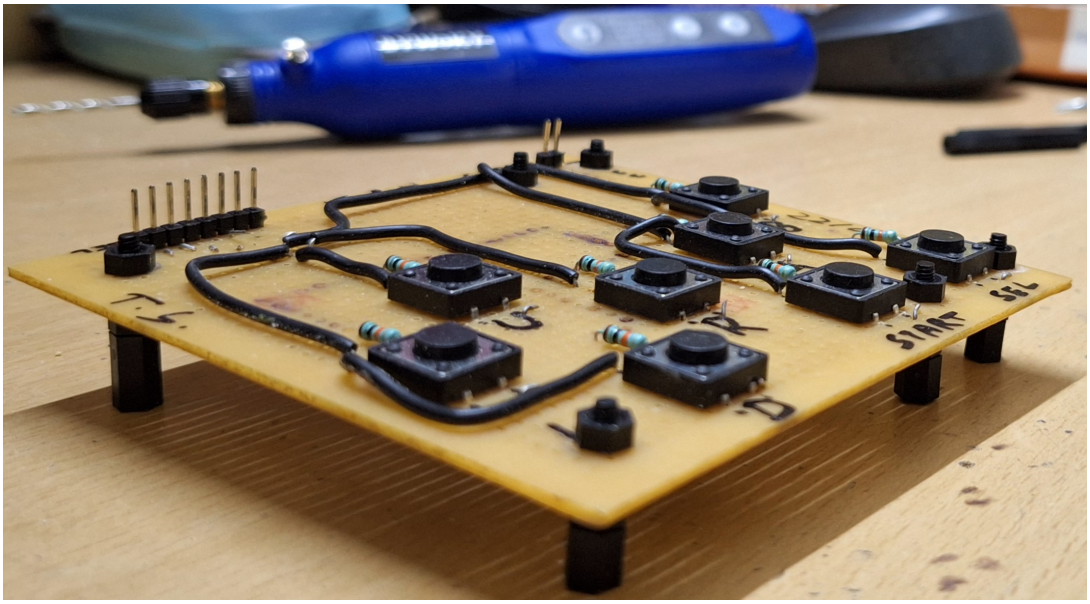


Figure 45: Key-Board PCB Side View

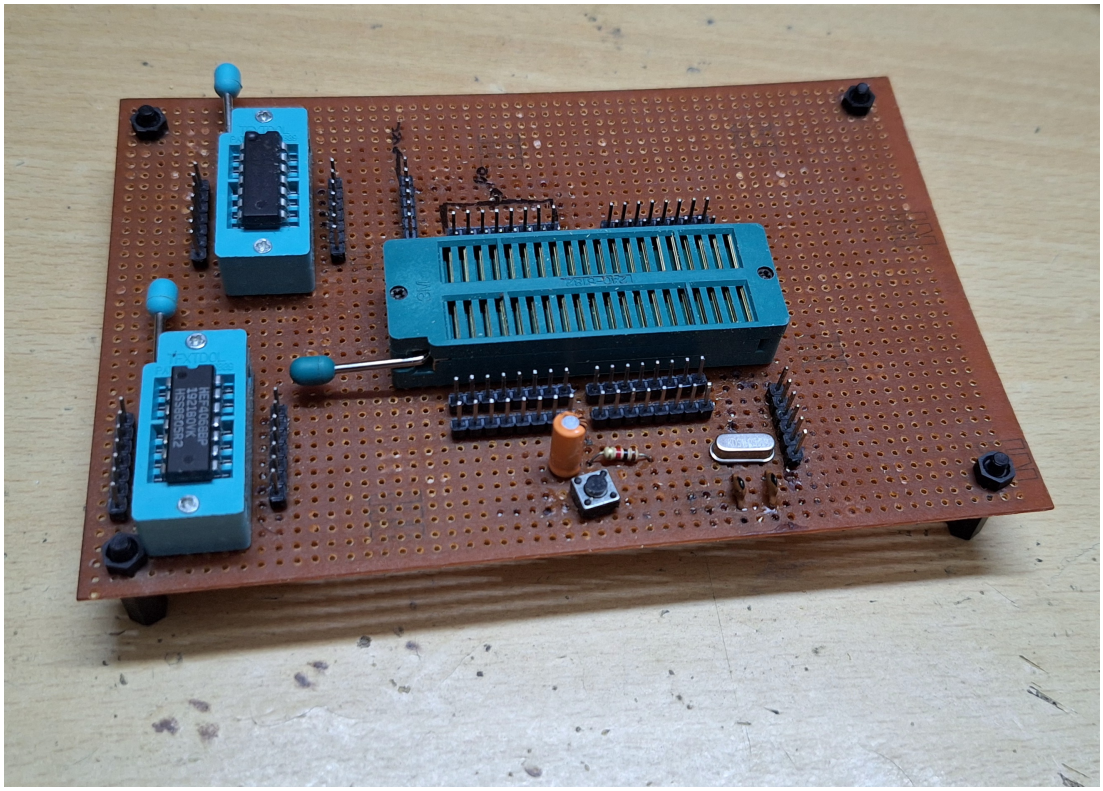


Figure 46: IC Holder PCB Top View

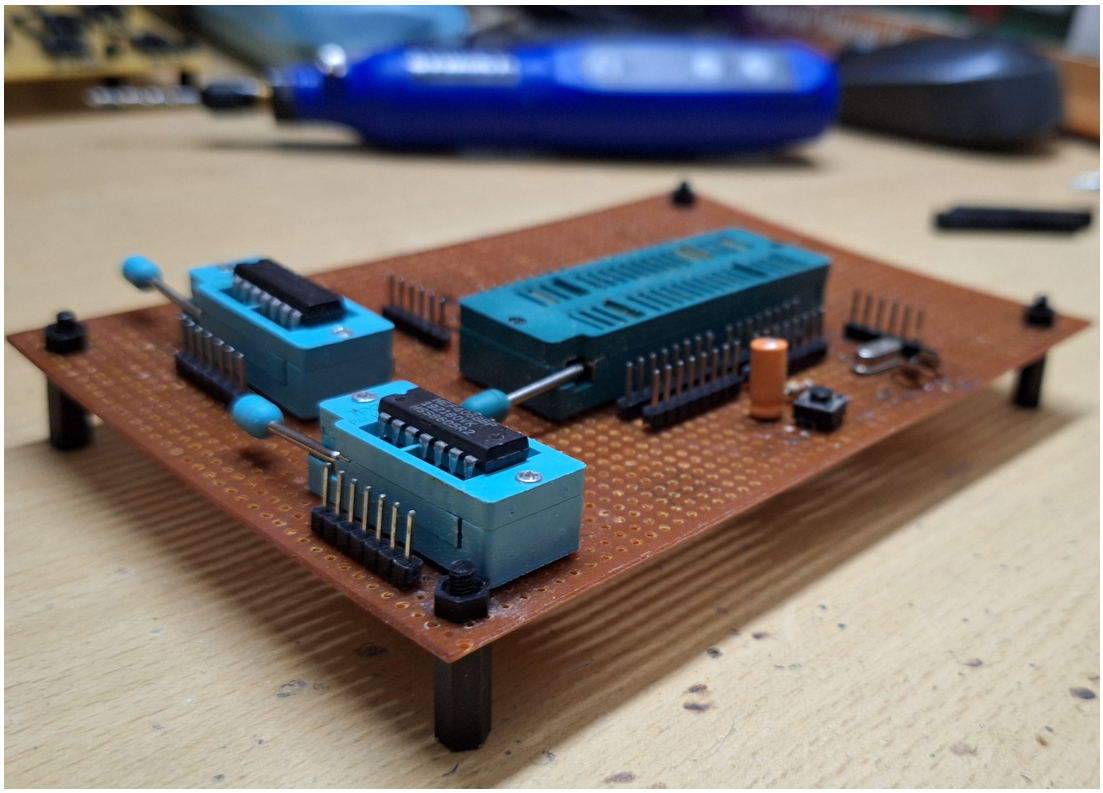


Figure 47: IC Holder PCB Side View

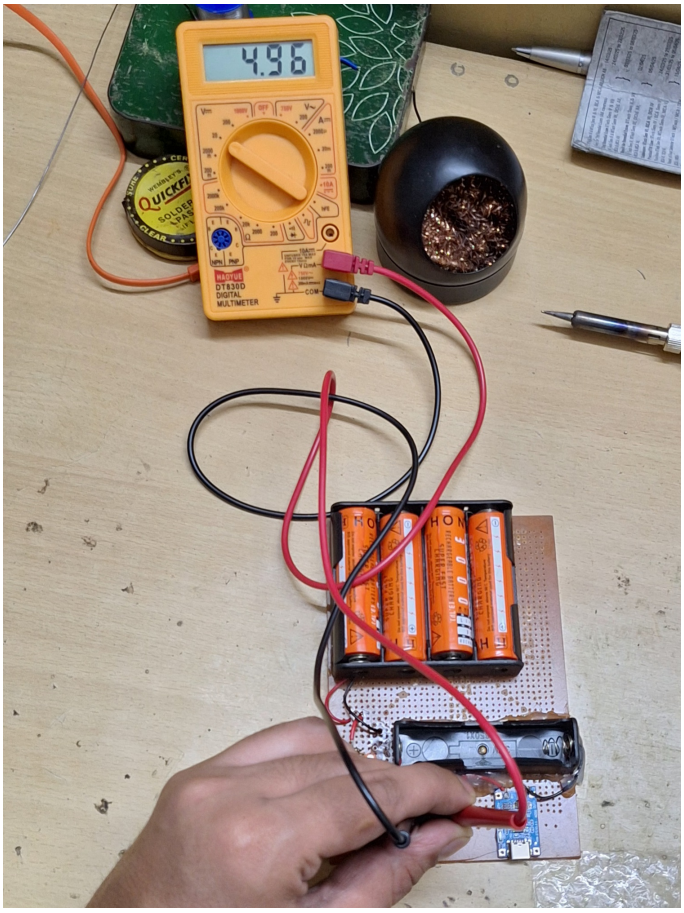


Figure 48: Battery Module PCB Providing Regulated Output

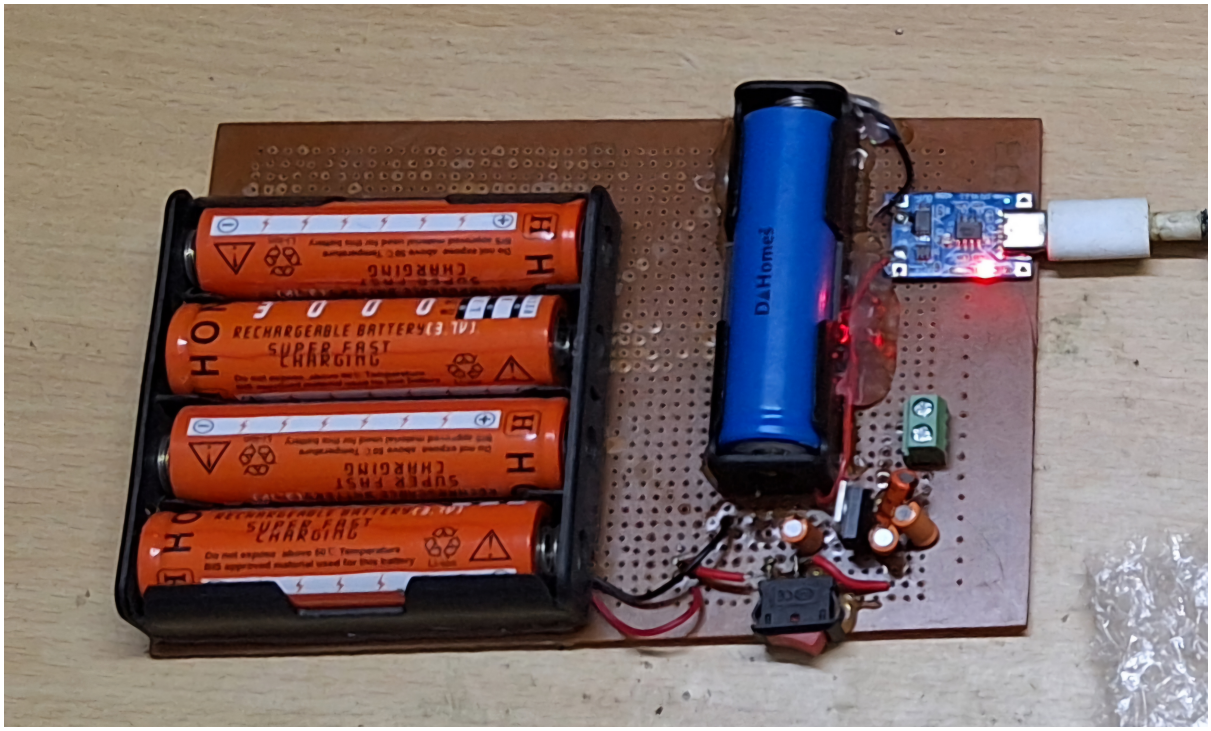


Figure 49: Battery Module PCB With Recharging Module in Action

4 Results and Discussion

- All graphical elements, including the snake, food, game grids, and cursor, were displayed correctly on the 128x64 GLCD.
- The Snake Game exhibited smooth movement, real-time collision detection leading to game over and accurate score updates. This indicated that all the software routines are in order in addition to proper functioning of the hardware.
- The highest score achieved during testing was 18, demonstrating stable and responsive gameplay.
- The 4-in-a-Row Game had its core logic successfully validated through sample attempts, though full gameplay testing on the GLCD remains pending due to time constraints.
- Now a game obviously requires quick responses to player's quick actions through key pressing. So in this our game rapid push button inputs were accurately detected, ensuring instantaneous movement of the snake without any noticeable delay.
- In the GUI, the cursor movement was proper, and the user's desired game was correctly selected.

The Following Are Some Proof of Results:

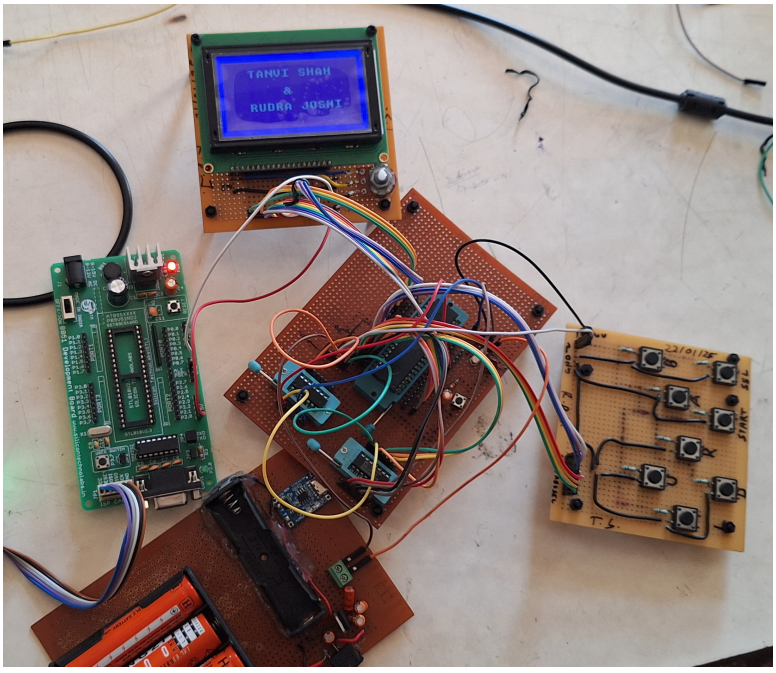


Figure 50: Output Of Whole Hardware-1

Figure 51: Working snake game

Figure 52: Working GUI

5 Conclusion

The project successfully achieved its objective of developing a functional gaming system on an 8051-based microcontroller with a graphical interface. The structured approach, from simulation in Proteus to hardware implementation, ensured a smooth transition and efficient debugging.

This work highlights the potential for embedded gaming applications on limited hardware resources, such as a small-sized LCD screen with limited graphics.While the current implementation focused on fundamental gameplay mechanics, future improvements could introduce enhanced graphics, animations, and additional features to enrich the user experience.

6 Future IoT Integration with ESP-32

6.1 Introduction

ESP-32 is a low-power, Wi-Fi-and-Bluetooth-enabled microcontroller widely used for IoT applications. Although ESP-32 has not yet been implemented in this project, it is considered for future integration to enable cloud-based score tracking.

6.2 Proposed Hardware Components

The following components will be required for integrating ESP-32 with the gaming system in future developments:

- ESP-32 Development Board – Main microcontroller for IoT communication.
- Power Supply Module – Provides regulated 3.3V for ESP-32.
- Communication Interface – SPI, I2C, or UART for interfacing with the gaming system.

6.3 Proposed Software Architecture

In the future, the ESP-32 firmware will be designed to handle the following tasks:

1. Wi-Fi Configuration – Connects ESP-32 to a wireless network.

2. HTTP or MQTT Communication – Sends and receives data from a cloud server.
3. Score Data Processing – Reads the final game score from the 8051 microcontroller and transmits it to the cloud.
4. Game State Updates – Provides cloud-based monitoring of game performance.

6.4 Proposed Implementation

The ESP-32 will be programmed using either the Arduino IDE or ESP-IDF, utilizing the following key libraries:

- `WiFi.h` – Manages Wi-Fi connections.
- `HTTPClient.h` – Handles REST API requests for cloud communication.

6.5 How It Would Work

6.5.1 ESP32 Monitors 8051

Once integrated, the ESP-32 will detect when the 8051 finishes the game and displays the score by monitoring a GPIO pin. The 8051 will set this pin HIGH when the score is available.

6.5.2 Score Retrieval from 8051

- If the 8051 sends the score using an 8-bit parallel output, the ESP-32 will read it from GPIOs 18-25.
- If serial communication (UART) is used, the ESP-32 will be programmed to receive the score accordingly.

6.5.3 Upload to Firebase

ESP-32 will send the retrieved score as a JSON request to Google Firebase via HTTP, allowing cloud-based storage and remote access.

ESP32 Pin	8051 Pin	Function
GPIO4	P3.1	Score Ready Signal (HIGH when score is available)
GPIO18-25	Available P2.x Pins	8-bit score data from 8051 (if using parallel output)
GND	GND	Common Ground

Table 3: Proposed ESP32 to 8051 Connections

6.5.4 Proposed Connections Between ESP32 and 8051

6.5.5 Algorithm for ESP-32 IoT Integration and Score Upload

The Following is Algorithm that has been thought of for ESP-32 Integration:

Algorithm 1 ESP-32 Score Retrieval and Upload Process

Initialization:
    Configure Wi-Fi on ESP-32
    Initialize GPIO pins and UART (if used)

Monitor: Wait for Score Ready Signal on GPIO4
if Score Ready Signal is HIGH **then**
    if Using Parallel Communication **then**
        Read 8-bit score from GPIO18–GPIO25
    else
        Read score via UART
    end if
    Format the score as JSON
    Send HTTP POST request to Firebase with JSON payload
end if
Loop: Return to monitoring for the next score ready signal

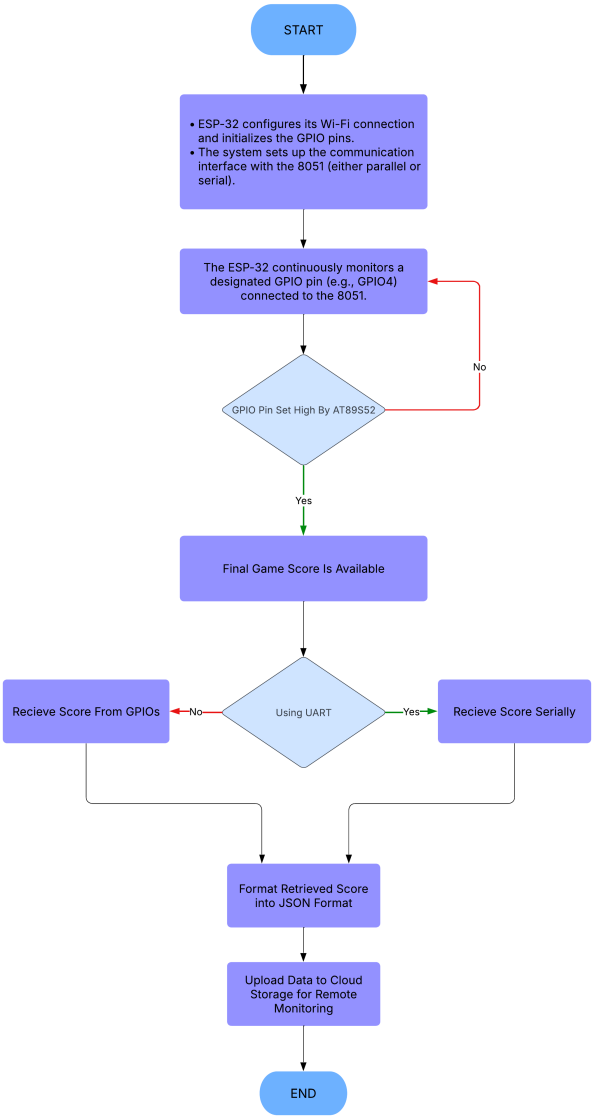


Figure 53: Block Diagram of SCORE_CHECK

7 Comparison of Our Project with Historical Gaming Consoles

Embarking on our project to develop games on the **AT89S52 microcontroller** with a **JHD12864E (128x64) GLCD** has provided a profound understanding of the evolution and technical intricacies of gaming hardware. By comparing our project with historical gaming consoles, we can appreciate the advancements and challenges in the gaming industry. Below is a comparative analysis.

7.1 GLCD Specifications:

Our project utilizes the JHD12864E 128×64 monochrome graphical LCD. Key specifications:

- **Resolution:** 128×64 pixels
- **Display Type:** STN, Blue-White, Positive Transflective
- **Viewing Angle:** 6 o'clock
- **Operating Voltage:** Single power supply (+5V)
- **Interface:** 8-bit parallel communication
- **Driving Method:** 1/64 Duty, 1/9 Bias
- **Module Size:** 93.0 mm × 70.0 mm × 12.7 mm
- **Weight:** About 50 g

7.2 Insights Gained from Our Project

Resource Constraints: Working with limited memory (256 Bytes RAM) and processing power (8-bit CPU) highlighted the importance of efficient coding practices and optimization techniques.

Hardware-Software Integration: Developing games on the AT89S52 required a deep understanding of hardware capabilities, akin to early console developers who maximized performance within hardware limitations.

Evolution of Graphics: Implementing games on a monochrome 128×64 GLCD provided perspective on the advancements from simple graphics to the high-definition visuals in modern consoles.

Game Media Transition: Our use of internal memory contrasts with the evolution from cartridges to optical media, reflecting on how storage solutions have influenced game design and distribution.

Technical Challenges: Addressing challenges such as limited input methods and display constraints paralleled the hurdles faced by pioneers in the gaming industry.

The Following Table Provides a Comprehensive Summary of Comparision of our Project with Older Arcade Gaming Consoles form which it was originally inspired:

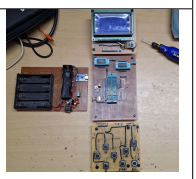
Console	CPU	Memory	Graphics	Game Media	Release Year	Image
Our Project	8-bit, 11.0592 MHz	256 Bytes RAM	128×64 Monochrome GLCD	Internal	2025	
Magnavox Odyssey	No CPU (Discrete)	N/A	Overlay-based	Custom PCBs	1972	
Atari 2600	8-bit, 1.19 MHz	128 Bytes RAM	160×192, 128 colors	Cartridges	1977	
NES	8-bit, 1.79 MHz	2 KB RAM	256×240, 48 colors	Cartridges	1983	
Sega Genesis	16-bit, 7.6 MHz	64 KB RAM	320×224, 512 colors	Cartridges	1988	
Sony PlaySta-tion	32-bit, 33.9 MHz	2 MB RAM	640×480, 16.7M colors	CDs	1994	
Microsoft Xbox	32-bit, 733 MHz	64 MB RAM	1920×1080, 16.7M colors	DVDs	2001	

Table 4: Comparison of Our Project with Historical Gaming Consoles

By engaging in this project, we have not only developed technical skills but also gained a historical perspective on the gaming industry's evolution, enriching our appreciation for both past and present gaming technologies

8 Tools and Technologies Used

The following tools and technologies were used in the development of this project:

1. **Keil uVision** – For 8051 Assembly code development (2k Limit For Hex File Generation).
2. **MCU 8051 IDE** – For 8051 Assembly code development (No Size Limit for Hex File Generation Encountered until now).
3. **Proteus** – For circuit simulation and testing.
4. **ProgISP** – For burning/downloading Hex file Into Actual AT89S52 Chip.
5. **Arduino IDE** – For programming and uploading code to the ESP-32 (For Future IoT Implementations).
6. **ChatGPT** – For understanding and refining fundamental algorithmic knowledge about gaming constructs in Assmebly **(Not Used for Generating Programs...! All the codes are typed by developers own hands)**.

Bibliography

[1] Atmel. *8051 Microcontroller Datasheet*. Available online.

[2] KS0108. *Graphics LCD Datasheet*. Available online.

[3] Manish K. Patel. *The 8051 Microcontroller Based Embedded Systems*. McGraw Hill Education.

[4] Muhammad A. Mazidi and Janice G. Mazidi. *The 8051 Microcontroller and Embedded Systems*. 2nd Edition, Pearson Education.

[5] Rajib Mall. *Real-Time Systems Theory and Practice*. 2nd Edition, Pearson Education. (Will be utilized at later stages.)

[6] "The Insane Engineering of Gameboy" (Initial Inspiration).

[7] "Nokia 3310 Snake on a Microcontroller" (For clearing fundamental logic-related concepts).

[8] NesHacker (For understanding the working and architecture of Nintendo Gameboy on which the project is based).

[9] Gameboy Memory Analysis.

[10] "Keypad and LCD with 8051 Microcontroller," EngineersGarage.

Appendix A

Annexure: Relevant Codes, User Manual, & Datasheet

G-Yantra Assembly Code

```
;----------------------------------------
; G-Yantra 8051 Based Gaming Replica    |
; Has come to life because of collective|
;Efforts of The Following Authors               |
;Tanvi Shah & Rudra Joshi                              |
;----------------------------------------

;P1->D0-D7
;P2.0->RS (Reg. Select)
;P2.1->R/W'
;P2.2->E
;P2.3->RST (RESET)
;P2.4->CS1
;P2.5->CS2

;the following are steps that we'll follow:
;1. Initialize GLCD
;2. Select GLCD Half
;3. Select Page
;4. Display Text

;For Keyboard :
;p0 is used to Interface Buttons for 8x1 Board

;---The Purpose of this GLCD Subroutines is as follows---
;to divide the screen into grid of 128 pixel grid which
;contains 8x8 pixel array which can be controlled individually
;also on this display move a user drawn element as per the user inputs

;also the follwing register banks have been used
;for various operations:
; reg-0->LCD Operations
; reg-1->Keyboard Operations
; reg-2->Snake Game Operations/4 in a Row Based Operations
; reg-3->Random Use

;Memory Map Made for Made This Version Which
;is where We Would Like To Draw Your Attention To

;1. Code In Code Memory has been segemnted as Follows:

;-> form 0010h to 0054h main labeled code is written
```

```
;-> from 0064h to 01A3h GLCD and Delay Related Routines are
;written

;-> from 00250h to 02C4h Keyboard ISR is Written
;for Snake Game and GUI

;-> from 002d0h to 0341h Keyboard ISR is Written
;for 4 in a Row

;-> from 0343h to 079bh Look-Up Table for
;Various Things Lies There

;-> from 079dh to 0CFEh Main Gameboy Related Routines
;and Snake Games Are Written

;-> from 0d00h to 0FDAh 4 in a Row Game Related Routines Are Written

;2. for GUI Related Purposes bit-wise memory location
;08h to 0ch have been used appropriately

;3. Keyboard ISR has been made more generalised as it should be !!!

;however ISR for 4 four In A Row Game is Seprate
;from Keyboard ISR for Snake Game + GUI

;AND THAT'S IT ENJOY READING ALL THESE CODES
;TRYING TO UNDERSTAND WHAT THEY DO...! :)

org 0000h

        sjmp main

org 0003h

        jb 0ch,keyboard_normal ;If 0ch bit is Set then Keyboard
        ;of Snake Game + GUI Is Executed

        jnb 09h,key_board_isr_4IR ;If 0ch bit is not set and 09h
        ;is also clr then 4 in a row ISR is Executed

        keyboard_normal:
        ajmp key_board_isr

        key_board_isr_4IR:
        ajmp keyboard_isr_4_in_a_row

org 0010h
```

```
        main: ;main logic inception starts here

        mov p0,#0ffh ;set p0 as input port

        setb p2.3 ;set rest pin to 1 i.e. inactive mode for GLCD

        mov sp,#5ch ;move sp to ram loaction 5ch

        mov tmod,#02h ;SET Timer-0 to 8-Bit Auto Reload
        mov th0,#00h
        setb tr0

        orl tcon,#01h ;Set External Interrupt-0 Bit as Edge Trigerred

        mov ie,#00h ;Disbale all Interrupts till one them
        ;is enabled as and when required

        lcall delay ;some delay is introduced purposefully
        lcall delay     ;to give GLCD some time to initialize properly
        lcall delay
        lcall delay
        lcall delay
        lcall delay

        mov a,#3fh ;set z coordinate associated with GLCD

        lcall cmdwrt

        lcall clrscreen ;clr the GLCD Screen
        lcall g_yantra_intro ;Show G-Yantra Intro

        lcall clrscreen
        lcall developers_intro ;Show Developers Intro

        setb 08h ;set bit location 08h
        setb 0ch ;set bit location 0ch

        lcall gui ;call GUI Subroutine

        ljmp game_select ;call game_select subroutine

        stay: sjmp stay


org 0064h ;here lies the codes for GLCD and GUI Related Operations
```

```
cmdwrt: ;to give commands to GLCD

        push psw

        clr psw.3
        clr psw.4

        acall delay
        mov p1,a ;move command in accumulator to p1
        clr p2.0 ;select command register (rs-pin)
        clr p2.1 ;set lcd to write mode (r/w'-pin)
        setb p2.2 ;set enable signal (e-pin)
        acall delay ;call delay subroutine
        clr p2.2 ;give a high to low pulse
        acall delay

        pop psw

ret ;for cmdwrt

datawrt: ;to give data to GLCD

        push psw

        clr psw.3
        clr psw.4

        acall delay
        mov p1,a ;move data in accumulator to p1

        setb p2.0 ;select data register (rs-pin)
        clr p2.1 ;set lcd to write mode (r/w'-pin)
        setb p2.2 ;set enable signal (e-pin)
        acall delay ;call delay subroutine
        clr p2.2 ;give a high to low pulse
        acall delay

        pop psw

ret ;for datawrt

display_char: ;subroutine to display a charcter
;form lookup table r5,r0 of reg bak 0 used

                push psw
                push acc
```

```
                    clr psw.3
                    clr psw.4

                    mov r5,#00h     ;Initialize index for looping
                    ;through the string

                    mov r0,#08

                    back2nxt:

                    mov a,r5;mov cuurent index into a
                    movc a, @a+dptr;Load the character from the string
                    lcall datawrt;Call datawrt to display the character
                    lcall delay;call delay
                    inc r5;increment index to access next character

                    djnz r0,back2nxt

                    pop acc
                    pop psw

        ret ;for display_character

        display_string: ;used to display a string here
        ;we pass dptr value, total characters
        ;and first character location r5 of rb3 used

                  push psw

                  setb psw.3
                  setb psw.4

                  mov r5,a  ; total characters in string

                  nxt_no:
                         lcall display_char
                         lcall nxt_dptr
                         inc b
                         mov a,b
                         lcall choose_coord
                  djnz r5 ,nxt_no

                 pop psw

        ret ;for display_string

        nxt_dptr: ;used in subroutine of display_string only...!
```

```
        ;r4 of rb3
        ;to transverse dptr through different characters of string

                mov r4,#8
                incr:inc dptr
                djnz r4,incr

        ret ;for nxt_dptr

        delay:;1ms delay assuming clk freq 12MHz
        ;r7 and r6 of reg bank 0 used

                        push psw

                        clr psw.3
                        clr psw.4

                        mov r7,#2
                        here2:mov r6,#255
                        here1:djnz r6,here1
                        djnz r7,here2

                        pop psw

        ret ;for delay

        delay1s:;1sec delay generation assuming 12Mhz Clk
        ;r7,r5 and r6 of reg bank 0 used

                mov r7,#02
                here0:
                        mov r6,#250
                here10:
                        mov r5,#250
                here20:
                        nop
                        nop
                djnz r5,here20
                djnz r6,here10
                djnz r7,here0

                pop psw
```

```
ret ;for delay1s

dboun: ;delay subroutine for keypad
; r4,r5 of reg bank 1

        push psw

        setb psw.3 ;select reg-1 for Keyboard Opreations
        clr psw.4

        mov r4,#10d
        dloop2:mov r5,#250d
        dloop1:nop
        nop
        djnz r5,dloop1
        djnz r4,dloop2

        pop psw

ret ;for dboun

set_column: ;selects a particular column form where
;to write data for a given selected page
;r2 of reg b3

        push b

        mov b,#08d ;to set the starting column address
        ;to the one meant by pixel grid value

        mul ab
        mov b,a ;copy a in b for book-keeping

        subb a,#40h ;check if the number in accumulator is
        ;greater than 64 in decimal to take decision

        jnc right_half ;jump to right half
        ;if number is >=64 in decimal

        ;logic for selecting on left half of screen

        mov 1ah,#00h ; r2 of reg-bank-3

        clr p2.4 ;set cs1 to select first half of glcd
        setb p2.5 ;(actullay inverted logic is used
        ;for proteus simulation hence a bit change
```

```
            ;in these and rst instructions is seen)

            mov a,b ;reload a with original number
            add a,#40h ;add the number plus the 40h which is
            ;command for slecting 0th column in glcd

            acall cmdwrt;call command function to
            ;select a particular column

            sjmp column_set

            right_half: ;logic for right of screen

                    mov 1ah,#01h ; r2 of reg-bank-3

                    setb p2.4 ;selecting right half of screen
                    clr p2.5

                    add a,#40h ;since for right half values
                    ;we'll add to 40h

                    acall cmdwrt ;call command function
                    ;to select a particular column

    column_set:

            pop b
            clr c

    ret ;for set_column

    set_pg_cntrl: ;Selects one of 8 vertical pages,
    ;each representing 8 rows of pixels.

            push b

            mov b,a ;copy a in reg-b temporarily

            mov a,1ah

            jnz rhalf

            clr p2.4 ;select left half
            setb p2.5

            sjmp done
```

```
        rhalf: ;if carry isn't 1 set screen to left half

        setb p2.4
        clr p2.5

        done:mov a,b
        add a,#0b8h ;add numbers form 0 to 7 to b8h to select
        ;any pg out of the available 8 pgs

        acall cmdwrt

        pop b

ret ;for set_pg_cntrl

clrscreen: ;r4,r3,r2,r1 of reg bank 0 used

        push psw

        clr psw.3
        clr psw.4

        clr p2.4 ;cs1=0
        clr p2.5;cs2=0

        mov r4,#0b8h
        mov r3,#8

        mov a,#40h ;first col
        lcall cmdwrt
        lcall delay
        mov a,#0c0h ;z
        lcall cmdwrt
        lcall delay
        mov dptr,#clear

        ag:     mov a,r4; initially at first page
                lcall cmdwrt
                lcall delay
                mov r2,#8
                back1:lcall display_char
                djnz r2,back1
                inc r4
        djnz r3,ag

        pop psw
```

```
        ret ;for clear screen

        choose_coord: ;this subroutine sets the cursor value
        ;to the coordinates contained in reg-a
        ;r7 of rb3

                push psw
                push acc
                push 1fh

                setb psw.3
                setb psw.4

                mov r7,a ;a will contain coordinates which
                ;we would like to set as page and column

                anl a,#0fh
                lcall set_column

                mov a,r7 ;copy the coordinates saved
                ;temporarily back to reg-a

                anl a,#0f0h
                swap a
                lcall set_pg_cntrl

                pop 1fh
                pop acc
                pop psw

        ret ;for choose_coord

org 250h ;keyboard isr starts here

        key_board_isr: ;r0,r1 of reg bank 1

                push psw
                push acc
                setb psw.3 ;select reg-1 for Keyboard Opreations
                clr psw.4

                identify:lcall dboun ;now the program serves to check
                mov a,p0 ;which key is pressed

                mov r0,#00h
                mov r1,#08h
```

```
again: rrc  a ;key indentification logic starts here
jc next_key
sjmp found

next_key: inc r0
djnz r1,again
sjmp returnback

found:
mov a,r0; reg where key code is stored
mov 13h,a; 13h (reg-3 of reg-bank-2) stores
;the direction coordinates

jb 08h,gui_funcs ;check bitwise loaction if it is set

jb 0ah,check_start ;check if start button is pressed

sjmp returnback

gui_funcs:

cjne a,#02h,sel_down ;if a=up then clear down cursor
;and update cursor to new position

        setb 09h;set game select bit
        mov a,#50h
        lcall choose_coord
        mov dptr,#clear
        lcall display_char

        mov a,#30h
        lcall choose_coord
        mov dptr,#cursor
        lcall display_char

        sjmp returnback

sel_down:

cjne a,#03h,sel_button_pressed
;if a=down then clear up cursor
;and update cursor to new position

        clr 09h;clr game select bit
        mov a,#30h
        lcall choose_coord
        mov dptr,#clear
```

```
                        lcall display_char

                        mov a,#50h
                        lcall choose_coord
                        mov dptr,#cursor
                        lcall display_char

                        sjmp returnback

                sel_button_pressed: ;check if select button is
                ;pressed in while branching from GUI subroutine

                cjne a,#07h,returnback

                        clr 08h ;clr selection bit

                sjmp returnback

                check_start: ;check if start button is pressed
                ;in while branching form game_select subroutine

                cjne a,#06h,returnback

                        setb 0bh; set start bit 0bh
                        clr 0ah ;for breaking out of start button
                        ;pressed loop
                        clr 0ch

                returnback:

                pop acc
                pop psw

                reti;for key-board isr

org 2d0h ;KEYBOARD ISR FOR 4 IN A ROW GAME STRATS Here

        keyboard_isr_4_in_a_row:

                push psw
                push acc
                setb psw.3 ;select reg-1 for Keyboard Opreations
                clr psw.4

                identify_4IR:lcall dboun ;now the program
                ;serves to check
```

```
mov a,p0 ;which key is pressed

mov r0,#00h
mov r1,#08h

again_4IR: rrc  a;key indentification logic starts here
jc next_key_4IR
sjmp found_4IR

next_key_4IR: inc r0
djnz r1,again_4IR
sjmp returnback_4IR

found_4IR:
mov a,r0; reg where key code is stored

cjne a,#00h,ch_01 ;if key 1 is pressed means
;move the coin right this logic is for game2
;for same key for game1 has different logic

        mov a,12h ;clear old coin position coin pos
        ;saved in reg r2(12h) of rb2

        lcall choose_coord
        mov dptr,#clear
        lcall display_char

;below 4 lines are to make sure that
;the coin is still in range...
;our range is 17h to 1dh so if it is at
;1dh so next position will be 17h


        mov a,12h
        cjne a,#1dh,goto_nxt_4IR ;if it is at 1dh
        ;location i.e. last positoin then

        mov 12h,#16h ;set it to first position again
        ;but 1 less than it bcz in next instruction
        ;it will be incremented

        goto_nxt_4IR:inc 12h

        sjmp repeat_4IR


ch_01:cjne a,#01h,sel_button_pressed_4IR
```

```
                ;if key 2 is pressed means move the coin to left
                ;this logic is for game2
                ;for same key game1 has different logic

                        mov a,12h ;clear old coin position;coin pos saved
    in reg r2 of rb2
                        lcall choose_coord
                        mov dptr,#clear
                        lcall display_char

                ;below 4 lines are to make sure that the
                ;coin is still in range...

                ;our range is 17h to 1dh so if it is
                ;at 1dh so next position will be 17h


                        mov a,12h
                        cjne a,#17h,goto_nxt1_4IR ;if it is at 1dh location

                        ;i.e. last positoin then
                        ;set it to first position again but 1 less than
                        ;it bcz in next instruction it will be incremented

                        mov 12h,#1eh
                        goto_nxt1_4IR:dec 12h


                        repeat_4IR:
                        mov a,12h
                        lcall choose_coord

                        jnb 01h,player2 ;if 01h is set means p1's turn
                        ;is going on , else p2 turn so display the cursor
                        ;coin accordingly

                        mov dptr,#c_p1
                        lcall display_char

                        sjmp returnback_4IR

                        player2:
                        mov dptr,#c_p2
                        lcall display_char

                        sjmp returnback_4IR
```

```
        sel_button_pressed_4IR: ;check if select button is pressed

                cjne a,#07h,returnback_4IR

                setb 00h; to mark that position is
                ;selected 00h bit is set

                cpl 01h ; to mark that now turn of
                ;next player so 01h is toggled

        returnback_4IR:

                pop acc
                pop psw

                reti;for key-board isr_4IR

org 343h;lookup tables  for char
                ;black=1 ;upper nibble =lower 4 bits of 8 bits of col

        clear:  db 00h,00h,00h,00h,00h,00h,00h,00h

        cursor: db 0x00, 0xFF, 0xFF, 0x7E, 0x7E, 0x3C, 0x18, 0x00
        ;Code for Cursor (>)

        grid: db 0ffh,0c3h,81h,81h,81h,81h,0c3h,0ffh;

        coin1: db 0ffh,0ffh,0e7h,0dbh,0dbh,0e7h,0ffh,0ffh;
        coin2: db 0ffh,0ffh,0e7h,0c3h,0c3h,0e7h,0ffh,0ffh;

        c_p1: db 00h,3ch,42h,5ah,5ah,42h,3ch,00h; player 1 coin
        c_p2: db 00h,3ch,42h,42h,42h,42h,3ch,00h; player 2 coin

        dash:DB 00h,00h,18h,18h,18h,18h,00h,00h; -

        food: db 0ch, 12h, 22h, 44h, 44h, 22h, 12h, 0ch

        body_horizontal:db 18h,3ch,3ch,3ch,3ch,3ch,3ch,3ch,18h
        head_horizontal: db 18h,3ch,7eh,7eh,7eh,7eh,3ch,18h

        body_vertical:db 00h, 00h, 7eh, 0ffh, 0ffh, 7eh, 00h,00h
        head_vertical:db  00h, 3ch, 7eh, 0ffh, 0ffh, 7eh, 3ch, 00h

        bented_char_1: db 18h,3ch,7ch,7ch,0fch,7ch,00h,00h;-|
        bented_char_2:db 18h,3ch,3eh,3fh,3fh,3eh,00h,00h;_|
```

```
    bented_char_3: db 00h,00h,3eh,3fh,3fh,3eh,3ch,18h;|_
    bented_char_4:db 00h,00h,7ch,0fch,0fch,7ch,3ch,1ch;|-

    boundary_collision_values:db 00h,0fh,7fh,70h;4
    db 01h, 02h, 03h, 04h, 05h, 06h, 07h, 08h, 09h, 0ah, 0bh, 0ch, 0dh,
0eh;14
    db 1fh, 2fh, 3fh, 4fh, 5fh, 6fh;6
    db 7eh, 7dh, 7ch, 7bh, 7ah, 79h, 78h, 77h, 76h, 75h, 74h, 73h, 72h,
71h;14
    db 60h, 50h, 40h, 30h, 20h, 10h;6

    start_food_pos:db 11h,6eh,17h,61h,22h,5dh,28h,55h,33h,49h
    db 16h,64h,15h,51h,25h,63h,37h,59h,44h,1eh

    chA: DB 124,126,19,19,126,124,0,0; A
    chB: DB 127,127,73,73,127,54,0,0; B

    ; Characters 0-9
    C0: DB 00h,3eh,7fh,51h,49h,7fh,3eh,00h; 0
    C1: DB 00h,80h,88h,0feh,0feh,80h,80h,00h; 1
    C2: DB 00h,0c4h,0e6h,0a2h,92h,9eh,8ch,00h; 2
    C3: DB 00h,44h,0c6h,92h,92h,0feh,6ch,00h; 3
    C4: DB 00h,30h,28h,24h,0feh,0feh,20h,00h; 4
    C5: DB 00h,4eh,0ceh,8ah,8ah,0fah,72h,00h; 5
    C6: DB 00h,7ch,0feh,92h,92h,0f6h,64h,00h; 6
    C7: DB 00h,06h,06h,0e2h,0fah,1eh,06h,00h; 7
    C8: DB 00h,6ch,0feh,92h,92h,0feh,6ch,00h; 8
    C9: DB 00h,4ch,0deh,92h,92h,0feh,7ch,00h; 9

    ;characters of string gaming
    gaming: DB 62,127,65,81,81,113,0,0;G
                    DB 124,126,19,19,126,124,0,0;A
                    DB 127,127,6,12,6,127,127,0;M
                    DB 65,65,127,127,65,65,0,0;I
                    DB 127,127,6,12,24,127,127,0;N
                    DB 62,127,65,81,81,113,0,0;G

    ;characters of string YANTRA
    yantra:DB 7,15,120,120,15,7,0,0;Y
               DB 124,126,19,19,126,124,0,0;A
               DB 127,127,6,12,24,127,127,0;N
               DB 1,1,127,127,1,1,0,0;T
               DB 127,127,9,25,127,102,0,0;R
               DB 124,126,19,19,126,124,0,0;A

    ;characters of string G-YANTRA
```

```
        g_yantra:DB 62,127,65,81,81,113,0,0 ;G
                    DB 00h,00h,18h,18h,18h,18h,00h,00h ;-
                    DB 7,15,120,120,15,7,0,0 ;Y
                    DB 124,126,19,19,126,124,0,0 ;A
                    DB 127,127,6,12,24,127,127,0 ;N
                    DB 1,1,127,127,1,1,0,0 ;T
                    DB 127,127,9,25,127,102,0,0 ;R
                    DB 124,126,19,19,126,124,0,0 ;A

        ;characters of string SELECT
        select: DB 38,111,73,73,123,50,0,0 ;S
                   DB 127,127,73,73,65,65,0,0 ;E
                   DB 127,127,64,64,64,64,0,0 ;L
                   DB 127,127,73,73,65,65,0,0 ;E
                   DB 62,127,65,65,65,65,0,0 ;C
                   DB 1,1,127,127,1,1,0,0 ;T

        ;characters of string SCORE
        score: DB 38,111,73,73,123,50,0,0 ;S
               DB 62,127,65,65,65,65,0,0 ;C
               DB 62,127,65,65,127,62,0,0 ;O
               DB 127,127,9,25,127,102,0,0 ;R
               DB 127,127,73,73,65,65,0,0 ;E

        ;characters of string GAME
        game: DB 62,127,65,81,81,113,0,0 ;G
              DB 124,126,19,19,126,124,0,0 ;A
              DB 127,127,6,12,6,127,127,0 ;M
              DB 127,127,73,73,65,65,0,0 ;E

        ;characters of string over
        over: DB 62,127,65,65,127,62,0,0 ;O
              DB 31,63,96,96,63,31,0,0 ; V
              DB 127,127,73,73,65,65,0,0 ;E
              DB 127,127,9,25,127,102,0,0 ;R

        ;characters of string DEVELOPED
        developed:DB 127,127,65,65,127,62,0,0 ;D
                     DB 127,127,73,73,65,65,0,0 ;E
                     DB 31,63,96,96,63,31,0,0 ;V
                     DB 127,127,73,73,65,65,0,0 ;E
                     DB 127,127,64,64,64,64,0,0 ;L
                     DB 62,127,65,65,127,62,0,0 ;O
                     DB 127,127,9,9,15,6,0,0 ;P
                     DB 127,127,73,73,65,65,0,0 ;E
                     DB 127,127,65,65,127,62,0,0 ;D
```

```
        ;characters of string BY
        by:DB 127,127,73,73,127,54,0,0 ;B
           DB 7,15,120,120,15,7,0,0 ;Y

        ;characters of string IN
        in:DB 65,65,127,127,65,65,0,0 ;I
           DB 127,127,6,12,24,127,127,0 ;N


        ;characters of string TO
        to:DB 1,1,127,127,1,1,0,0 ;T
           DB 62,127,65,65,127,62,0,0 ;O

        ;characters of string PRESS
        press: DB 127,127,9,9,15,6,0,0 ; P
                   DB 127,127,9,25,127,102,0,0 ;R
                   DB 127,127,73,73,65,65,0,0 ;E
                   DB 38,111,73,73,123,50,0,0 ;S
                   DB 38,111,73,73,123,50,0,0 ;S

        ;characters of string PLAY
        play: DB 127,127,9,9,15,6,0,0 ; P
                  DB 127,127,64,64,64,64,0,0 ;L
                  DB 124,126,19,19,126,124,0,0 ;A
                  DB 7,15,120,120,15,7,0,0 ;Y

        ;characters of string PLAYER
        player: DB 127,127,9,9,15,6,0,0 ;P
                        DB 127,127,64,64,64,64,0,0 ;L
                        DB 124,126,19,19,126,124,0,0 ;A
                        DB 7,15,120,120,15,7,0,0 ;Y
                        DB 127,127,73,73,65,65,0,0 ;E
                        DB 127,127,9,25,127,102,0,0 ;R

        ;characters of string START
        start: DB 38,111,73,73,123,50,0,0 ;S
                   DB 1,1,127,127,1,1,0,0 ;T
                   DB 124,126,19,19,126,124,0,0 ;A
                   DB 127,127,9,25,127,102,0,0 ;R
                   DB 1,1,127,127,1,1,0,0 ;T

        ;characters of string ROW
        row:DB 127,127,9,25,127,102,0,0 ;R
                DB 62,127,65,65,127,62,0,0 ;O
                DB 127,127,48,24,48,127,127,0 ; W

        ;characters of string RUDRA
```

```
        rudra:DB 127,127,9,25,127,102,0,0 ;R
                 DB 63,127,64,64,127,63,0,0 ;U
                 DB 127,127,65,65,127,62,0,0 ;D
                 DB 127,127,9,25,127,102,0,0 ;R
                 DB 124,126,19,19,126,124,0,0 ;A

        ;characters of string JOSHI
        joshi:DB 48,112,64,64,127,63,0,0 ;J
                 DB 62,127,65,65,127,62,0,0 ;O
                 DB 38,111,73,73,123,50,0,0 ;S
                 DB 127,127,8,8,127,127,0,0 ;H
                 DB 65,65,127,127,65,65,0,0 ;I

        ;characters of string &
        amp:db 60h,0f4h,9eh,0bah,6eh,0f4h,90h,00h ;&

        ;characters of string TANVI
        tanvi:DB 1,1,127,127,1,1,0,0 ;T
                 DB 124,126,19,19,126,124,0,0 ;A
                 DB 127,127,6,12,24,127,127,0 ;N
                 DB 31,63,96,96,63,31,0,0 ;V
                 DB 65,65,127,127,65,65,0,0 ;I

        ;characters of string SHAH
        shah: DB 38,111,73,73,123,50,0,0 ;S
                 DB 127,127,8,8,127,127,0,0 ;H
                 DB 124,126,19,19,126,124,0,0 ;A
                 DB 127,127,8,8,127,127,0,0 ;A

        ;characters of string SNAKE
        snake:DB 38,111,73,73,123,50,0,0 ;S
                 DB 127,127,6,12,24,127,127,0 ;N
                 DB 124,126,19,19,126,124,0,0 ;A
                 DB 127,127,8,28,119,99,0,0 ;K
                 DB 127,127,73,73,65,65,0,0 ;E

org 079dh

        g_yantra_intro: ;for project intro

                        push psw

                        mov a,#31h ;set the coordinates from where 1st
                        ;character of string is to be displayed

                        lcall choose_coord
                        mov dptr ,#gaming
```

```
                    mov b,a ; first char location in b
                    mov a,#6 ;total characters in a
                    lcall display_string ;location of string
                    ;first element in dptr

                    mov a,#38h ;set the coordinates from where 1st
                    ;character of string is to be displayed

                    lcall choose_coord
                    mov dptr ,#yantra
                    mov b,a ; first char location in b
                    mov a,#6 ;total characters in a
                    lcall display_string ;location of string first
                    ;element in dptr

                    ;to give user some time to read the text
                    lcall delay1s
                    lcall delay1s

                    mov a,#54h ;set the coordinates from where 1st
                    ;character of string is to be displayed

                    lcall choose_coord
                    mov dptr ,#g_yantra
                    mov b,a ; first char location in b
                    mov a,#8 ;total characters in a
                    lcall display_string ;location of string
                    ;first element in dptr

                    ;to give user some time to read the text
                    lcall delay1s
                    lcall delay1s

                    pop psw

    ret ;for g_yantra_intro

    developers_intro:;to display developers intro

                    push psw

                    mov a,#32h ;set the coordinates from where
                    ;1st character of string is to be displayed

                    lcall choose_coord
                    mov dptr ,#developed
                    mov b,a ; first char location in b
```

```
                        mov a,#9 ;total characters in a
                        lcall display_string ;location of string
                        ;first element in dptr

                        mov a,#3ch
                        lcall choose_coord
                        mov dptr ,#by
                        mov b,a ; first char location in b
                        mov a,#2 ;total characters in a
                        lcall display_string ;location of string
                        ;first element in dptr

                        ;delay is added to give user time to read the
    string
                        lcall delay1s

                        lcall clrscreen

                        mov a,#13h
                        lcall choose_coord
                        mov dptr ,#tanvi
                        mov b,a ; first char location in b
                        mov a,#5 ;total characters in a
                        lcall display_string ;location of string
                        ;first element in dptr

                        mov a,#19h
                        lcall choose_coord
                        mov dptr ,#shah
                        mov b,a ; first char location in b
                        mov a,#4 ;total characters in a
                        lcall display_string ;location of string
                        ;first element in dptr

                        mov a,#37h
                        lcall choose_coord
                        mov dptr,#amp
                        lcall display_char

                        mov a,#53h
                        lcall choose_coord
                        mov dptr ,#rudra
                        mov b,a ; first char location in b
                        mov a,#5 ;total characters in a
                        lcall display_string ;location of string
                        ;first element in dptr
```

```
                    mov a,#59h
                    lcall choose_coord
                    mov dptr ,#joshi
                    mov b,a ; first char location in b
                    mov a,#5 ;total characters in a
                    lcall display_string ;location of string
                    ;first element in dptr

                    lcall delay1s
                    lcall delay1s
                    lcall delay1s

                    pop psw

    ret ;for develeopers_intro

    gui: ; functions to provide a basic subroutine

            push psw

            setb psw.3
            setb psw.4

            lcall clrscreen

            clr 0ah ;clr bit 0ah for safety purposes

            mov a,#11h
            lcall choose_coord
            mov dptr ,#select
            mov b,a ; first char location in b
            mov a,#6 ;total characters in a
            lcall display_string ;location of string
            ;first element in dptr

            mov a,#18h
            lcall choose_coord
            mov dptr ,#chA
            lcall display_char ;location of string
            ;first element in dptr

            mov a,#1Ah
            lcall choose_coord
            mov dptr ,#game
            mov b,a ; first char location in b
            mov a,#4 ;total characters in a
            lcall display_string ;location of string
```

```
;first element in dptr

mov a,#32h
lcall choose_coord
mov dptr ,#C1
lcall display_char ;location of string
;first element in dptr

display_snake_game:
jb 0ah,choose_diff_coord_1
mov a,#34h
sjmp done1
choose_diff_coord_1:
mov a,#53h
done1:
lcall choose_coord
mov dptr ,#snake
mov b,a ; first char location in b
mov a,#5 ;total characters in a
lcall display_string ;location of string
;first element in dptr

jb 0ah,choose_diff_coord_2
mov a,#3ah
sjmp done2
choose_diff_coord_2:
mov a,#59h
done2:
lcall choose_coord
mov dptr ,#game
mov b,a ; first char location in b
mov a,#4 ;total characters in a
lcall display_string ;location of string
;first element in dptr

jb 0ah,in_range
sjmp continue

in_range:
ajmp back2game_select

continue:
mov a,#52h
lcall choose_coord
mov dptr ,#C2
lcall display_char ;location of string
;first element in dptr
```

```
display_4_in_a_row:

mov a,#54h
lcall choose_coord
mov dptr ,#C4
lcall display_char ;location of string
;first element in dptr

mov a,#56h
lcall choose_coord
mov dptr ,#in
mov b,a ; first char location in b
mov a,#2 ;total characters in a
lcall display_string ;location of string
;first element in dptr

mov a,#59h
lcall choose_coord
mov dptr ,#chA
lcall display_char ;location of string
;first element in dptr

mov a,#5bh
lcall choose_coord
mov dptr ,#row
mov b,a ; first char location in b
mov a,#3 ;total characters in a
lcall display_string ;location of string
;first element in dptr

jb 0ah,back2game_select

mov ie,#81h ;enable external interupt at
;this point to start accepting keyboard inputs

mov r3,#30h
clr c

mov a,r3
lcall choose_coord
mov dptr,#cursor
lcall display_char

setb 09h ;set game selection bit

wait_for_input: jb 08h,wait_for_input
```

```
                ;loop till a game is selected and
                ;selection but 08h is cleared

                pop psw

        ret ;for gui

        game_select: ;to start a given game

                push psw

                setb psw.3
                setb psw.4

                lcall clrscreen

                mov a,#12h
                lcall choose_coord
                mov dptr ,#press
                mov b,a ; first char location in b
                mov a,#5 ;total characters in a
                lcall display_string ;location of string
                ;first element in dptr

                mov a,#18h
                lcall choose_coord
                mov dptr ,#start
                mov b,a ; first char location in b
                mov a,#5 ;total characters in a
                lcall display_string ;location of string
                ;first element in dptr

                mov a,#34h
                lcall choose_coord
                mov dptr ,#to
                mov b,a ; first char location in b
                mov a,#2 ;total characters in a
                lcall display_string ;location of string
                ;first element in dptr

                mov a,#37h
                lcall choose_coord
                mov dptr ,#play
                mov b,a ; first char location in b
                mov a,#4 ;total characters in a
                lcall display_string ;location of string
                ;first element in dptr
```

```
        setb 0ah ;set bit loaction ah for string display

        jb 09h,snake_selected ;if bit 09h is set the
        ;snake game was selected else other game was selected

        ajmp display_4_in_a_row ;display 4 in a row from
        ;previously written commands and hence save space

        back2game_select:

        sjmp wait_for_start

        snake_selected:

        ljmp display_snake_game

        wait_for_start: jnb 0bh,wait_for_start ;keep on looping
        ;here till start button is pressed

        clr 0bh ;clear start button once its use case is over

        lcall clrscreen

        jnb 09h,four_in_a_row_selected ;if 09h bit location
        ;is zero then call 4 in a Row Game

        ljmp snake_game


        four_in_a_row_selected:

        ljmp four_4_in_a_row


snake_game:

        setb psw.4
        clr psw.3 ;set Reg-Bank-2 For Game Operations

;lets clear what each register of this register bank represents:

;r0->stores the value of ram location 30h from where
;the coordinates of snakes body position can be accessed

;r1->stores the coordinates of head of snake
```

```
        ;r2->stores the coordinates of tail of snake

        ;r3->stores the direction in which the snake is supposed
        ;to move currently acts as direction register

        ;r4->stores the length of the midlle body section of snake

        ;1dh reg-5 of RB-3->tores the score of snake

        ;r6,r7->these are kept free for any copying
        ;use in any game related operation

        ;r5->food position

        ;Here Coordinates follow (y,x) System where
        ;y=rows= page number of glcd (0-7) and
        ;x=columns= 8x8 grid columns of glcd (0-fh)

                mov r0,#30h ;store the value of ram locations
                ;that will be used to store body coordinates
                ;set the intial coordinates of snake:
                ;head->(y,x)=>0,3, body->0,1, tail->0,1

                mov r1,#14h ;set initial head coordinates
                mov @r0,#13h ;set initial body coordinates
                mov r2,#13h     ;set initial tail coordinates

                mov r3,#00h ;initially start moving towards left
                mov r4,#02h ;length at start contains only one
                ;middle section this is required to match with
                ;the later subroutines for increasing length

                mov r5,#35h ; set the initial food position

                mov 19h,@r0 ;19h (r1 of reg-bank-3) for old body coord
                mov b,r2 ;b for old tail

                mov a,r1
                mov r6,a ; r6 for old head
                mov 18h,r1 ;18h (r0 of reg-bank-3) for head

                mov 1dh,#00 ;seting the score register to
                ;zero immdiate value at start of game

                mov a,r5 ; load a with initial food coordinates
                lcall choose_coord ;these instructions are concerned
                ;with displaying food at location present in r5
```

```
            mov dptr,#food
            lcall display_char

            sg_loop:

            mov ie,#00h
            lcall calc_pos
            lcall update_pos
            lcall update_lcd
            mov ie,#81h
            lcall delay1s

            sjmp sg_loop

      calc_pos: ;subroutine to calculate position of head

            setb psw.4 ;reg-bank-2
            clr psw.3

            clr c
            ;For Our Case p0.0->Right, p0.1->Left
            ;p0.2->Up, p0.3->Down

            mov a,r1 ;store the old snake heads
            ;coordinates temporary in r6

            mov r6,a

            ;the follwoing three lines of codes will assist later
            ;in assigning proper charcters as per various situations...

            mov 18h,a ;store old head coordinate in 18h
            ;i.e. r0 of reg bank-3

            mov 1bh,r2; store old tail coordinates in 1ch
            ;i.e. r3 of reg bank-3

            mov 19h,@r0 ; store old body coordinates in 19h
            ;i.e. r1 of reg bank-3

            mov a,r3 ;mov direction info to reg-a
            mov 1ch,a ;store the current direction in 1ch
            ;i.e. r4 of reg-bank-3

            cjne a,#00h,nxt_01
            inc r1 ;mov snake's head in +-ve x direction
```

```
                sjmp ext

                nxt_01:

                cjne a,#01h,nxt_02
                dec r1 ;mov snake's head in -ve x direction
                sjmp ext

                nxt_02:

                cjne a,#02h,nxt_03
                mov a,r1
                subb a,#10h ;dec upper niblle i.e. y--
                mov r1,a
                sjmp ext

                nxt_03:

                cjne a,#03h,ext
                mov a,r1
                add a,#10h ;inc upper nibble i.e. y++
                mov r1,a

        ext:
        ret ;for calc_pos subroutine

        update_pos: ;this subroutine updates the coordinates of
        ;snakes body exluding head

                setb psw.4 ;reg-bank-2
                clr psw.3

                mov b,r2 ;store the previous tail location
                ;in b to later clear it

                mov a,r0 ;mov the coordinates of middle position to tail
                add a,r4 ;to give access of memory location body
                ;element immediately ahead of tail

                dec a ;for indexing as r4=01h by default
                mov r0,a

                mov a,@r0 ;access the coordinates of the body
                ;elements

                mov r2,a ;append those coordinates to current tail
```

```
mov a,r6; store the old snake heads position to
;1st element immediately behind head of middle segment

mov r0,#30h ;reloading r0 to original value as this
;body element is the one that is exactly behind head

mov @r0,a ;copying old head coordinates in middle
;segment exactly behind the head

cjne r4,#01h,len_nt_1

sjmp ext_1

len_nt_1:

push 14h ;temporarily (reg-R4 of Reg-Bank-2)

dec r4 ;again for indexing purposes

mov a,r0 ;access the last element of array
add a,r4 ;place its memory location in r0
mov r0,a

continue_len_logic:

push b ;temproarily

mov b,r0;store the memory location of
;currently accessed element

dec r0

mov a,@r0;access the coordinates stored at the place

push 10h ;temporarily (reg-R0 of Reg-Bank-2)

mov r0,b ;access the element behind
;the cuurent using this

mov @r0,a;load the coordinate values of element
;ahead in element behind

pop 10h ;(reg-R0 of Reg-Bank-2)

pop b

djnz r4,continue_len_logic ;at the end of loop r0
```

```
                ;will again be 30h Atleast that's what I hope

                pop 14h ;(reg-R4 of Reg-Bank-2)

                mov r0,#30h ;to be on safer side

        ext_1:

        ret ;for update_pos

        update_lcd: ;this function converts the coordinates
        ;to lcd values and displays the same
        ;lcd clr and cursor off yet to be given

                setb psw.4 ;reg-bank-2
                clr psw.3

                lcall clear_tail

                ;remember that old tail coordinate after
                ;executing above subroutine still remains in reg-b

                lcall update_head_position

                push 14h ;temporarily

                loop_body_elements:

                cjne r0,#30h,skip_itr ;if the current r0=30h then
                ;and only then body update is allowed otherwise
                ;updation is skipped

                lcall update_body_position ;this is because this
                ;subroutine is specially designed for body element
                ;just behind head

                skip_itr: ;now we may wonder then how
                ;is the game working if there is no provision to
                ;update rest of array elements based
                ;on formulas derived...?
                ;this because update_pos subroutine effectively
                ;shifts the coordinates
                ;that rest of the body elements follow what
                ;the 1st body element followed when it was there
                ;and their animation/charcter is then cleared
                ;when a tail element arrives
                ;at same position so in this way
```

```
                ;we optimized for memory and computations for
                ;each element instead we focus on only 3-elements
                ;and the rest of the thing follows...

                inc r0

                djnz r4,loop_body_elements

                pop 14h

                mov r0,#30h ;to be on safe side

                lcall update_tail_position

                mov a,r1 ; store new head position in r1
                cjne a,15h,exit_update_lcd ; compare head position
                ;with current food position

                inc 1dh ;i.e. R5 of reg-bank-3 used for score tracking

                inc r4 ;of reg-bank-2 which contains
                ;the length of middle body seection

                lcall food_pos ;update food_pos if it merge with head

                exit_update_lcd:
                lcall check_coll

        ret ;for update_lcd

        clear_tail: ;this subroutine clears the tail on glcd
        ;at old coordinates

                mov a,b ; old tail in b
                lcall choose_coord
                mov dptr,#clear
                lcall display_char

        ret;for clear tail

        update_head_position:

                mov a,r1 ;load the current head coordinates in reg-a
                lcall choose_coord ;set the coordinates on GLCD

                clr c
                subb a,18h ; find h(result)=(h[new]-h[old])
```

```
            ;what follows now is a switch case type instructions
            ;based on the patterns I have observed
            ;and tabulated in my diary

            cjne a,#10h,next_head_coord1

            mov dptr,#head_vertical
            sjmp exit_head

            next_head_coord1:

            cjne a,#0f0h,next_head_coord2

            mov dptr,#head_vertical
            sjmp exit_head

            next_head_coord2:

            cjne a,#01h,next_head_coord3

            mov dptr,#head_horizontal
            sjmp exit_head

            next_head_coord3:

            mov dptr,#head_horizontal

            exit_head:

            lcall display_char

    ret ;for update_head_position

    update_body_position:

            mov a,@r0 ;load the current body coordinates in a
            lcall choose_coord ;set the coordinates on GLCD

            clr c
            subb a,19h ;find b(ro)=(b[new]-b[old])

            cjne a,#01h,next_body_coord_1 ;checks if bro=01h

                    mov a,r1 ;if a=01h check for br status value
                    clr c
                    subb a,@r0 ;b(r)=(h[new]-b[new])
```

```
                cjne a,#01h,skip_bro01_01

                        mov dptr,#body_horizontal
                        sjmp exit_body

                skip_bro01_01:

                cjne a,#10h,skip_bro01_02

                        mov dptr,#bented_char_1
                        sjmp exit_body

                skip_bro01_02:

                        mov dptr,#bented_char_2
                        sjmp exit_body

        next_body_coord_1:

        cjne a,#10h,next_body_coord_2 ;checks if bro=10h

                mov a,r1 ;if a=10h check for br status value
                clr c
                subb a,@r0 ;b(r)=(h[new]-b[new])

                cjne a,#01h,skip_bro10_01

                        mov dptr,#bented_char_3
                        sjmp exit_body

                skip_bro10_01:

                cjne a,#10h,skip_bro10_02

                        mov dptr,#body_vertical
                        sjmp exit_body

                skip_bro10_02:

                        mov dptr,#bented_char_2
                        sjmp exit_body

        next_body_coord_2:

        cjne a,#0f0h,next_body_coord_3 ;checks if bro=0f0h
```

```
                        mov a,r1 ;if a=0f0h check for br status value
                        clr c
                        subb a,@r0 ;b(r)=(h[new]-b[new])

                        cjne a,#01h,skip_bro0f0_01

                                mov dptr,#bented_char_4
                                sjmp exit_body

                        skip_bro0f0_01:

                        cjne a,#0f0h,skip_bro0f0_02

                                mov dptr,#body_vertical
                                sjmp exit_body

                        skip_bro0f0_02:

                                mov dptr,#bented_char_1
                                sjmp exit_body

                next_body_coord_3: ;if nothing matches then bro=0ffh

                        mov a,r1 ;if a=0ffh check for br status value
                        clr c
                        subb a,@r0 ;b(r)=(h[new]-b[new])

                        cjne a,#10h,skip_bro0ff_01

                                mov dptr,#bented_char_4
                                sjmp exit_body

                        skip_bro0ff_01:

                        cjne a,#0f0h,skip_bro0ff_02

                                mov dptr,#bented_char_3
                                sjmp exit_body

                        skip_bro0ff_02:

                                mov dptr,#body_horizontal

                exit_body:

                lcall display_char
```

```
        ret ;for update_body_position

        update_tail_position:

                mov a,r2 ;load the current tail coordinates in a
                lcall choose_coord ;set the coordinates on GLCD

                clr c
                subb a,1bh ;find t(ro)=(t[new]-t[old])

                cjne a,#01h,next_tail_coord_1 ;checks if tro=01h

                        mov a,r0 ;access the last element of array
                        add a,r4 ;place its memory location in r0
                        mov r0,a
                        dec r0 ;for indexing purposes
                        mov a,@r0 ;if a=01h check for tr status value
                        clr c
                        subb a,r2 ;t(r)=(b[new]-t[new])

                        cjne a,#01h,skip_tro01_01

                                mov dptr,#body_horizontal
                                sjmp exit_tail

                        skip_tro01_01:

                        cjne a,#10h,skip_tro01_02

                                mov dptr,#bented_char_1
                                sjmp exit_tail

                        skip_tro01_02:

                                mov dptr,#bented_char_2
                                sjmp exit_tail

                next_tail_coord_1:

                cjne a,#10h,next_tail_coord_2 ;checks if tro=10h

                        mov a,r0 ;access the last element of array
                        add a,r4 ;place its memory location in r0
                        mov r0,a
                        dec r0 ;for indexing purposes
                        mov a,@r0 ;if a=10h check for tr status value
                        clr c
```

```
                        subb a,r2 ;t(r)=(b[new]-t[new])

                        cjne a,#01h,skip_tro10_01

                                mov dptr,#bented_char_3
                                sjmp exit_tail

                        skip_tro10_01:

                        cjne a,#10h,skip_tro10_02

                                mov dptr,#body_vertical
                                sjmp exit_tail

                        skip_tro10_02:

                                mov dptr,#bented_char_2
                                sjmp exit_tail

                next_tail_coord_2:

                cjne a,#0f0h,next_tail_coord_3 ;checks if tro=0f0h

                        mov a,r0 ;access the last element of array
                        add a,r4 ;place its memory location in r0
                        mov r0,a
                        dec r0 ;for indexing purposes
                        mov a,@r0 ;if a=0f0h check for tr status value
                        clr c
                        subb a,r2 ;t(r)=(b[new]-t[new])

                        cjne a,#01h,skip_tro0f0_01

                                mov dptr,#bented_char_4
                                sjmp exit_tail

                        skip_tro0f0_01:

                        cjne a,#0f0h,skip_tro0f0_02

                                mov dptr,#body_vertical
                                sjmp exit_tail

                        skip_tro0f0_02:

                                mov dptr,#bented_char_1
                                sjmp exit_tail
```

```
            next_tail_coord_3: ;if nothing matches then tro=0ffh

                    mov a,r0 ;access the last element of array
                    add a,r4 ;place its memory location in r0
                    mov r0,a
                    dec r0 ;for indexing purposes
                    mov a,@r0 ;if a=0ffh check for tr status value
                    clr c
                    subb a,r2 ;t(r)=(b[new]-t[new])

                    cjne a,#10h,skip_tro0ff_01

                            mov dptr,#bented_char_4
                            sjmp exit_tail

                    skip_tro0ff_01:

                    cjne a,#0f0h,skip_tro0ff_02

                            mov dptr,#bented_char_3
                            sjmp exit_tail

                    skip_tro0ff_02:

                            mov dptr,#body_horizontal

            exit_tail:

            lcall display_char

            mov r0,#30h

    ret ;for update_tail_position

    food_pos:;r5 of reg bank 2 and r6 of reg bank 2,
    ;need to chage r6


            setb psw.4
            clr psw.3
            push 16h

            mov dptr,#start_food_pos ;in dptr the starting value
            ;where all random food coordinates are stored...
            ;for now total 20 coordinates are there
```

```
                mov r5,tl0 ;random value in r5
                mov a,r5 ;random value in a
                anl a,#19 ;limit the random value in
                ;0 to 19(total 20 coordinates saved) bcz it is basically
                ;the count which we will add to dptr to point
                ;to some random coordinate

                mov r6,a ;r6 will remember the count i.e form
                ;'#start' which position in lookup
                ;we are pointing that count

                movc a,@a+dptr ;mov the coord corresponding to pointed
                ;value in a...so now a has random food position but
                ;randomness is controlled

                mov r5,a ;mov the food coordinate into r5

                check_overlap:

                        mov 20h,r1 ;due to syntax limitaion of cjne
                        ;we are using 20h to store specific value
                        ;(head,tail or other body element) at a time
                        ;so that it can be compared with the newly
                        ;generated conrolled random food coordinate
                        ;which is till now stored in r5 and a

                        cjne a,20h,ch_ot ;compare food coord with head
    coord

                        sjmp nxt_coord ;if equal we have to change
                        ;coord so jmp to nxt_coord

                ch_ot:mov 20h,b ;mov old tail coord in 20h
                          cjne a,20h,ch_t ;if not equal we will compare
                          ;tail and food coordinate

                          sjmp nxt_coord


                ch_t:mov 20h,r2 ;mov tail coord in 20h
                         cjne a,20h,ch_b ;if not equal we will
                         ;compare tail and food coordinate

                         sjmp nxt_coord ;if equal jmp to nxt coord
                         ;else go for body comparision

                ch_b: mov 20h,@r0 ;first mov 1st body elemnt coord
```

```
              ;that is stored in 30h (r0 points 30h) in 20h
                        cjne r4,#01h,ch_b_len_nt1 ;if length is one then
                        ;we only have to check whether food overlaps
                        ;with one body element but if it is > 1 the we'll
                        ;have to check for (r4) body elemnts...
                        ;(r4)=length of body elements

                        cjne a,20h,finally_done ;check for 1 elemnt as r4
   =1
                        ;and if no overlapping then done otherwise
   nxt_coord

                        sjmp nxt_coord

              ch_b_len_nt1: push 10h ;first push r0=10h bcz due to
              ;limitaion of reg we will use r0 to point diff elements

                                    push 14h ;r4=14h ,reg bank 2

              do_again: cjne a,20h,nxt_element ;a=food coord and
              ;20h = coord in 30h for first iteration...
              ;compare and decide accordingly

                                  sjmp nxt_coord

              nxt_element: inc r0     ; element in 30h is compared the
              ;in r0 so now it point to 31h

                                       mov 20h,@r0 ;mov value stored in
   31h
                                       ;in 20h

                                       djnz r4,do_again ;repeat for r4
   times
                                       ;i.e if body length is 2 then r4=2
    and
                                       ;this will repeat for 30h and 31h

                                       pop 14h ;pop r4
                                       pop 10h ; pop r0

                                       sjmp finally_done ;if till here it
    has
                                       ;not jumped to nxt_coord this
   means
                                       ;there is no overlapping so we are
    good to go
```

```
        nxt_coord: cjne r6,#19,barabar ;r6 has count of
        ;which random position out stored 10 we are using...
        ;if count is 10 then it means that it will be pointing
        ;to last element of pur lookup array so we will
        ;re-initialize ut by making r6 0

                        mov r6,#00h ;re initialize r6
                        sjmp check_again

        barabar: inc r6 ; if r6<19 then no issue we can
        ;simply in it and then get respective value

                        mov a,r6 ;mov count in a...so say if
                        ;r6 was 3h so after incr it is 4h and
                        ;so now a has 4 stored

        check_again: movc a,@a+dptr     ;thus from the stored
        ;values choose 4th one

                        sjmp check_overlap ;as it entered
                        ;nxt_coord it meant there was overlaping
                        ;somewhere so after giving new coordinates
                        ;again check if still overlapping exists

finally_done:

        mov r5,a

        lcall choose_coord
        mov dptr,#food
        lcall display_char
        pop 16h

        mov a,r4 ;cuurent body element length

        cjne a,#01h,len_incremented

        sjmp exit_food_pos

        len_incremented:

        dec a ;for indexing puropses
        add a,r0 ;find the element location (memory)
        ;associated with increased length
```

```
                mov r0,a ;and copy its location in r0

                mov a,r2 ;copy cuurent tail location
                ;in element location pointed by r2

                mov @r0,a

                mov r0,#30h ;after this restore r0 to
                ;original value since this is required
                ;to be done only for tail and the new
                ;body element add ahead of it

                mov r2,b ;store the current tails coordinates
                ;to the one from where it was
                ;cleared earlier because food element is eaten
                ;and condition requires to do so

        exit_food_pos:

        ret ;ret from food_pos

        check_coll: ;this subroutine checks for collision
        ;with GLCD boundary coordinates

                mov a,r1

                push psw

                setb psw.4
                setb psw.3

                mov r7,a
                mov dptr,#boundary_collision_values
                mov r0,#44

                chk_nxt:clr a
                                movc a,@a+dptr
                                cjne a,1fh,cont
                                ljmp game_over
                                cont:inc dptr
                djnz r0,chk_nxt

                pop psw

        ret;for check_coll

        self_collision: ;this subroutine checks for
```

```
;self-collision of snake with itself

        ;this is called at Keyboards ISR only and checks
        ;if the latest key press is for or against
        ;a given 1-d Direction

        cjne a,#00h,check_left ;if not right then check left

                clr c
                subb a,1ch ;subtract the old direction info

                clr c;

                subb a,#0ffh ;00h-01h =0ffh thus after this
                ;if a=0 then left is detected
                ;i.e. initial movement was left
                ;but new movment is towards right

                jnz exit_self_collision

                sjmp game_over_call ;however if a=0
                ;then right key pressed
                ;for a head moving left
                ;and hence we jump to game-over

        check_left:

        cjne a,#01h,check_up ;if not left then check up

                clr c
                subb a,1ch ;subtract the old direction info

                clr c;

                subb a,#01h ;01h-00h =01h thus after this
                ;if a=0 then right is detected
                ;i.e. initial movement was right
                ;but new movment is towards left

                jnz exit_self_collision

                sjmp game_over_call ;however if
                ;a=0 then right key pressed
                ;for a head moving left
                ;and hence we jump to game-over

        check_up:
```

```
            cjne a,#02h,check_down ;if not right then ckeck left

                    clr c
                    subb a,1ch ;subtract the old direction info

                    clr c;

                    subb a,#0ffh ;02h-03h =0ffh thus after this
                    ;if a=0 then down is detected
                    ;i.e. initial movement was down
                    ;but new movment is towards up

                    jnz exit_self_collision

                    sjmp game_over_call ;however if a=0
                    ;then up key pressed for a head moving down
                    ;and hence we jump to game-over

            check_down:

                    clr c
                    subb a,1ch ;subtract the old direction info

                    clr c;

                    subb a,#01h ;03h-02h =01h thus after this
                    ;if a=0 then up is detected
                    ;i.e. initial movement was up but
                    ;new movment is towards down

                    jnz exit_self_collision

                    game_over_call:ljmp game_over ;however
                    ;if a=0 then up key pressed for
                    ;a head moving down and hence we call game-over

    exit_self_collision:

    ret ;for self_collision

    calc_score: ;this subroutine calculates the score and
    ;diplays the 2-digit code form 0-99 in decimal number system
    ;so yes this code can show you correct scores only for
    ;values form 0-99 in decimal also the logic used to
    ;display a given binary number/stored as hex number to
    ;its equivalent BCD is shown in Book By MKP Sir Ch-9 Ex-9.1
```

```
;One Can Refer that for that logic understanting

        mov a,18h ;move cuurent score value in reg-a

        mov b,#10 ;move 10d in reg-b

        div ab ;quotient is stored in a (10's place digit) &
        ;remainder is stored in b (1's Digit Number)

        clr 00h ;clr bit location for iterating
        ;for both the digits

        now_1s_place:

        cjne a,#00h,check_1 ;checkes if content
        ;of a is 00h takes appropriate action similar
        ;thing is done for other instructions below

                mov dptr,#C0
                sjmp go_to_display

        check_1:

        cjne a,#01h,check_2

                mov dptr,#C1
                sjmp go_to_display

        check_2:

        cjne a,#02h,check_3

                mov dptr,#C2
                sjmp go_to_display

        check_3:

        cjne a,#03h,check_4

                mov dptr,#C3
                sjmp go_to_display

        check_4:

        cjne a,#04h,check_5

                mov dptr,#C4
```

```
                sjmp go_to_display

        check_5:

        cjne a,#05h,check_6

                mov dptr,#C5
                sjmp go_to_display

        check_6:

        cjne a,#06h,check_7

                mov dptr,#C6
                sjmp go_to_display

        check_7:

        cjne a,#07h,check_8

                mov dptr,#C7
                sjmp go_to_display

        check_8:

        cjne a,#08h,check_9

                mov dptr,#C8
                sjmp go_to_display

        check_9:

                mov dptr,#C9

        go_to_display:

                jb 00h,almost_there ;check if carry
                ;is set or not if not
                ;then load 10's digit number
                ;on GLCD else jmp to label


                mov a,#3ah
                lcall choose_coord
                lcall display_char
                sjmp next_digit ;after displaying 10's digit
                ;at appropraite place in glcd
```

```
                    ;make decsions for 1's digit

                    almost_there:

                    mov a,#3bh ;since carry was set now display
                    ;1's digit number on glcd and
                    ;then return from subroutine

                    lcall choose_coord
                    lcall display_char
                    sjmp exit_calc_score

            next_digit:

            setb 00h
            mov a,b
            sjmp now_1s_place ;iterate the procedure
            ;again for 1's digit number

    exit_calc_score:

    ret ;for calc score

    disp_score: ;this subroutine displays the
    ;score of user after a game-over

            mov a,#34h
            lcall choose_coord
            mov dptr ,#score
            mov b,a ; first char location in b
            mov a,#5 ;total characters in a
            lcall display_string ;location of string
            ;first element in dptr

            lcall calc_score ;calculate score as per
            ;value in score register and display the digits

    ret ;for display_score

    game_over: ;this subroutine is used to display gameover
    ;string along with score display at end

            lcall clrscreen
            mov a,1dh;
        mov 18h,a
            mov a,#36h
            lcall choose_coord
```

```
                mov dptr ,#game
                mov b,a ; first char location in b
                mov a,#4 ;total characters in a
                lcall display_string ;location of string
                ;first element in dptr

                mov a,#46h
                lcall choose_coord
                mov dptr ,#over
                mov b,a ; first char location in b
                mov a,#4 ;total characters in a
                lcall display_string ;location of string
                ;first element in dptr

                lcall delay1s
                lcall delay1s
                lcall delay1s
                lcall delay1s

                lcall clrscreen
                jnb 09h,ends
                lcall disp_score

                ends: sjmp ends

org 0d00h

        ;_4_in_a_row game starts here

                ;reg bank 2 used for game operations

                ;r0 (10h)-> used as pointer to memory mapped
                ;locations corresponding to GLCD Coordinates
                ;which is from 30h to 59h in RAM

                ;r1 (11h)-> used as pointers to memory loaction
                ;of which column is filled by what amount
                ;which is from 21h to 27h

                ;r2 (12h)-> col position info stored in this reg

                ;r3 -> temporarily store the the value upto which
                ;a given selected column is filled

                ;r4 -> contains the glcd coordinates where the coin
                ;is suppose to fell once the user presses select
```

```
                ;r5 -> stores the memory mapped address value of any
                ;given GLCD coordinates

                ;r6 -> stores the memory mapped address the currently
                ;filled element

                ;r7 -> stores the count value for score
                ;related subroutines

                ;bit 00h -> column where coin is to be placed is
                ;seletcted this bit is used to indicate that

                ;bit 01h -> to determine a given players turn

                ;bit 03h -> to determine if get_memory_mapped
                ;subroutines is in use

                ;also for ram location 30h to 59h the data in these
                ;memory locations are mapped to corresponding grid
                ;element coordinates of glcd

                ;data present in them means the follwing things:

                ;00h means the given grid element is void of any coin

                ;01h means the given grid element is filled with
                ;coin of player-1

                ;02h means the given grid element is filled with
                ;coin of player-2

        four_4_in_a_row:

                setb psw.4
                clr psw.3
                ;starting functions called once to setup
                ;the starting of game screen

                lcall display_grid_4IR
                lcall display_other_4IR ;display score and
                ;player whose turn is there
                lcall clr_ram_4IR

                setb 01h ;bit set to mark that starting
                ;turn will be of player 1

                mov r0,#30h ;r0 points to the memory mapped
```

```
                ;locations for the GLCD Screen
                ;Which contains information about
                ;type of data in a given grid cell

                mov r1,#22h ;r1 points to the location
                ;which contains the no. of filled cell of chosen col

                main_game_4IR:

                clr 00h ;is low until select is pressed till that
                ;player is deciding position where he/she
                ;has to fill coin

                wait_4IR:jnb 00h ,wait_4IR

                lcall coin_drop_4IR;once position is selected by
                ;given player select is pressed then first coin
                ;has to get in chosen column

                lcall update_cursor_4IR ; based on change that turn
                ;of next player display changes in screen

                lcall score_check_4IR

                sjmp main_game_4IR

        clr_ram_4IR: ;used to clear 42(D) ram locations
        ;with 00h starting from 30h

                mov r7,#42 ;no. of ram locations to be cleared

                repeate_till_done_4IR:

                mov @r0,#00h ;initialize all the ram locations with
                ;00h which means all grid elements are empty

                inc r0
                djnz r7,repeate_till_done_4IR

                mov r0,#30h ;make sure to re-initialize r0 to
                ;correct default value

        ret ;for clr_ram

        display_grid_4IR:

                mov r7,#7 ;no. of columns
```

```
                mov r6,#6 ;no. of rows

                mov r5,#27h ;coordinate location from where
                ;1st element of grid starts

                nxt_4IR:

                        mov a,r5 ;display grind character at
                        ;location value present in r5

                        lcall choose_coord
                        mov dptr,#grid
                        lcall display_char

                        inc r5

                        djnz r7,nxt_4IR ;this is repeated  7-times
                        ;i.e. till a row is completed

                        mov a,r5 ;the following subtraction
                        ;is done because while the loop rotates
                        ;7-times an extra increment takes

                        clr c ;place when the condition turns
                        ;out to be false and hence to match things
                        ;properly we subtract 1 from value of r5 reg

                        subb a,#1

                        add a,#10h ;this is done to transverse to next row
                        anl a,#0f0h
                        add a,#07h
                        mov r5,a

                        mov r7,#7 ;reinitialize r7 to cover for the new row

                        djnz r6,nxt_4IR ;repeat the process till entire
                        ;grid is drawn on screen

                        mov a,#17h ;this is to place the coin of
                        ;initial player as cursor

                        mov r2,a
                        lcall choose_coord
                        mov dptr,#c_p1
                        lcall display_char
```

```
        ret; return for display grid

        display_other_4IR: ;used to display other useful
        ;info at start of the game

                mov a,#10h ;set the coordinates from where
                ;1st character of string is to be displayed

                lcall choose_coord
                mov dptr ,#player
                mov b,a ; first char location in b
                mov a,#6 ;total characters in a
                lcall display_string ;location of string
                ;first element in dptr

                mov a,#30h  ;this cursor position will
                ;change after each turn

                lcall choose_coord
                mov dptr ,#cursor
                lcall display_char ;

                mov a,#32h ;
                lcall choose_coord
                mov dptr ,#c_p1
                lcall display_char ;

                mov a,#52h ;
                lcall choose_coord
                mov dptr ,#c_p2
                lcall display_char ;

        ret; return for display_other

        coin_drop_4IR: ;deals with where to place
        ;the coin after select button is pressed

                mov a,r2 ;used to clear coin from
                ;current cursor location

                lcall choose_coord
                mov dptr,#clear ;now p2 turn has started
                ;thus his/her coin as cursor

                lcall display_char

                lcall fill_cell_4IR
```

```
            jb 01h,pl2_2 ; jump if 01h is set , if its set then p1
            ;has selected the position and coin has to go in that
            ;so grid coin of p1 but cursor coin of p2 as now p2
            ;turn is there seeing the logic it may be hard to digest
            ;it is because the 01h bit is complemented in keyboard
            ;isr as soon as select is pressed and because of that
            ;we follow an invereted logic here

            lcall choose_coord
            mov dptr,#coin1 ;grid coin of p1...as
            ;bit is clr means its turn is just over

            lcall display_char
            lcall update_ram_4IR

            mov a,r2
            lcall choose_coord
            mov dptr,#c_p2 ;now p2 turn has started
            ;thus his/her coin as cursor

            lcall display_char
            sjmp exit_coin_drop_4IR

            pl2_2:
            lcall choose_coord
            mov dptr,#coin2
            lcall display_char
            lcall update_ram_4IR

            mov a,r2
            lcall choose_coord
            mov dptr,#c_p1 ;now p1 turn has started
            ;thus his/her coin as cursor

            lcall display_char

      exit_coin_drop_4IR:
      ret;return for coin_drop

      fill_cell_4IR:;r1 points to 21h, 21h to 27h
      ;contains the value till which given column is filled

            mov a,r2; lower nibble of r2 contains
            ;the col which is selected ,

            anl a,#0fh ; col which is selected is now in a
```

```
add a,r1 ; reg-a points to the location which contains
;the information that how much cells are
;filled in chosen col
; say r2=18h => a=08h then a=>29h

clr c ;now this is done because the grid starts
;from 7th column till dth column for each row

subb a,#07h ;while r1 is to point form 21 to 28h
;only so to avoid the indexing issue

mov r1,a ;we subtract 7 from the masked
;column value so that correct
;ram loaction corresponding to selected
;column can be choosen

mov a,@r1 ;content of 38h in a
; say it is zero so last row will be filled

mov r3,a ;temp reg to store the content of 38h

;below 3 lines are logic to change the memory content

inc a ; inc the content in 38h
mov @r1,a ; save it in 38h
mov r1,#22h ; r1 again points to first location

;below is to set the cell positon
;which is to be filled in reg-a

mov a,#07h ; last row i.e row sixth is
;in page 7 so set that in accumulator

clr c

subb a,r3 ;r3 has content of how much
;the chosen col is filled upto,
;subtracting that from the last row so say two coins
;were filled in given col
;then the nxt coin has to be in 6-2=4th
;row that is 7-2=5th page

swap a ; as our upper nibble is for page(row)
;and lower for col so swapping is done,
;this which cell has to be filled its page
;or row location is set
```

```
                mov r3,a ; again temporarily store that in r3
                mov a,r2 ; lower nibble of r2 contains selected col
                anl a,#0fh ; so now col information is in accumulator's
                ;lower nibble and upper is masked off

                add a,r3; now accumulator contains which cell
                ;has to be filled i.e exact coordinates of
                ;the location to be filled

                mov r4,a ;to be used later in ram updating subroutines

        ret; for fill_cell

        update_ram_4IR: ;used to update ram memory locations
        ;correspoding to the grid filled by a given player

                mov a,r4 ;restore the currently filled
                ;cell location in reg-a

                get_mm:

                anl a,#0f0h ;mask the upper nibble to
                ;get row of the filled coin

                swap a

                cjne a,#2,check_for_row3 ;since row is 2 no need
                ;to update r0 to desired value

                mov r0,#30h ;this done to properly map given
                ;ram memory locations to grid coordinates

                sjmp done_selecting

                check_for_row3:

                cjne a,#3,check_for_row4
                mov r0,#37h

                sjmp done_selecting

                check_for_row4:

                cjne a,#4,check_for_row5
                mov r0,#3eh
                mov a,r4
```

```
                sjmp done_selecting

                check_for_row5:

                cjne a,#5,check_for_row6
                mov r0,#45h

                sjmp done_selecting

                check_for_row6:

                cjne a,#6,check_for_row7
                mov r0,#4ch

                sjmp done_selecting

                check_for_row7:

                cjne a,#7,exit_update_ram
                mov r0,#53h

                done_selecting:

                mov a,r4
                anl a,#0fh ;now we extract the column information
                clr c
                subb a,#07h ;since each element of a given row
                ;starts from 7th column of glcd

                add a,r0
                mov r0,a
                mov r6,a ;stores the memory mapped address
                ;the currently filled element

                jb 03h,exit_getmm

                jb 01h,updt_p2 ;to take apt decision as to
                ;what value is to be filled

                mov @r0,#01h ;player-1's coin is filled at
                ;corresponding memory location

                sjmp exit_update_ram

                updt_p2:

                mov @r0,#02h
```

```
exit_update_ram:

mov r0,#30h

ret ;for update_ram

get_memory_mapped:

        setb 03h ;set 03h bit high until this
        ;function is being executed

        sjmp get_mm

        exit_getmm:

        clr 03h ;clr the bit value once the job is over

        mov r5,a ;store the memory mapped address value
        ;of any given GLCD coordinates

ret ;get memory mapped

score_check_4IR:

        lcall check_vertical ;check for vertical game over
        ;possiblity with currently filled element

        lcall check_horizontal ;check for horizontal game over
        ;possiblity with currently filled element

ret ;for score_check

check_vertical: ;check the winning possiblity vertically

        mov a,r4; lower nibble of r2 contains
        ;the col which is selected ,

        anl a,#0f0h ; col which is selected is now in a
        swap a
        mov r7,a ;store the number temporarily in r7

        clr c
        subb a,#04h ;subtract from 4 because a minimum of
        ;four values are required to be filled to
        ;start vertical checking
```

```
                jc continue_vertical ; if after subtracttion
                ;carry is not set means
                ;the total rows filled are either
                ;greater than or equal to 4

                mov a,r7 ;if the above condition is false
                ;then we are still required
                ;to check fill the newly added element belongs
                ;to 4th row or not

                cjne a,#04h,exit_vertical_algorithm ;if reg-a
                ;is 4 then we continue to implement our
                ;algorithm else we'll exit saving processing time

                continue_vertical:

                mov a,r4 ;move the coordinates of recently filled
                ;GLCD Coordinates in reg-a

                lcall get_memory_mapped

                mov a,r5 ;move the type of data corresponding
                ;to coordinates in reg-a

                mov r0,a ;copy the correpsonding memory
                ;mapped value in ro

                mov a,@r0 ;now using r0 as pointer fetch the data
                ;stored in mapped memory adderss

                mov 1bh,a ;move the type of data corresponding to
                ;coordinates in 1bh (i.e. R3 of RB-3)

                mov a,r4 ;move the coordinates of recently filled
                ;GLCD Coordinates in reg-a

                add a,#10h ;since for every vertical check we have
                ;to go down a element to check and to go down
                ;by 1-Row We Increase by 10h;

                lcall get_memory_mapped

                mov a,r5 ;move the type of data
                ;corresponding to coordinates in reg-a
                mov r0,a
                mov a,@r0
                mov 1ch,a ;move the type of data corresponding
```

```
                ;to coordinates in 1ch (i.e. R4 of RB-3)

                mov a,r4 ;move the coordinates of recently
                ;filled GLCD Coordinates in reg-a

                add a,#20h ;since for every vertical
                ;check we have to go down
                ;a element to check and to go down
                ;by 1-Row We Increase by 10h;

                lcall get_memory_mapped

                mov a,r5 ;move the type of data corresponding
                ;to coordinates in reg-a

                mov r0,a
                mov a,@r0
                mov 1dh,a ;move the type of data corresponding
                ;to coordinates in 1ch (i.e. R5 of RB-3)

                mov a,r4 ;move the coordinates of recently filled
                ;GLCD Coordinates in reg-a

                add a,#30h ;since for every vertical
                ;check we have to go down a element to check
                ;and to go down by 1-Row We Increase by 10h;

                lcall get_memory_mapped

                mov a,r5 ;move the type of data corresponding
                ;to coordinates in reg-a

                mov r0,a
                mov a,@r0
                mov 1eh,a ;move the type of data corresponding to
                ;coordinates in 1ch (i.e. R5 of RB-3)

                mov a,1bh ;now copy the contents of 1bh in
                ;reg-a for comparision purposes

                cjne a,1ch,exit_vertical_algorithm ;compare reg-a with
                ;elements below it if their data type are same

                cjne a,1dh,exit_vertical_algorithm ;coninue to
                ;check for 3-values

                cjne a,1eh,exit_vertical_algorithm ;else exit at
```

```
                ;first different value

                ljmp game_over

                exit_vertical_algorithm:

                mov r0,#30h ;reinitialize r0 with default value for
                ;which it was to be used as pointer for other routines

        ret ;for check_vertical

        cmp:

                mov a,1bh ;now copy the contents of 1bh
                ;in reg-a for comparision purposes

                cjne a,1ch,exit_cmp ;compare reg-a with elements
                ;below it if their data type are same

                cjne a,1dh,exit_cmp ;coninue to check for 3-values
                cjne a,1eh,exit_cmp ;else exit at first different value

                ljmp game_over

                here:sjmp here
                exit_cmp:

        ret; cmp

        check_horizontal: ;check the winning
        ;possiblity horizontally

                mov a,r4; lower nibble of r4 contains
                ;the col which is selected ,

                anl a,#0fh ; col which is selected is now in acc

                ;check which col is selected and accordingly
                ;take decision

                ;c1 => check for 3 right col

                ;c2 => check for 3 left col

                ;c3 => check for 2 right cols and 1 Left

                ;c4 => check for 1 right col and 2 left
```

```
cjne a,#07h,ch8_4IR ;07 means 1st col is selected
;so only 3 right check possibility is there

        lcall c1_4IR
        sjmp exit_horizontal_algorithm

ch8_4IR:
cjne a,#08h,ch9_4IR

        lcall c1_4IR
        lcall c3_4IR
        sjmp exit_horizontal_algorithm

ch9_4IR:
cjne a,#09h,cha_4IR

        lcall c1_4IR
        lcall c3_4IR
        lcall c4_4IR
        sjmp exit_horizontal_algorithm

cha_4IR:

cjne a,#0ah,chb_4IR

        lcall c1_4IR
        lcall c2_4IR
        lcall c3_4IR
        lcall c4_4IR
        sjmp exit_horizontal_algorithm

chb_4IR:

cjne a,#0bh,chc_4IR

        lcall c2_4IR
        lcall c3_4IR
        lcall c4_4IR
        sjmp exit_horizontal_algorithm

chc_4IR:

cjne a,#0ch,chd_4IR

        lcall c2_4IR
        lcall c4_4IR
```

```
                        sjmp exit_horizontal_algorithm

                chd_4IR:
                cjne a,#07h,ch8_4IR

                        lcall c2_4IR

                exit_horizontal_algorithm:


                mov r0,#30h ;reinitialize r0 with default value
                ;for which it was to be used as pointer
                ;for other routines

        ret ;for check_vertical


        c1_4IR: ;3 Right

                mov a,r4
                lcall get_memory_mapped

                mov a,r5 ;move the type of data corresponding
                ;to coordinates in reg-a

                mov r0,a ;copy the correpsonding memory mapped
                ;value in ro

                mov a,@r0 ;now using r0 as pointer fetch the data
                ;stored in mapped memory adderss

                mov 1bh,a ; just filled cell's content
                ;(01/02) stored in 1bh

                mov a,r4
                lcall get_memory_mapped

                mov a,r5 ;move the type of data corresponding
                ;to coordinates in reg-a

                add a,#01h ;mem add of nxt col of same row

                mov r0,a ;copy the correpsonding memory
                ;mapped value in ro

                mov a,@r0 ;now using r0 as pointer fetch the data
                ;stored in mapped memory adderss
```

```
            mov 1ch,a ; nxt col filled cell's content (01/02/00)
            ;stored in 1ch

            mov a,r4

            lcall get_memory_mapped

            mov a,r5 ;move the type of data corresponding
            ;to coordinates in reg-a

            add a,#02h ; inc to two cols
            mov r0,a ;copy the correpsonding
            ;memory mapped value in ro

            mov a,@r0 ;now using r0 as pointer fetch the
            ;data stored in mapped memory adderss

            mov 1dh,a ; nxt to nxt col filled cell's
            ;content (01/02/00) stored in 1dh

            mov a,r4
            lcall get_memory_mapped

            mov a,r5 ;move the type of data corresponding
            ;to coordinates in reg-a

            add a,#03h
            mov r0,a ;copy the correpsonding memory mapped
            ;value in ro

            mov a,@r0 ;now using r0 as pointer fetch the data
            ;stored in mapped memory adderss

            mov 1eh,a ; nxt to nxt to next col filled cell's
            ;content (01/02/00) stored in 1eh

            lcall cmp

      ret; c1


      c2_4IR: ;3 Left

            mov a,r4
            lcall get_memory_mapped
```

```
mov a,r5 ;move the type of data corresponding
;to coordinates in reg-a

mov r0,a ;copy the correpsonding memory mapped
;value in ro

mov a,@r0 ;now using r0 as pointer fetch the data
;stored in mapped memory adderss

mov 1bh,a ;recently filled cell's content
;(01/02) stored in 1bh

mov a,r4
lcall get_memory_mapped

mov a,r5 ;move the type of data corresponding
;to coordinates in reg-a

clr c

subb a,#01h ; prev col meme mapped location in acc
mov r0,a ;copy the correpsonding memory
;mapped value in ro

mov a,@r0 ;now using r0 as pointer fetch the data stored
;in mapped memory adderss

mov 1ch,a; prev col filled cell's content
;(01/02/00) stored in 1ch

mov a,r4
lcall get_memory_mapped

mov a,r5 ;move the type of data corresponding
;to coordinates in reg-a
clr c
subb a,#02h
mov r0,a ;copy the correpsonding memory
;mapped value in ro

mov a,@r0 ;now using r0 as pointer fetch
;the data stored in mapped memory adderss

mov 1dh,a ;prev to prev col filled cell's
;content (01/02/00) stored in 1dh

mov a,r4
```

```
                lcall get_memory_mapped

                mov a,r5 ;move the type of data
                ;corresponding to coordinates in reg-a

                clr c
                subb a,#03h
                mov r0,a ;copy the correpsonding memory
                ;mapped value in ro

                mov a,@r0 ;now using r0 as pointer fetch
                ;the data stored in mapped memory adderss

                mov 1eh,a ; prev to prev to prev col filled
                ;cell's content (01/02/00) stored in 1dh

                lcall cmp

        ret;c2

        c3_4IR: ;2Right 1Left

                mov a,r4 ;move the coordinates of recently
                ;filled GLCD Coordinates in reg-a

                lcall get_memory_mapped

                mov a,r5 ;move the type of data corresponding
                ;to coordinates in reg-a

                mov r0,a ;copy the correpsonding memory
                ;mapped value in ro

                mov a,@r0 ;now using r0 as pointer fetch
                ;the data stored in mapped memory adderss

                mov 1bh,a ;move the type of data corresponding
                ;to coordinates in 1bh (i.e. R3 of RB-3)

                ;1L
                mov a,r4 ;move the coordinates of recently
                ;filled GLCD Coordinates in reg-a

                lcall get_memory_mapped

                mov a,r5 ;move the type of data corresponding
                ;to coordinates in reg-a
```

```
clr c
subb a,#01h ;since for every vertical
;check we have to go down a element to check
;and to go down by 1-Row We Increase by 10h

mov r0,a
mov a,@r0
mov 1ch,a ;move the type of data corresponding
;to coordinates in 1ch (i.e. R4 of RB-3)

;2R
mov a,r4 ;move the coordinates of recently
;filled GLCD Coordinates in reg-a

lcall get_memory_mapped

mov a,r5 ;move the type of data corresponding
;to coordinates in reg-a

add a,#01h ;since for every vertical check we have
;to go down a element to check and to go down
;by 1-Row We Increase by 10h;

mov r0,a
mov a,@r0
mov 1dh,a ;move the type of data corresponding
;to coordinates in 1ch (i.e. R5 of RB-3)

mov a,r4 ;move the coordinates of recently
;filled GLCD Coordinates in reg-a

lcall get_memory_mapped

mov a,r5 ;move the type of data corresponding
;to coordinates in reg-a

add a,#02h ;since for every vertical check we have
;to go down a element to check and to go down
;by 1-Row We Increase by 10h;

mov r0,a
mov a,@r0
mov 1eh,a ;move the type of data corresponding
;to coordinates in 1ch (i.e. R5 of RB-3)

lcall cmp
```

```
ret;c2

c4_4IR: ;2Left 1Right

        mov a,r4 ;move the coordinates of recently
        ;filled GLCD Coordinates in reg-a

        lcall get_memory_mapped

        mov a,r5 ;move the type of data corresponding
        ;to coordinates in reg-a

        mov r0,a ;copy the correpsonding memory
        ;mapped value in ro

        mov a,@r0 ;now using r0 as pointer fetch the
        ;data stored in mapped memory adderss

        mov 1bh,a ;move the type of data corresponding to
        ;coordinates in 1bh (i.e. R3 of RB-3)

        ;2L
        mov a,r4 ;move the coordinates of recently
        ;filled GLCD Coordinates in reg-a

        lcall get_memory_mapped

        mov a,r5 ;move the type of data corresponding
        ;to coordinates in reg-a

        clr c
        subb a,#01h ;since for every vertical check we have to
        ;go down a element to check and to go down by 1-Row
        ;We Increase by 10h

        mov r0,a
        mov a,@r0
        mov 1ch,a ;move the type of data corresponding to
        ;coordinates in 1ch (i.e. R4 of RB-3)

        mov a,r4 ;move the coordinates of recently
        ;filled GLCD Coordinates in reg-a

        lcall get_memory_mapped

        mov a,r5 ;move the type of data corresponding
```

```
                ;to coordinates in reg-a

                clr c
                subb a,#02h ;since for every vertical check we have to
                ;go down a element to check and to go down by 1-Row
                ;We Increase by 10h

                mov r0,a
                mov a,@r0
                mov 1dh,a ;move the type of data corresponding to
                ;coordinates in 1ch (i.e. R5 of RB-3)

                ;1R
                mov a,r4 ;move the coordinates of recently
                ;filled GLCD Coordinates in reg-a

                lcall get_memory_mapped

                mov a,r5 ;move the type of data corresponding to
                ;coordinates in reg-a

                add a,#01h ;since for every vertical check we have to
                ;go down a element to check and to go down by
                ;1-Row We Increase by 10h

                mov r0,a
                mov a,@r0
                mov 1eh,a ;move the type of data corresponding
                ;to coordinates in 1ch (i.e. R5 of RB-3)

                lcall cmp

        ret;c4

        update_cursor_4IR:

                jnb 01h ,p12_sel

                ;if 01h bit is high the new turn is of player-1
                ;and we set the cursor accordingly and if not the
                ;new turn will be player-2 and again curosor there
                ;will be set accordingly

                mov a,#50h
                lcall choose_coord
                mov dptr ,#clear
                lcall display_char
```

```
                mov a,#30h
                lcall choose_coord
                mov dptr ,#cursor
                lcall display_char

                sjmp exit_update_screen

                pl2_sel:
                mov a,#30h
                lcall choose_coord
                mov dptr ,#clear
                lcall display_char

                mov a,#50h
                lcall choose_coord
                mov dptr ,#cursor
                lcall display_char

                exit_update_screen:

        ret;return for update_cursor

end
```

ESP-32 Code for IoT Integration

Since This is a implementation yet to be explored in future the authors requests that one should not see it as final code but as a rather snippet of what is possible in future implementations.

```
#include <WiFi.h>
#include <HTTPClient.h>

#define SCORE_PIN 4  // GPIO4 connected to 8051 output pin

const char* ssid = "YourSSID";
const char* password = "YourPASSWORD";
const char* firebase_url = "https://your-firebase-database.com/game_scores.
    json";

void setup() {
    Serial.begin(115200);
    WiFi.begin(ssid, password);

    pinMode(SCORE_PIN, INPUT);  // ESP32 listens for a HIGH signal from
    8051

    while (WiFi.status() != WL_CONNECTED) {
        delay(500);
        Serial.print(".");
    }
    Serial.println("\nWiFi␣Connected!");
}

void loop() {
    if (digitalRead(SCORE_PIN) == HIGH) {  // 8051 triggers HIGH when score
     is ready
        delay(500);  // Small delay to ensure stable reading
        int score = readScoreFrom8051();  // Read the actual score from
    8051
        uploadScoreToCloud(score);
    }
}

// Function to read score from 8051 (using GPIOs)
int readScoreFrom8051() {
    int score = 0;

    // Assuming 8051 sends 8-bit score using GPIOs 18-25 on ESP32
    for (int i = 0; i < 8; i++) {
```

```
        score |= (digitalRead(18 + i) << i); // Read 8-bit score bit by bit
    }

    Serial.println("Score␣Received:␣" + String(score));
    return score;
}

// Function to upload score to Firebase
void uploadScoreToCloud(int score) {
    HTTPClient http;
    http.begin(firebase_url);
    http.addHeader("Content-Type", "application/json");

    String jsonPayload = "{\"score\":" + String(score) + "}";
    int httpResponseCode = http.POST(jsonPayload);
    http.end();

    Serial.print("Score␣uploaded,␣response:␣");
    Serial.println(httpResponseCode);
}
```

G-Yantra User Manual

Developed by

Tanvi A. Shah
&
Rudra D. Joshi

Contents

1. Introduction

1.1 Overview

G-Yantra is a handheld gaming console powered by an 8051 microcontroller with a 128x64 Graphical LCD. It offers classic games like Snake game and 4-in-a-Row with intuitive button controls. The console is designed for gaming enthusiasts and hobbyists interested in embedded systems and retro-style games.

G-Yantra serves as an educational project that demonstrates the capabilities of microcontrollers in gaming applications. It provides a practical learning experience in embedded programming, graphics rendering, and user interface design. The system is designed to be portable, energy-efficient, and easy to use, making it an ideal platform for both gaming and experimentation.

The device is built using minimal yet efficient hardware components, allowing users to explore low-level programming concepts such as interrupt handling, display interfacing, and memory management. By integrating user-friendly button controls and a simple menu system, G-Yantra ensures an engaging and accessible gaming experience.

Quick Summary

G-Yantra is a DIY gaming console featuring an 8051 microcontroller, a graphical LCD, and customizable game options. It is a compact and portable system designed for both entertainment and educational purposes, offering hands-on experience in embedded system development.

1.2 Key Features

- Optimized Assembly Programming: Efficiently coded in Assembly language for maximum performance on the AT89S52 microcontroller.
- Graphical LCD Interface: Uses a 128x64 GLCD with precise pixel control for smooth graphics rendering.
- Button-Based User Input: Directional and selection buttons for seamless game control and interaction.
- Rechargeable Battery Support: Provides a continuous power supply, ensuring uninterrupted gameplay.
- Preloaded Games: Snake game, 4-in-a-Row

2. Hardware Overview

2.1 System Components

Component	Description
Microcontroller	AT89S52
Display	128x64 GLCD
Input Controls	Directional buttons, Select, Start, A/B buttons
Power Source	Four, 3.3V Li-Ion Rechargeable Battery

Table 1: G-Yantra Hardware Specifications

2.2 Button Functions

The table below defines the functions of each button used in the games.

Button	Function
Up	Move Up
Down	Move Down
Left	Move Left
Right	Move Right
Select	Choose an Option
Start	Start/Pause Game
Action A/B	Perform Game Action

Table 2: Button Functions

2.2.1 Usage in GUI

- **Up, Down, Left, Right**: Controls the movement of the cursor.
- **Start**: Starts the game.
- **Select**: Used to select the game from the menu.

2.2.2 Usage in Snake Game

- **Up, Down, Left, Right**: Controls the movement of the snake.

2.2.3 Usage in 4 in a Row

- **Left, Right**: Moves the game piece horizontally across the columns.
- **Select**: Drops the piece into the selected column.

3. Getting Started

This section provides step-by-step instructions on powering the device on and off, as well as navigating through the menu to access different features and games.

3.1 Powering On

- Turn on the switch; the screen will light up.
- If the device does not power on, check the battery status and recharge if needed.
- Upon successful startup, the G-Yantra logo and developer's introduction will appear, followed by the main menu.

3.2 Navigating the Menu

The menu system allows users to access different features and games available on the device. Use the following buttons to navigate:

- Up/Down Buttons: Scroll through the available options.
- Select Button: Choose the highlighted option.
- Start Button: Starts a selected game.

3.3 Accessing Games

- After powering on, navigate to the Games section using the Up/Down buttons.
- Select the desired game using the Select button.
- Press Start to begin playing.

4. Games and Gameplay

This section provides an overview of the available games, their objectives, controls, and gameplay mechanics.

4.1 Snake Game

Objective: Control the snake to eat food and grow longer while avoiding collisions with itself and the screen boundaries.

Gameplay:

- The game begins with a small snake positioned on the screen.
- The player controls the snake's movement using the directional buttons.
- Food items appear randomly on the screen. Each time the snake eats food, its length increases.
- The game ends if the snake collides with itself or the screen edges.
- The player's score is based on the number of food items consumed.

Controls:

- Up Button – Move the snake up.
- Down Button – Move the snake down.
- Left Button – Move the snake left.
- Right Button – Move the snake right.

4.2 4-in-a-Row

Objective: Align four of your tokens in a row before your opponent does, either horizontally, vertically, or diagonally.

Gameplay:

- The game is played on a grid (7 columns × 6 rows).
- Players take turns dropping tokens into a column.
- The token will fall to the lowest available position in that column.
- The first player to align four of their tokens in a row, column, or diagonally wins.
- If the grid is completely filled without a winner, the game ends in a draw.

Controls:

- Left Button – Move the token selection left.
- Right Button – Move the token selection right.
- Select Button – Drop the token into the selected column.

Game Modes:

- **Player vs. Player:** Two players take turns placing their tokens.

Both games provide engaging and competitive gameplay experiences while testing the player's strategic thinking and reflexes.

5. Troubleshooting

If you encounter any issues while using G-Yantra, refer to the table below for common problems and their solutions.

Issue	Solution
Display not turning on	Ensure the power switch is turned on. Check if the battery is charged and properly connected. If the issue persists, try resetting the system using the reset button provided on the board.
Buttons not responding	Check for any loose connections in the wiring. If the buttons still do not respond, reset the controls using the reset button provided on the board.
Game freezing	Restart the console by turning it off and on again. If the problem persists, ensure the firmware is correctly installed and up to date.
Screen flickering	Verify the power supply and ensure a stable connection. If using a battery, check its charge level and replace it if necessary.

Table 3: Common Issues and Fixes

If the issue is not listed above or persists after following the suggested solutions, refer to the user manual or seek technical support.

6. Technical Specifications

Specification	**Details**
Microcontroller	AT89S52
Display	128x64 Graphical LCD (GLCD)
Power	four 3.3V Rechargeable Battery (Li-ion/Li-Po)
Programming Language	Assembly Language with optimized routines for efficiency
Clock Speed	11.0592 MHz (External Crystal)
Memory	Internal 8KB ROM, 256B RAM
Input Controls	8 Button Interface (4-way directional, 2 action, 2 function buttons)
Operating Voltage	5V DC regulated supply
Charging Port	Type-C

Table 4: Technical Specifications

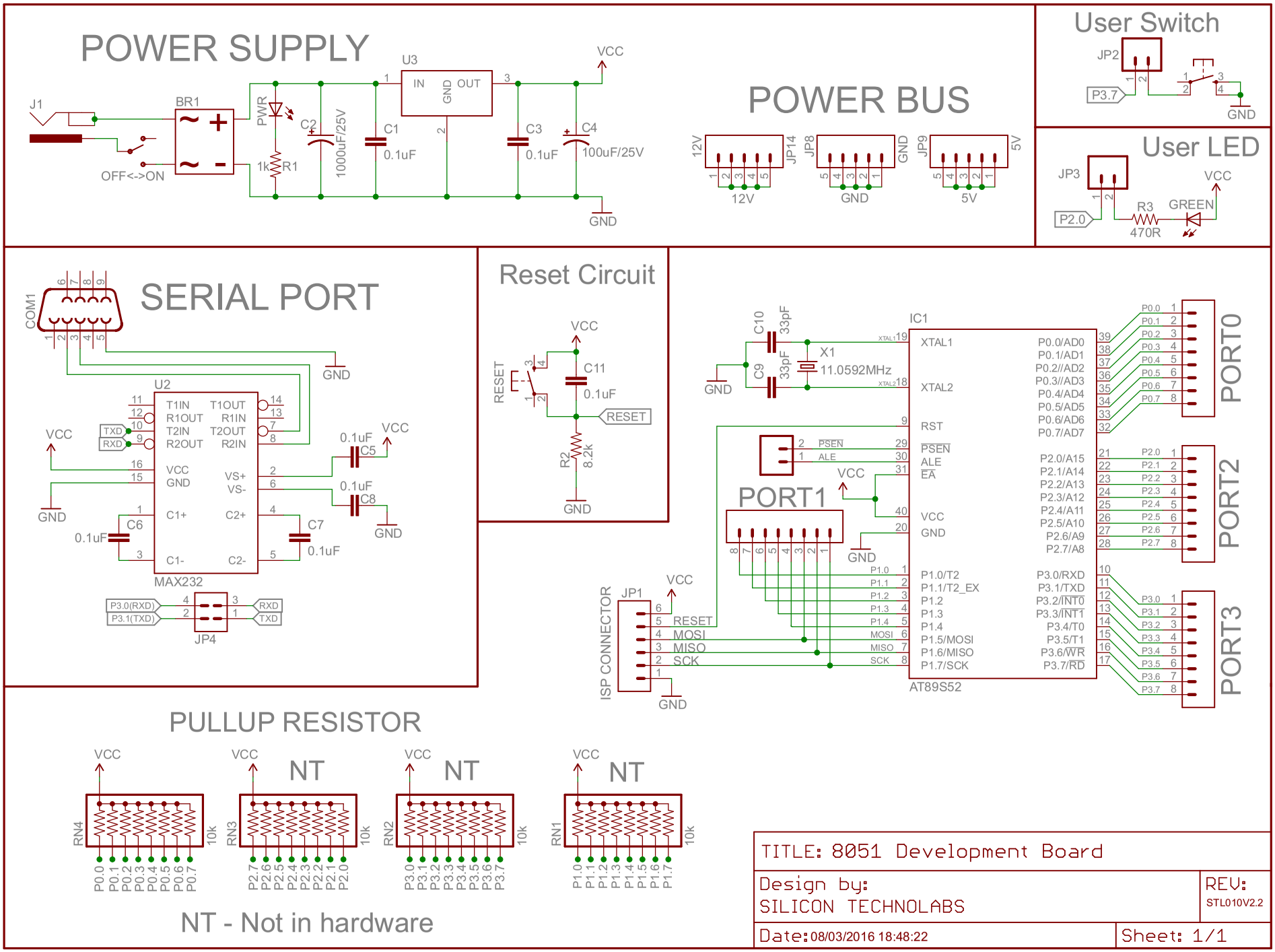
7. Contact & Support

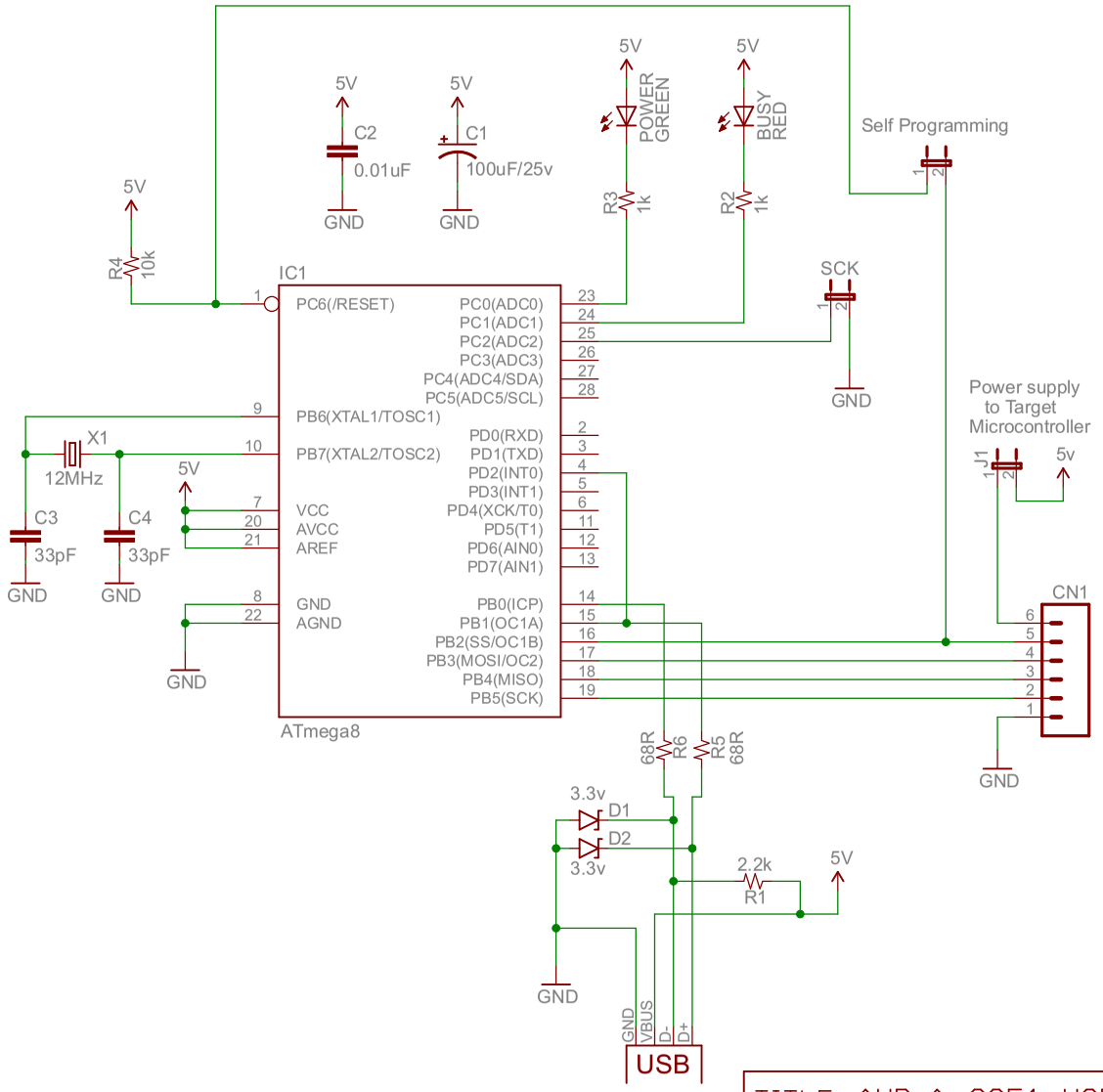
For technical support or inquiries, please contact the developers:

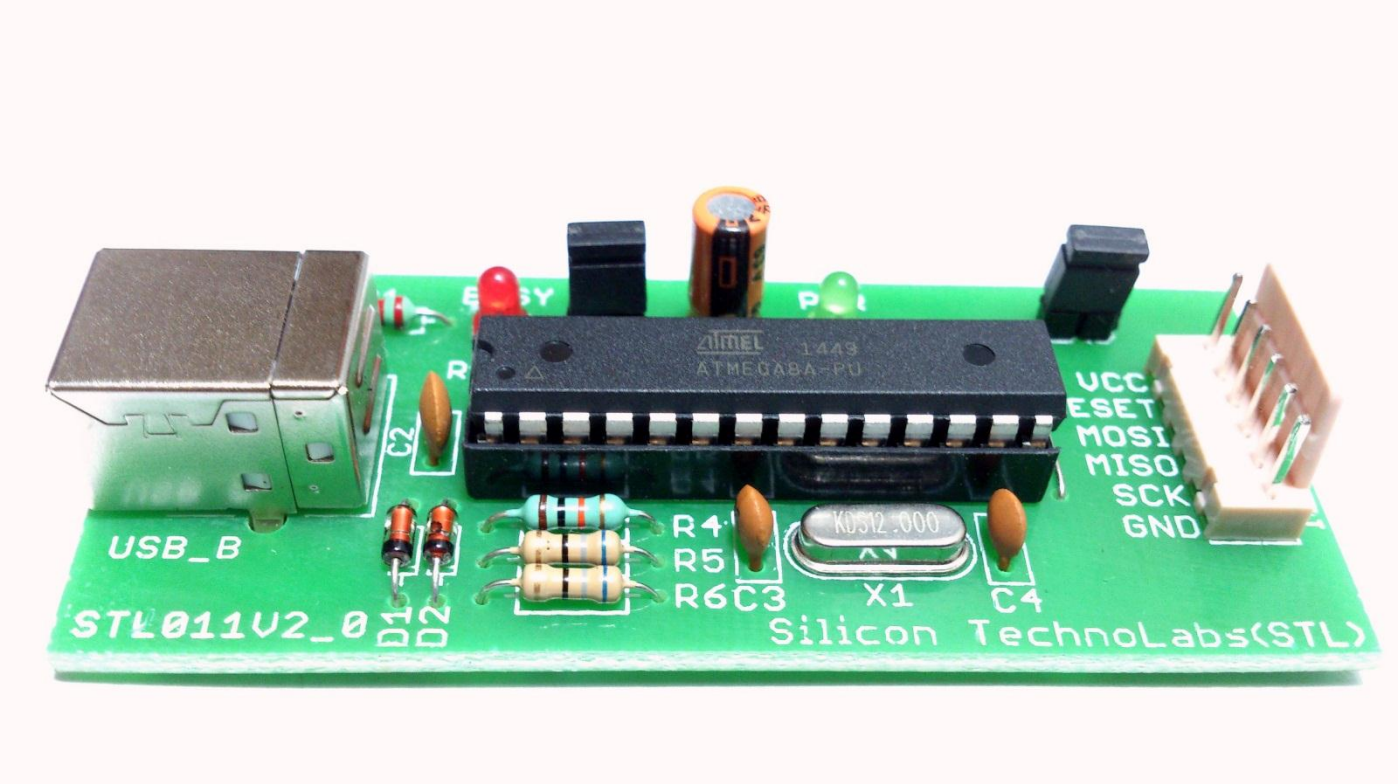
Rudra Joshi rudrajoshi002@gmail.com
Tanvi Shah shahtanvi2906@gmail.com

Necessary Circuit Schematics of Development Board Used in the Project and The Hardware that was soldered are as follows:

1. **8051 Development Board Circuit Schematic**
2. **AVR & 8051 USB ISP Programmer Circuit Schematic**
3. **AVR & 8051 USB ISP Programmer User Manual**
4. **Soldered Hardware Circuit Schematic**


POWER SUPPLY
J1
BR1
OFF<->ON
PWR
1k
R1
C2
1000uF/25V
U3
IN
GND
OUT
C1
0.1uF
C3
0.1uF
C4
100uF/25V
VCC
GND
POWER BUS
JP14
12V
JP8
GND
JP9
5V
User Switch
JP2
P3.7
GND
User LED
JP3
P2.0
R3
470R
GREEN
VCC
SERIAL PORT
COM1
GND
U2
T1IN
T1OUT
R1OUT
R1IN
T2IN
T2OUT
R2OUT
R2IN
VCC
GND
VS+
VS-
C1+
C2+
C1-
C2-
MAX232
TXD
RXD
C5
C8
C7
C6
0.1uF
P3.0(RXD)
P3.1(TXD)
JP4
Reset Circuit
RESET
C11
R2
8.2k
IC1
XTAL1
XTAL2
RST
PSEN
ALE
EA
C10
C9
33pF
X1
11.0592MHz
PORT1
PORT0
PORT2
PORT3
P0.0/AD0
P0.1/AD1
P0.2//AD2
P0.3//AD3
P0.4/AD4
P0.5/AD5
P0.6/AD6
P0.7/AD7
P2.0/A15
P2.1/A14
P2.2/A13
P2.3/A12
P2.4/A11
P2.5/A10
P2.6/A9
P2.7/A8
P1.0/T2
P1.1/T2_EX
P1.2
P1.3
P1.4
P1.5/MOSI
P1.6/MISO
P1.7/SCK
P3.0/RXD
P3.1/TXD
P3.2/INT0
P3.3/INT1
P3.4/T0
P3.5/T1
P3.6/WR
P3.7/RD
AT89S52
JP1
ISP CONNECTOR
MOSI
MISO
SCK
PULLUP RESISTOR
RN4
RN3
RN2
RN1
10k
NT
NT - Not in hardware
TITLE: 8051 Development Board
Design by:
SILICON TECHNOLABS
REV:
STL010V2.2
Date: 08/03/2016 18:48:22
Sheet: 1/1


5V
C2
0.01uF
GND
5V
C1
100uF/25v
GND
5V
POWER
GREEN
R3
1k
5V
BUSY
RED
R2
1k
Self Programming
5V
R4
10k
IC1
PC6(/RESET)
PC0(ADC0)
PC1(ADC1)
PC2(ADC2)
PC3(ADC3)
PC4(ADC4/SDA)
PC5(ADC5/SCL)
SCK
GND
PB6(XTAL1/TOSC1)
PD0(RXD)
PD1(TXD)
PD2(INT0)
PD3(INT1)
PD4(XCK/T0)
PD5(T1)
PD6(AIN0)
PD7(AIN1)
X1
12MHz
PB7(XTAL2/TOSC2)
5V
VCC
AVCC
AREF
C3
33pF
C4
33pF
GND
GND
GND
GND
AGND
PB0(ICP)
PB1(OC1A)
PB2(SS/OC1B)
PB3(MOSI/OC2)
PB4(MISO)
PB5(SCK)
ATmega8
Power supply
to Target
Microcontroller
J1
5v
CN1
GND
68R
R6
R5
68R
3.3v
D1
D2
3.3v
2.2k
R1
5V
GND
GND
VBUS
D-
D+
USB
TITLE: AVR & 8051 USB ISP Programmer
Design by:
SILICON TECHNOLABS
REV:
STL011V2.3
Date: 02-01-2016 14:55:40
Sheet: 1/1



USB
AVR and AT89Sxx
ISP Programmer

1. About USB AVR and AT89Sxx ISP Programmer

USB AVR and AT89Sxx ISP Programmer is low cost USB based programmer. This programmer will work with a wide variety of Atmel AVR and AT89Sxx microcontroller. They quite compact, but the design is really elegant. The USB interface is achieved by using an atmega8 processor and the rest is done in firmware.

2. Features

- Allows you to read or write the microcontroller flash, EEPROM, fuse bit and lock bits.
- Support for Windows, Mac OS X and Linux.
- SCK option to support targets with low clock speed (<1.5MHz).
- 5KB/sec Maximum write speed.
- There is 5V supply option for target so no need of any external supply.
- 6 pin polarized ISP interface.

3. Supported Software

- **AVRdude** - Version 5.2 or later. AVRdude is available for many platforms.(1)
- **Khazama AVR Programmer** - Windows XP/Vista GUI application for USBasp and avrdude.(1)
- **Progisp -** Windows GUI application for AVR and AT89Sxx.(2)

Note:

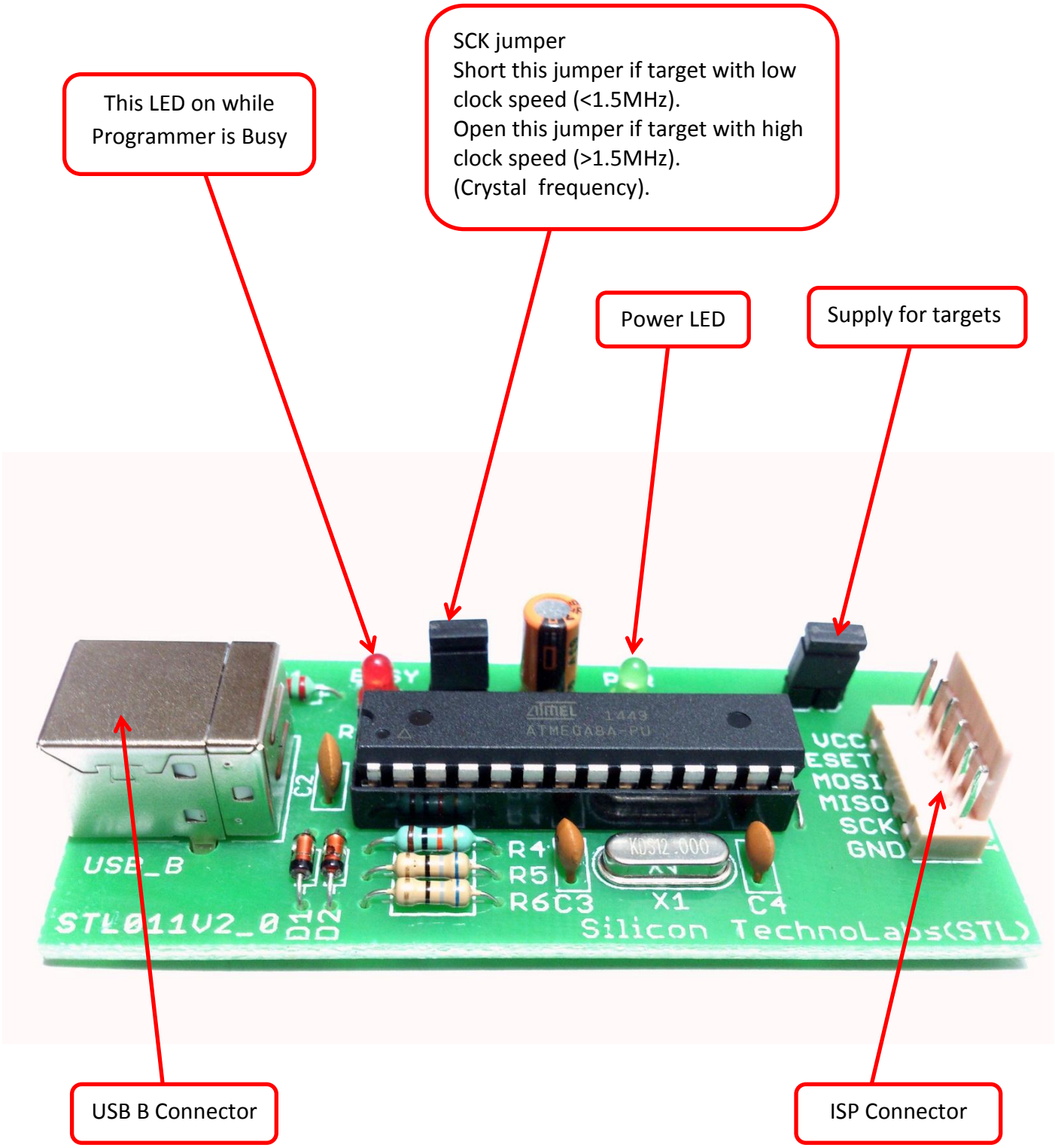
1. **Khazama AVR Programmer And AVRdude does not support AT89SXX.**
2. **Progisp support both AVR and AT89SXX controller.**

4. Specifications

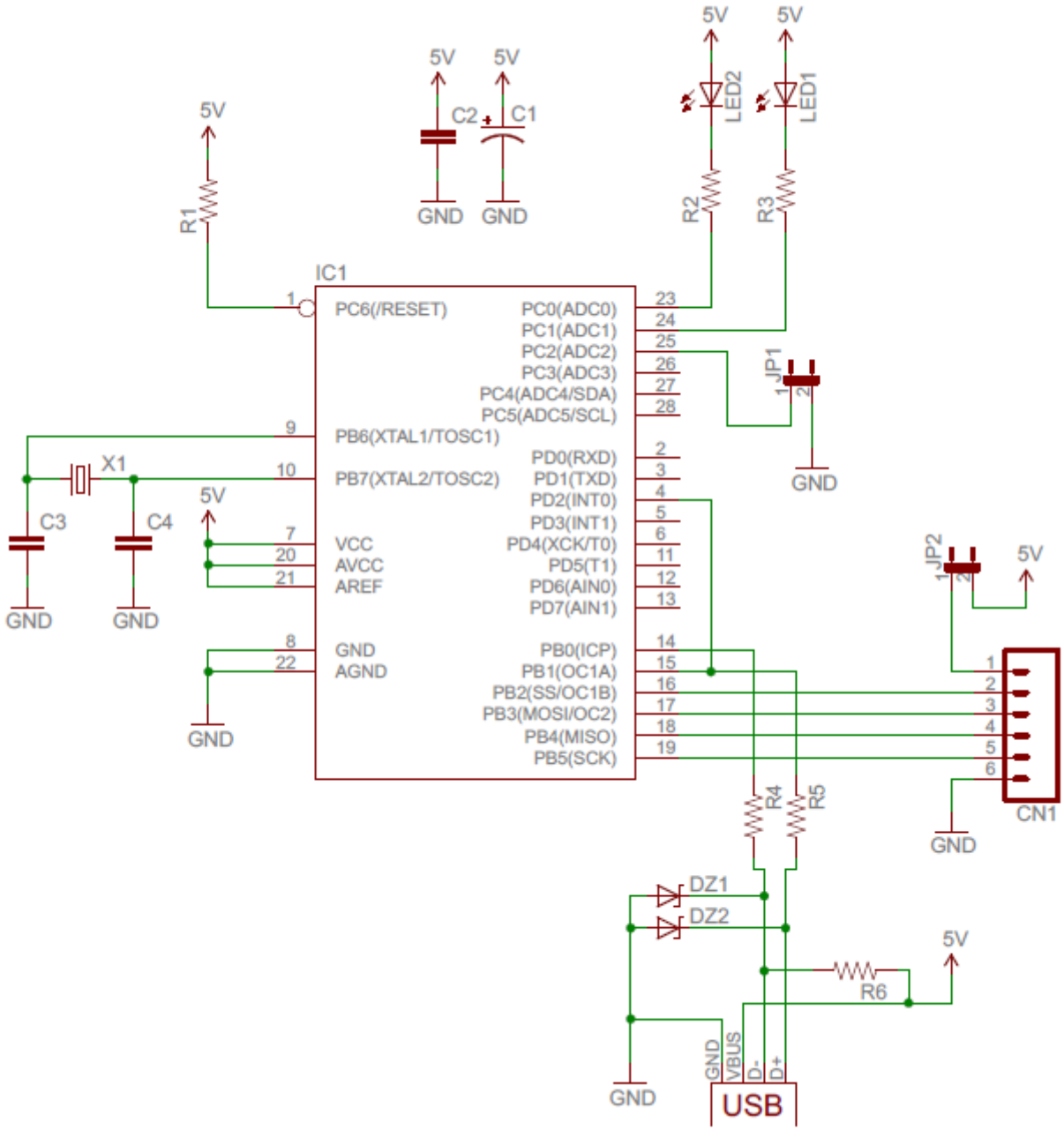
- Size: 74 x 37 x 12mm
- Supported Microcontroller:
-

ATmega Series				
ATmega8	ATmega8A	ATmega48	ATmega48A	ATmega48P
ATmega48PA	ATmega88	ATmega88A	ATmega88P	ATmega88PA
ATmega168	ATmega168A	ATmega168P	ATmega168PA	ATmega328
ATmega328P	ATmega103	ATmega128	ATmega128P	ATmega1280
ATmega1281	ATmega16	ATmega16A	ATmega161	ATmega162
ATmega163	ATmega164	ATmega164A	ATmega164P	ATmega164PA
ATmega169	ATmega169A	ATmega169P	ATmega169PA	ATmega2560
ATmega2561	ATmega32	ATmega32A	ATmega324	ATmega324A
ATmega324P	ATmega324PA	ATmega329	ATmega329A	ATmega329P
ATmega329PA	ATmega3290	ATmega3290A	ATmega3290P	ATmega64
ATmega64A	ATmega640	ATmega644	ATmega644A	ATmega644P
ATmega644PA	ATmega649	ATmega649A	ATmega649P	ATmega6490
ATmega6490A	ATmega6490P	ATmega8515	ATmega8535	
Tiny Series				
ATtiny12	ATtiny13	ATtiny13A	ATtiny15	ATtiny25
ATtiny26	ATtiny45	ATtiny85	ATtiny2313	ATtiny2313A
Classic Series				
AT90S1200	AT90S2313	AT90S2333	AT90S2343	AT90S4414
AT90S4433	AT90S4434	AT90S8515		
AT90S8535				
CAN Series				
AT90CAN128				
PWM Series				
AT90PWM2	AT90PWM3			
AT89Sxx Series				
AT89S51	AT89S52	AT89S53		

5. Hardware Details



6. Schematics



7. Installation

In order to complete the installation, you need to follow several steps:

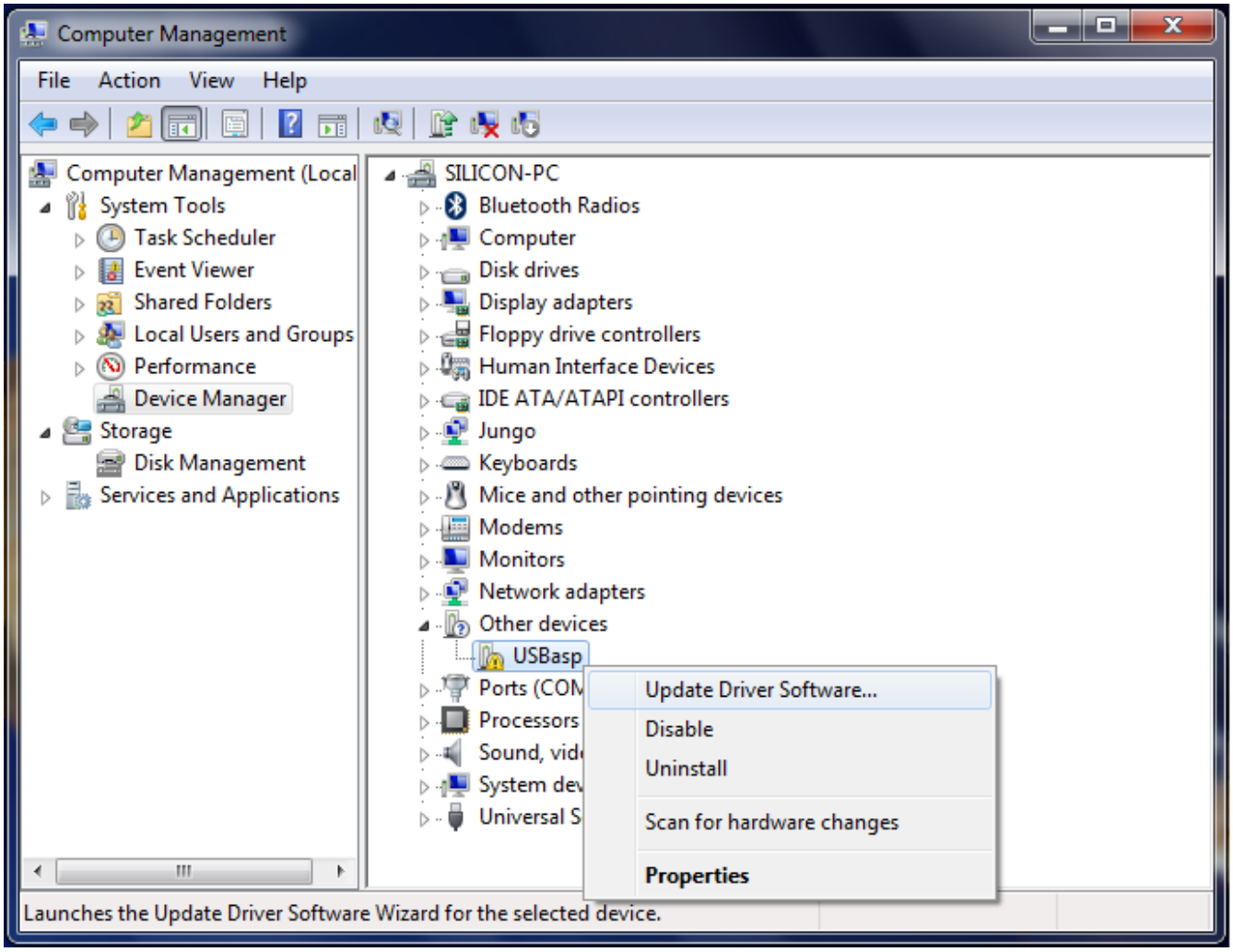
The first step is to connect the AVR & AT89Sxx ISP programmer to the USB port of your PC through USB-A to B cable. The AVR & AT89Sxx ISP programmer will work on a wide variety of operating systems, this procedure will only focus on Window 7.

Required items

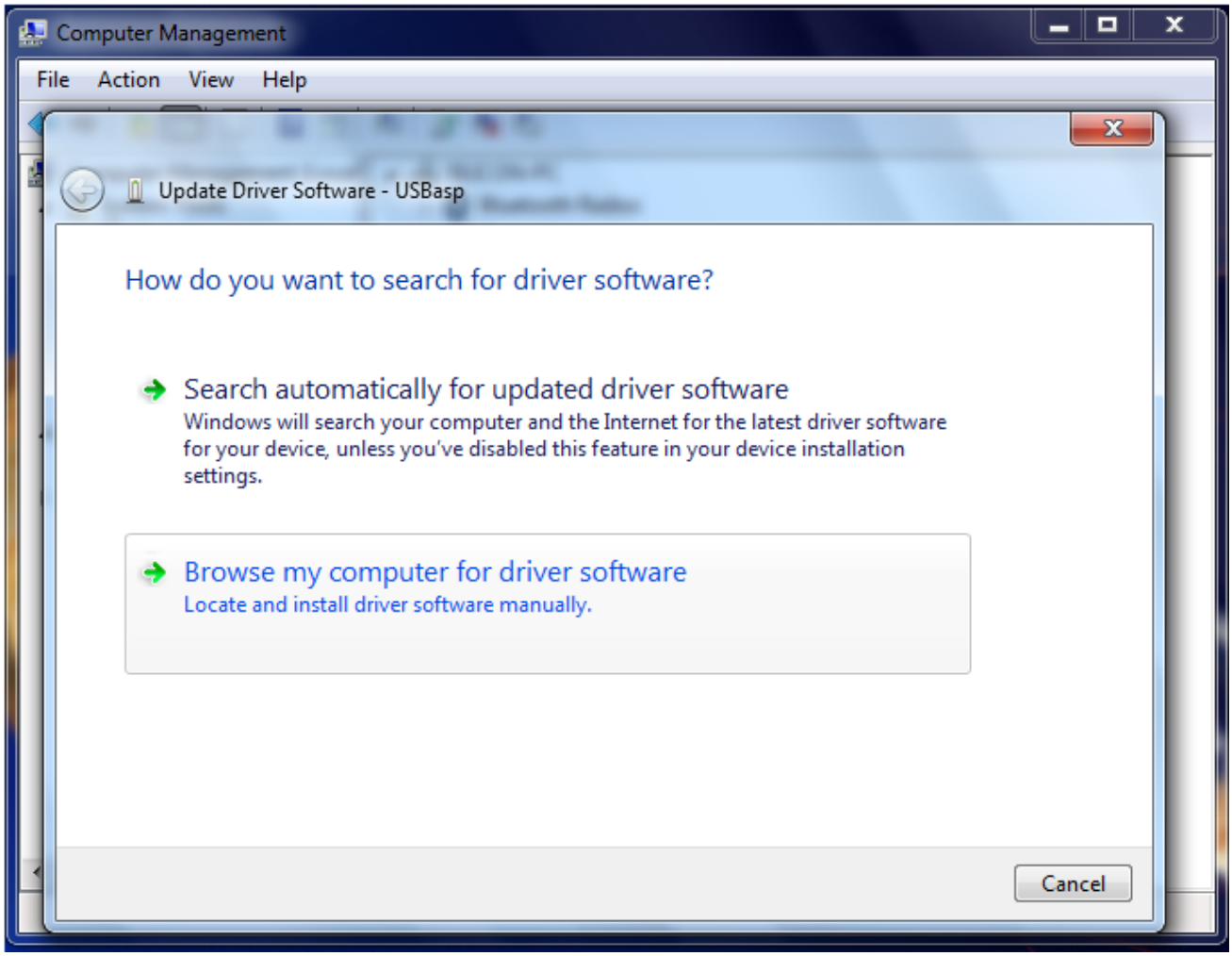
A. AVR & AT89Sxx ISP programmer.
B. USBasp drivers can be downloaded from here.

Procedure to install the AVR & AT89Sxx ISP programmer

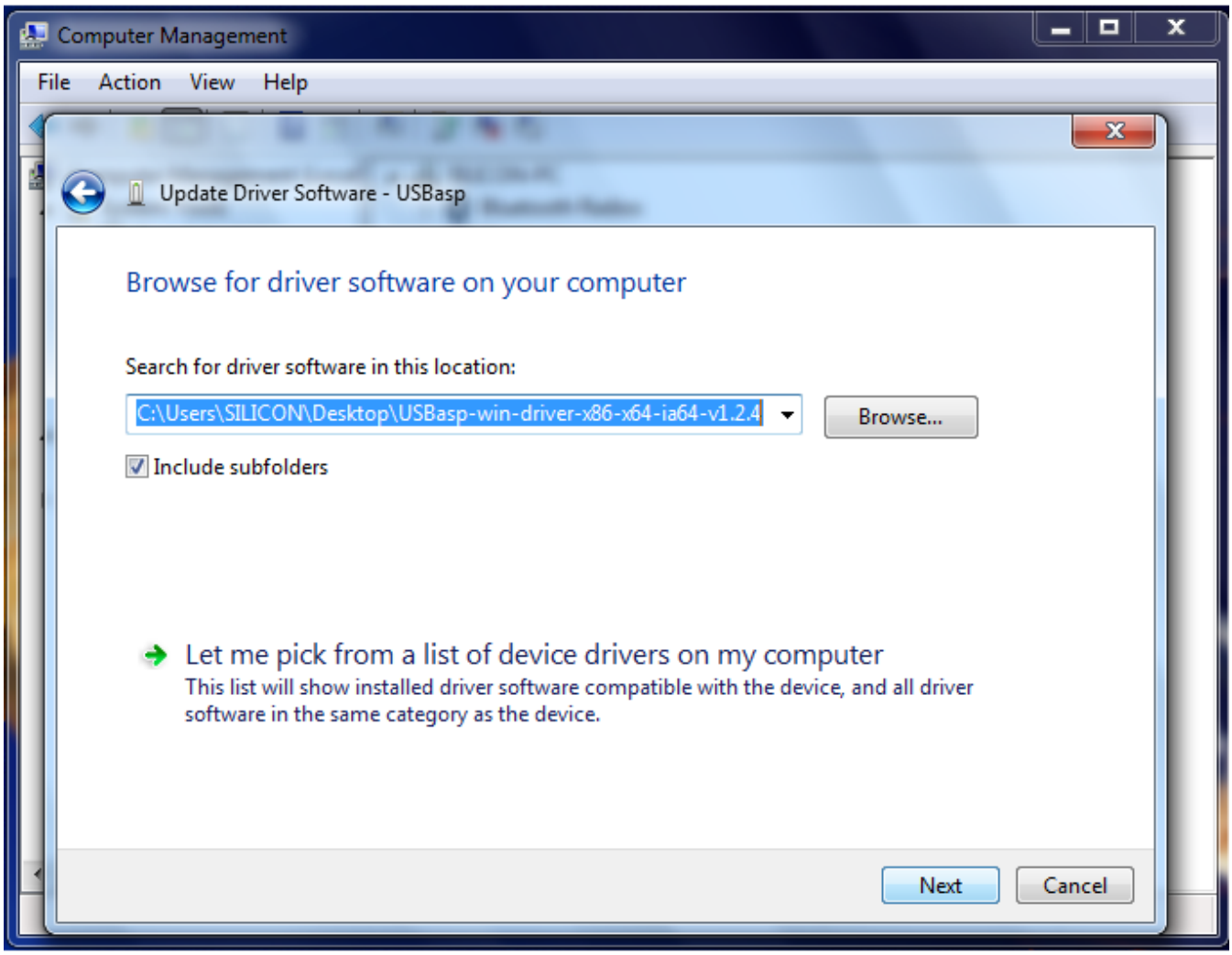
A. Connect programmer to available USB port in your PC through USB cable.
B. Go into the device manager and find the entry for the USBasp and it should be displayed with a yellow alert icon on it. Then right click on the device and select "**Update Driver Software**".



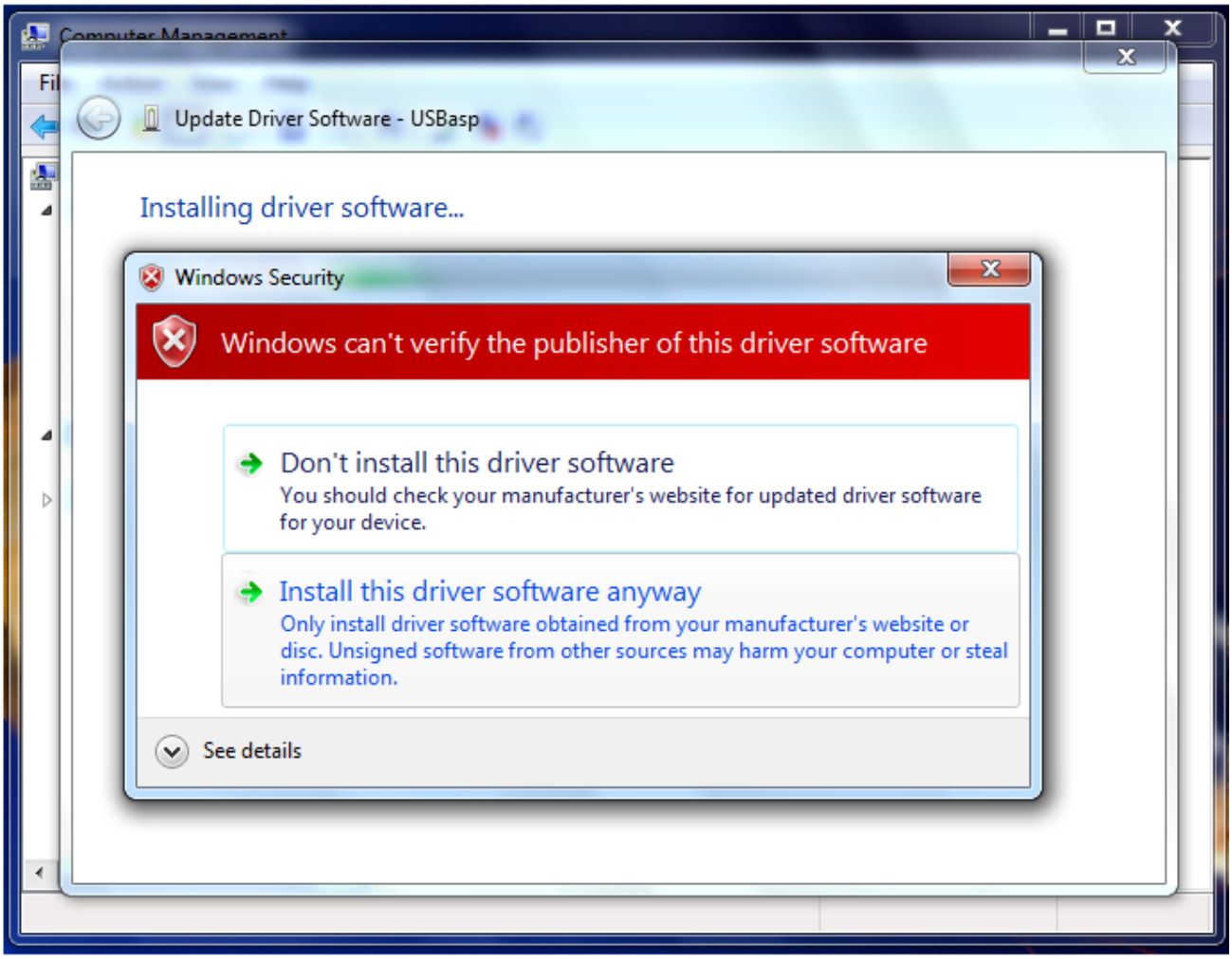
c. After you left click the "**Update Driver Software**", it will come out with "How **do you want to search for driver software?**" Then choose the second one which is "**Browse my computer for driver software**" and click into it.



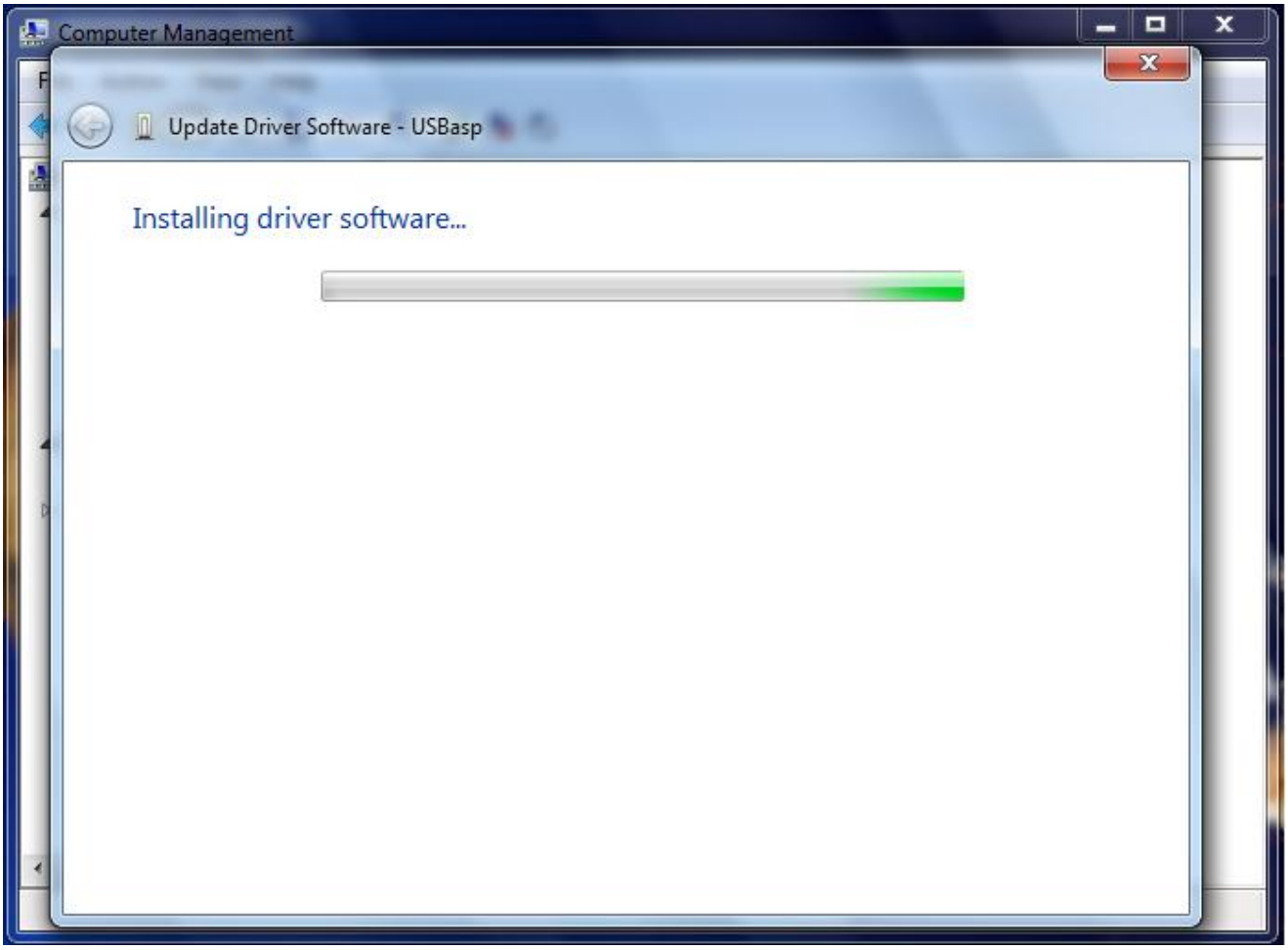
D. After that, you will see the screen which will prompt out "**Browse for driver software on your computer**". In this step, you need to select the folder where you unzipped the driver files then click "**Next**".



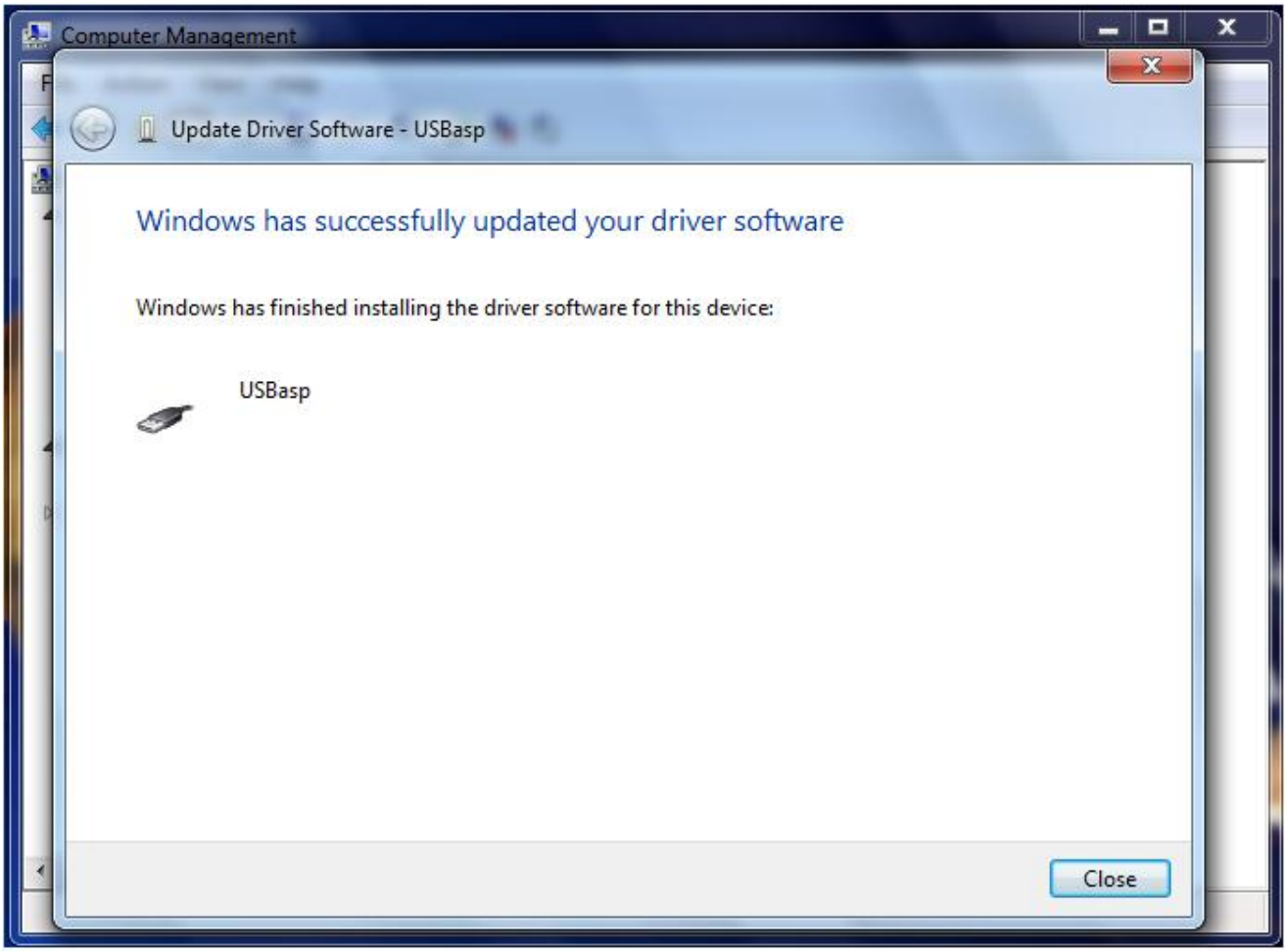
E. Next, the windows will prompt out a "**Windows Security**" with a red warning dialog. Do not worry about it, and just click "**Install this driver software anyway**" and the driver will install.

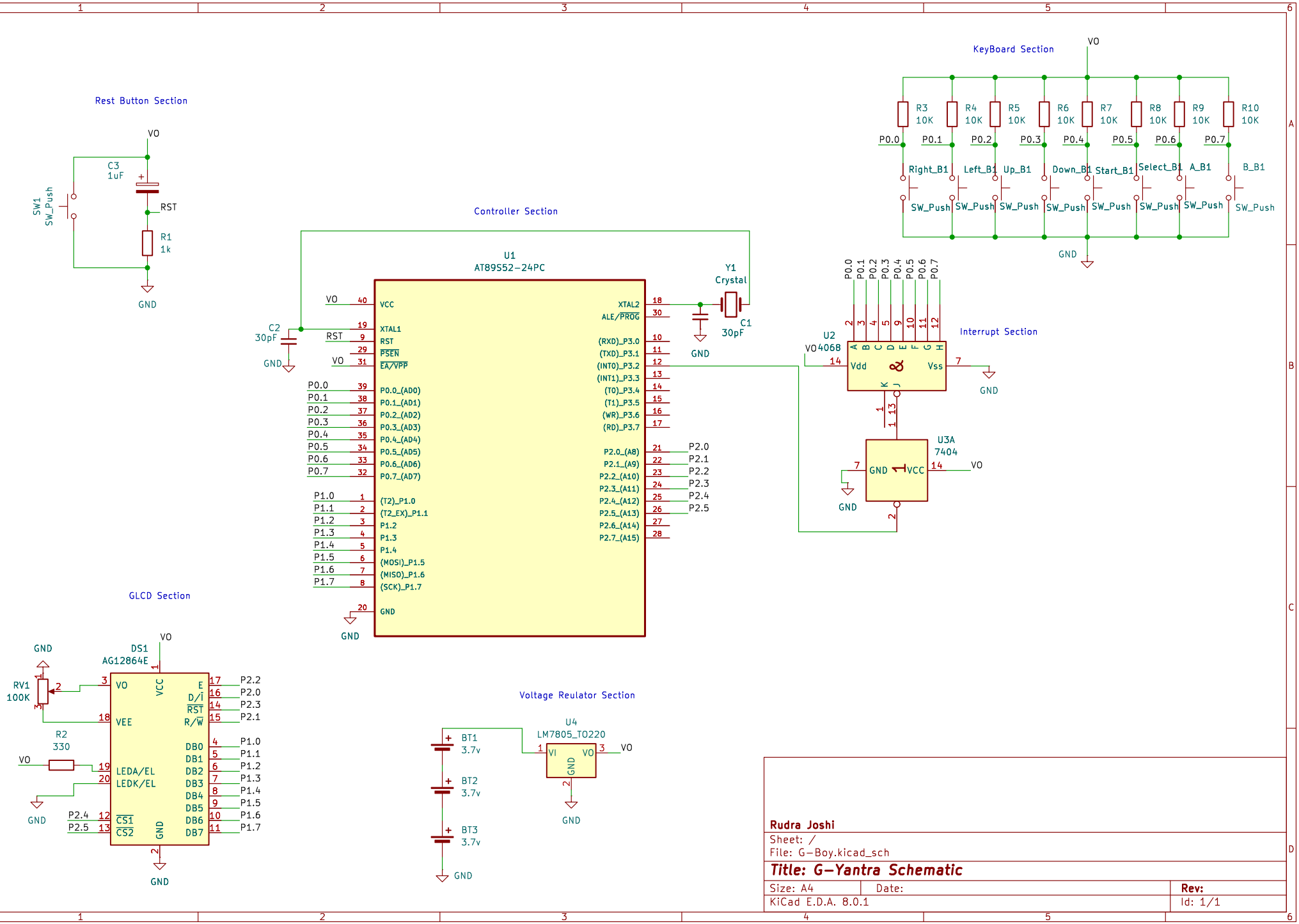


F. After click it, the next step is to wait a few seconds to let your computer to process the installation of driver software.



G. Now, you can use the Programmer to do the programming for the microcontroller.




Rest Button Section
VO
C3
1uF
RST
SW1
SW_Push
R1
1k
GND
KeyBoard Section
VO
R3 10K
R4 10K
R5 10K
R6 10K
R7 10K
R8 10K
R9 10K
R10 10K
P0.0
P0.1
P0.2
P0.3
P0.4
P0.5
P0.6
P0.7
Right_B1
Left_B1
Up_B1
Down_B1
Start_B1
Select_B1
A_B1
B_B1
SW_Push
GND
Controller Section
U1
AT89S52-24PC
Y1
Crystal
C2
30pF
C1
30pF
VCC
XTAL1
RST
PSEN
EA/VPP
XTAL2
ALE/PROG
(RXD)_P3.0
(TXD)_P3.1
(INT0)_P3.2
(INT1)_P3.3
(T0)_P3.4
(T1)_P3.5
(WR)_P3.6
(RD)_P3.7
P0.0_(AD0)
P0.1_(AD1)
P0.2_(AD2)
P0.3_(AD3)
P0.4_(AD4)
P0.5_(AD5)
P0.6_(AD6)
P0.7_(AD7)
(T2)_P1.0
(T2_EX)_P1.1
P1.2
P1.3
P1.4
(MOSI)_P1.5
(MISO)_P1.6
(SCK)_P1.7
P2.0_(A8)
P2.1_(A9)
P2.2_(A10)
P2.3_(A11)
P2.4_(A12)
P2.5_(A13)
P2.6_(A14)
P2.7_(A15)
U2
4068
Vdd
Vss
Interrupt Section
U3A
7404
GND
VCC
GLCD Section
DS1
AG12864E
RV1
100K
R2
330
VO
VEE
E
D/I
R/W
RST
CS1
CS2
LEDA/EL
LEDK/EL
DB0
DB1
DB2
DB3
DB4
DB5
DB6
DB7
Voltage Reulator Section
U4
LM7805_TO220
VI
VO
GND
BT1
3.7v
BT2
3.7v
BT3
3.7v
Rudra Joshi
Sheet: /
File: G-Boy.kicad_sch
Title: G-Yantra Schematic
Size: A4
Date:
Rev:
KiCad E.D.A. 8.0.1
Id: 1/1

Data Sheets of Relevant Components:

1. **AT89S52 (8051 Microcontroller) Datasheet**
2. **Atmel 8051 Microcontroller Instruction Set**
3. **128x64 KS0108 Graphics LCD Datasheet**
4. **IC 4068 (8-I/P Nand Gate) Datasheet**
5. **IC 7404 (Not Gate) Datasheet**

Features

- **Compatible with MCS-51® Products**
- **8K Bytes of In-System Programmable (ISP) Flash Memory**
  - **Endurance: 1000 Write/Erase Cycles**
- **4.0V to 5.5V Operating Range**
- **Fully Static Operation: 0 Hz to 33 MHz**
- **Three-level Program Memory Lock**
- **256 x 8-bit Internal RAM**
- **32 Programmable I/O Lines**
- **Three 16-bit Timer/Counters**
- **Eight Interrupt Sources**
- **Full Duplex UART Serial Channel**
- **Low-power Idle and Power-down Modes**
- **Interrupt Recovery from Power-down Mode**
- **Watchdog Timer**
- **Dual Data Pointer**
- **Power-off Flag**

Description

The AT89S52 is a low-power, high-performance CMOS 8-bit microcontroller with 8K bytes of in-system programmable Flash memory. The device is manufactured using Atmel's high-density nonvolatile memory technology and is compatible with the industry-standard 80C51 instruction set and pinout. The on-chip Flash allows the program memory to be reprogrammed in-system or by a conventional nonvolatile memory programmer. By combining a versatile 8-bit CPU with in-system programmable Flash on a monolithic chip, the Atmel AT89S52 is a powerful microcontroller which provides a highly-flexible and cost-effective solution to many embedded control applications.

The AT89S52 provides the following standard features: 8K bytes of Flash, 256 bytes of RAM, 32 I/O lines, Watchdog timer, two data pointers, three 16-bit timer/counters, a six-vector two-level interrupt architecture, a full duplex serial port, on-chip oscillator, and clock circuitry. In addition, the AT89S52 is designed with static logic for operation down to zero frequency and supports two software selectable power saving modes. The Idle Mode stops the CPU while allowing the RAM, timer/counters, serial port, and interrupt system to continue functioning. The Power-down mode saves the RAM contents but freezes the oscillator, disabling all other chip functions until the next interrupt or hardware reset.

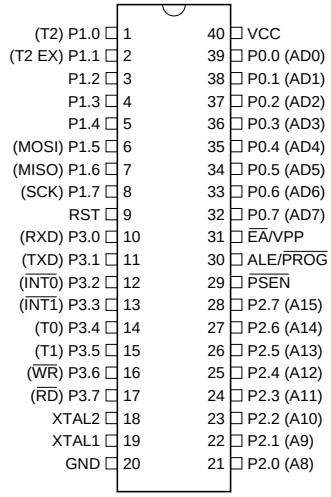
8-bit Microcontroller with 8K Bytes In-System Programmable Flash

AT89S52

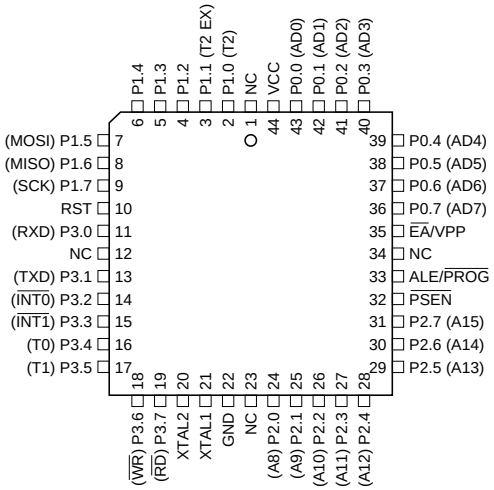
Rev. 1919A-07/01

Pin Configurations

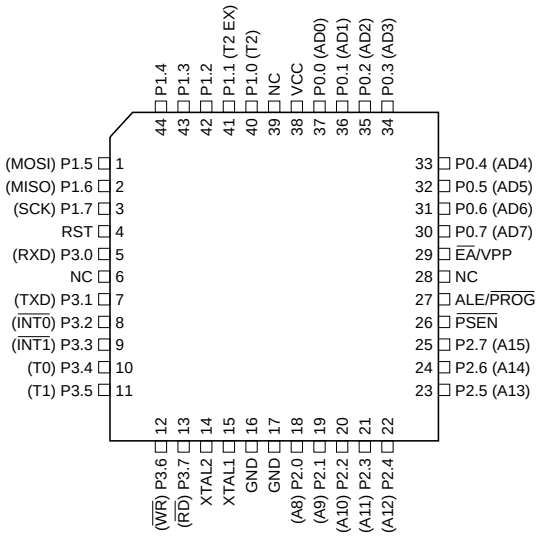
PDIP



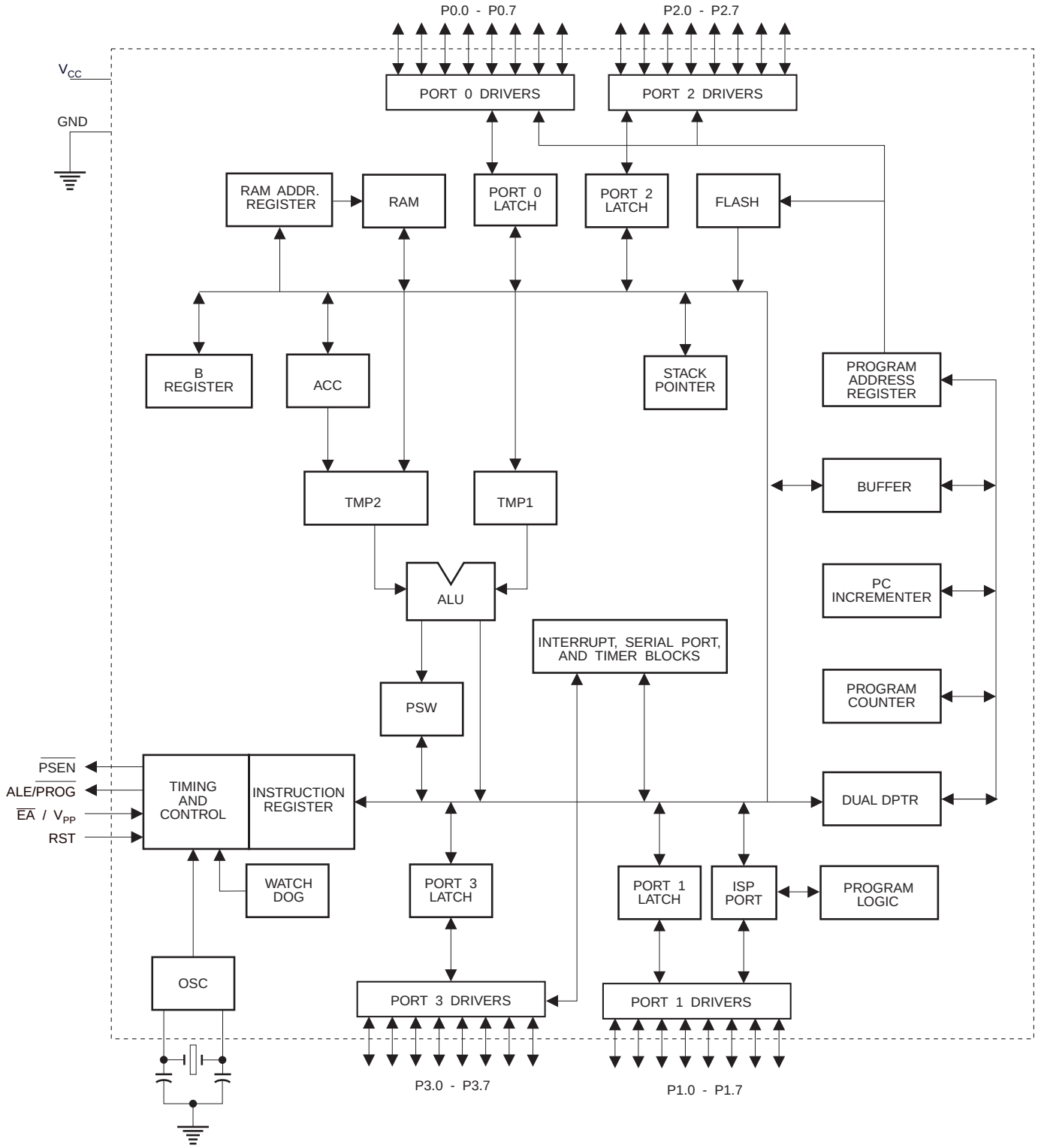
PLCC



TQFP



Block Diagram



Pin Description

VCC
Supply voltage.

GND
Ground.

Port 0
Port 0 is an 8-bit open drain bidirectional I/O port. As an output port, each pin can sink eight TTL inputs. When 1s are written to port 0 pins, the pins can be used as high-impedance inputs.

Port 0 can also be configured to be the multiplexed low-order address/data bus during accesses to external program and data memory. In this mode, P0 has internal pullups.

Port 0 also receives the code bytes during Flash programming and outputs the code bytes during program verification. External pullups are required during program verification.

Port 1
Port 1 is an 8-bit bidirectional I/O port with internal pullups. The Port 1 output buffers can sink/source four TTL inputs. When 1s are written to Port 1 pins, they are pulled high by the internal pullups and can be used as inputs. As inputs, Port 1 pins that are externally being pulled low will source current (I_{IL}) because of the internal pullups.

In addition, P1.0 and P1.1 can be configured to be the timer/counter 2 external count input (P1.0/T2) and the timer/counter 2 trigger input (P1.1/T2EX), respectively, as shown in the following table.

Port 1 also receives the low-order address bytes during Flash programming and verification.

Port Pin	Alternate Functions
P1.0	T2 (external count input to Timer/Counter 2), clock-out
P1.1	T2EX (Timer/Counter 2 capture/reload trigger and direction control)
P1.5	MOSI (used for In-System Programming)
P1.6	MISO (used for In-System Programming)
P1.7	SCK (used for In-System Programming)

Port 2
Port 2 is an 8-bit bidirectional I/O port with internal pullups. The Port 2 output buffers can sink/source four TTL inputs. When 1s are written to Port 2 pins, they are pulled high by the internal pullups and can be used as inputs. As inputs, Port 2 pins that are externally being pulled low will source current (I_{IL}) because of the internal pullups.

Port 2 emits the high-order address byte during fetches from external program memory and during accesses to external data memory that use 16-bit addresses (MOVX @ DPTR). In this application, Port 2 uses strong internal pullups when emitting 1s. During accesses to external data memory that use 8-bit addresses (MOVX @ RI), Port 2 emits the contents of the P2 Special Function Register.

Port 2 also receives the high-order address bits and some control signals during Flash programming and verification.

Port 3
Port 3 is an 8-bit bidirectional I/O port with internal pullups. The Port 3 output buffers can sink/source four TTL inputs. When 1s are written to Port 3 pins, they are pulled high by the internal pullups and can be used as inputs. As inputs, Port 3 pins that are externally being pulled low will source current (I_{IL}) because of the pullups.

Port 3 also serves the functions of various special features of the AT89S52, as shown in the following table.

Port 3 also receives some control signals for Flash programming and verification.

Port Pin	Alternate Functions
P3.0	RXD (serial input port)
P3.1	TXD (serial output port)
P3.2	$\overline{INT0}$ (external interrupt 0)
P3.3	$\overline{INT1}$ (external interrupt 1)
P3.4	T0 (timer 0 external input)
P3.5	T1 (timer 1 external input)
P3.6	$\overline{WR}$ (external data memory write strobe)
P3.7	$\overline{RD}$ (external data memory read strobe)

RST
Reset input. A high on this pin for two machine cycles while the oscillator is running resets the device. This pin drives High for 96 oscillator periods after the Watchdog times out. The DISRTO bit in SFR AUXR (address 8EH) can be used to disable this feature. In the default state of bit DISRTO, the RESET HIGH out feature is enabled.

ALE/$\overline{PROG}$
Address Latch Enable (ALE) is an output pulse for latching the low byte of the address during accesses to external memory. This pin is also the program pulse input ($\overline{PROG}$) during Flash programming.

In normal operation, ALE is emitted at a constant rate of 1/6 the oscillator frequency and may be used for external timing or clocking purposes. Note, however, that one ALE pulse is skipped during each access to external data memory.

If desired, ALE operation can be disabled by setting bit 0 of SFR location 8EH. With the bit set, ALE is active only during a MOVX or MOVC instruction. Otherwise, the pin is

weakly pulled high. Setting the ALE-disable bit has no effect if the microcontroller is in external execution mode.

$\overline{\text{PSEN}}$

Program Store Enable ($\overline{\text{PSEN}}$) is the read strobe to external program memory.

When the AT89S52 is executing code from external program memory, $\overline{\text{PSEN}}$ is activated twice each machine cycle, except that two $\overline{\text{PSEN}}$ activations are skipped during each access to external data memory.

$\overline{\text{EA}}$/VPP

External Access Enable. $\overline{\text{EA}}$ must be strapped to GND in order to enable the device to fetch code from external program memory locations starting at 0000H up to FFFFH. Note, however, that if lock bit 1 is programmed, $\overline{\text{EA}}$ will be internally latched on reset.

$\overline{\text{EA}}$ should be strapped to V_{CC} for internal program executions.

This pin also receives the 12-volt programming enable voltage (V_{PP}) during Flash programming.

XTAL1

Input to the inverting oscillator amplifier and input to the internal clock operating circuit.

XTAL2

Output from the inverting oscillator amplifier.

Table 1. AT89S52 SFR Map and Reset Values

0F8H									0FFH
0F0H	B 00000000								0F7H
0E8H									0EFH
0E0H	ACC 00000000								0E7H
0D8H									0DFH
0D0H	PSW 00000000								0D7H
0C8H	T2CON 00000000	T2MOD XXXXXX00	RCAP2L 00000000	RCAP2H 00000000	TL2 00000000	TH2 00000000			0CFH
0C0H									0C7H
0B8H	IP XX000000								0BFH
0B0H	P3 11111111								0B7H
0A8H	IE 0X000000								0AFH
0A0H	P2 11111111		AUXR1 XXXXXXX0				WDTRST XXXXXXXX		0A7H
98H	SCON 00000000	SBUF XXXXXXXX							9FH
90H	P1 11111111								97H
88H	TCON 00000000	TMOD 00000000	TL0 00000000	TL1 00000000	TH0 00000000	TH1 00000000	AUXR XXX00XX0		8FH
80H	P0 11111111	SP 00000111	DP0L 00000000	DP0H 00000000	DP1L 00000000	DP1H 00000000		PCON 0XXX0000	87H

Special Function Registers

A map of the on-chip memory area called the Special Function Register (SFR) space is shown in Table 1.

Note that not all of the addresses are occupied, and unoccupied addresses may not be implemented on the chip. Read accesses to these addresses will in general return random data, and write accesses will have an indeterminate effect.

User software should not write 1s to these unlisted locations, since they may be used in future products to invoke new features. In that case, the reset or inactive values of the new bits will always be 0.

Timer 2 Registers: Control and status bits are contained in registers T2CON (shown in Table 2) and T2MOD (shown in Table 3) for Timer 2. The register pair (RCAP2H, RCAP2L) are the Capture/Reload registers for Timer 2 in 16-bit capture mode or 16-bit auto-reload mode.

Interrupt Registers: The individual interrupt enable bits are in the IE register. Two priorities can be set for each of the six interrupt sources in the IP register.

Table 2. T2CON – Timer/Counter 2 Control Register

T2CON Address = 0C8H — Reset Value = 0000 0000B

Bit Addressable

Bit	TF2	EXF2	RCLK	TCLK	EXEN2	TR2	C/$\overline{T2}$	CP/$\overline{RL2}$
	7	6	5	4	3	2	1	0

Symbol	Function
TF2	Timer 2 overflow flag set by a Timer 2 overflow and must be cleared by software. TF2 will not be set when either RCLK = 1 or TCLK = 1.
EXF2	Timer 2 external flag set when either a capture or reload is caused by a negative transition on T2EX and EXEN2 = 1. When Timer 2 interrupt is enabled, EXF2 = 1 will cause the CPU to vector to the Timer 2 interrupt routine. EXF2 must be cleared by software. EXF2 does not cause an interrupt in up/down counter mode (DCEN = 1).
RCLK	Receive clock enable. When set, causes the serial port to use Timer 2 overflow pulses for its receive clock in serial port Modes 1 and 3. RCLK = 0 causes Timer 1 overflow to be used for the receive clock.
TCLK	Transmit clock enable. When set, causes the serial port to use Timer 2 overflow pulses for its transmit clock in serial port Modes 1 and 3. TCLK = 0 causes Timer 1 overflows to be used for the transmit clock.
EXEN2	Timer 2 external enable. When set, allows a capture or reload to occur as a result of a negative transition on T2EX if Timer 2 is not being used to clock the serial port. EXEN2 = 0 causes Timer 2 to ignore events at T2EX.
TR2	Start/Stop control for Timer 2. TR2 = 1 starts the timer.
C/$\overline{T2}$	Timer or counter select for Timer 2. C/$\overline{T2}$ = 0 for timer function. C/$\overline{T2}$ = 1 for external event counter (falling edge triggered).
CP/$\overline{RL2}$	Capture/Reload select. CP/$\overline{RL2}$ = 1 causes captures to occur on negative transitions at T2EX if EXEN2 = 1. CP/$\overline{RL2}$ = 0 causes automatic reloads to occur when Timer 2 overflows or negative transitions occur at T2EX when EXEN2 = 1. When either RCLK or TCLK = 1, this bit is ignored and the timer is forced to auto-reload on Timer 2 overflow.

Table 3a. AUXR: Auxiliary Register

AUXR Address = 8EH Reset Value = XXX00XX0B

Not Bit Addressable

	–	–	–	WDIDLE	DISRTO	–	–	DISALE
Bit	7	6	5	4	3	2	1	0

–	Reserved for future expansion	
DISALE	Disable/Enable ALE	
	DISALE	Operating Mode
	0	ALE is emitted at a constant rate of 1/6 the oscillator frequency
	1	ALE is active only during a MOVX or MOVC instruction
DISRTO	Disable/Enable Reset out	
	DISRTO	
	0	Reset pin is driven High after WDT times out
	1	Reset pin is input only
WDIDLE	Disable/Enable WDT in IDLE mode	
	WDIDLE	
	0	WDT continues to count in IDLE mode
	1	WDT halts counting in IDLE mode

Dual Data Pointer Registers: To facilitate accessing both internal and external data memory, two banks of 16-bit Data Pointer Registers are provided: DP0 at SFR address locations 82H-83H and DP1 at 84H-85H. Bit DPS = 0 in SFR AUXR1 selects DP0 and DPS = 1 selects DP1. The user should always initialize the DPS bit to the appropriate value before accessing the respective Data Pointer Register.

Power Off Flag: The Power Off Flag (POF) is located at bit 4 (PCON.4) in the PCON SFR. POF is set to “1” during power up. It can be set and rest under software control and is not affected by reset.

Table 3b. AUXR1: Auxiliary Register 1

AUXR1 Address = A2H Reset Value = XXXXXXX0B

Not Bit Addressable

	–	–	–	–	–	–	–	DPS
Bit	7	6	5	4	3	2	1	0

–	Reserved for future expansion	
DPS	Data Pointer Register Select	
	DPS	
	0	Selects DPTR Registers DP0L, DP0H
	1	Selects DPTR Registers DP1L, DP1H

Memory Organization

MCS-51 devices have a separate address space for Program and Data Memory. Up to 64K bytes each of external Program and Data Memory can be addressed.

Program Memory

If the $\overline{EA}$ pin is connected to GND, all program fetches are directed to external memory.

On the AT89S52, if $\overline{EA}$ is connected to V_{CC}, program fetches to addresses 0000H through 1FFFH are directed to internal memory and fetches to addresses 2000H through FFFFH are to external memory.

Data Memory

The AT89S52 implements 256 bytes of on-chip RAM. The upper 128 bytes occupy a parallel address space to the Special Function Registers. This means that the upper 128 bytes have the same addresses as the SFR space but are physically separate from SFR space.

When an instruction accesses an internal location above address 7FH, the address mode used in the instruction specifies whether the CPU accesses the upper 128 bytes of RAM or the SFR space. Instructions which use direct addressing access of the SFR space.

For example, the following direct addressing instruction accesses the SFR at location 0A0H (which is P2).

```
MOV 0A0H, #data
```

Instructions that use indirect addressing access the upper 128 bytes of RAM. For example, the following indirect addressing instruction, where R0 contains 0A0H, accesses the data byte at address 0A0H, rather than P2 (whose address is 0A0H).

```
MOV @R0, #data
```

Note that stack operations are examples of indirect addressing, so the upper 128 bytes of data RAM are available as stack space.

Watchdog Timer (One-time Enabled with Reset-out)

The WDT is intended as a recovery method in situations where the CPU may be subjected to software upsets. The WDT consists of a 13-bit counter and the Watchdog Timer Reset (WDTRST) SFR. The WDT is defaulted to disable from exiting reset. To enable the WDT, a user must write 01EH and 0E1H in sequence to the WDTRST register (SFR location 0A6H). When the WDT is enabled, it will increment every machine cycle while the oscillator is running. The WDT timeout period is dependent on the external clock frequency. There is no way to disable the WDT except through reset (either hardware reset or WDT overflow reset). When WDT overflows, it will drive an output RESET HIGH pulse at the RST pin.

Using the WDT

To enable the WDT, a user must write 01EH and 0E1H in sequence to the WDTRST register (SFR location 0A6H). When the WDT is enabled, the user needs to service it by writing 01EH and 0E1H to WDTRST to avoid a WDT overflow. The 13-bit counter overflows when it reaches 8191 (1FFFH), and this will reset the device. When the WDT is enabled, it will increment every machine cycle while the oscillator is running. This means the user must reset the WDT at least every 8191 machine cycles. To reset the WDT the user must write 01EH and 0E1H to WDTRST. WDTRST is a write-only register. The WDT counter cannot be read or written. When WDT overflows, it will generate an output RESET pulse at the RST pin. The RESET pulse duration is 96xTOSC, where TOSC=1/FOSC. To make the best use of the WDT, it should be serviced in those sections of code that will periodically be executed within the time required to prevent a WDT reset.

WDT During Power-down and Idle

In Power-down mode the oscillator stops, which means the WDT also stops. While in Power-down mode, the user does not need to service the WDT. There are two methods of exiting Power-down mode: by a hardware reset or via a level-activated external interrupt which is enabled prior to entering Power-down mode. When Power-down is exited with hardware reset, servicing the WDT should occur as it normally does whenever the AT89S52 is reset. Exiting Power-down with an interrupt is significantly different. The interrupt is held low long enough for the oscillator to stabilize. When the interrupt is brought high, the interrupt is serviced. To prevent the WDT from resetting the device while the interrupt pin is held low, the WDT is not started until the interrupt is pulled high. It is suggested that the WDT be reset during the interrupt service for the interrupt used to exit Power-down mode.

To ensure that the WDT does not overflow within a few states of exiting Power-down, it is best to reset the WDT just before entering Power-down mode.

Before going into the IDLE mode, the WDIDLE bit in SFR AUXR is used to determine whether the WDT continues to count if enabled. The WDT keeps counting during IDLE (WDIDLE bit = 0) as the default state. To prevent the WDT from resetting the AT89S52 while in IDLE mode, the user should always set up a timer that will periodically exit IDLE, service the WDT, and reenter IDLE mode.

With WDIDLE bit enabled, the WDT will stop to count in IDLE mode and resumes the count upon exit from IDLE.

UART

The UART in the AT89S52 operates the same way as the UART in the AT89C51 and AT89C52. For further information on the UART operation, refer to the ATMEL Web site (http://www.atmel.com). From the home page, select ‘Products’, then ‘8051-Architecture Flash Microcontroller’, then ‘Product Overview’.

Timer 0 and 1

Timer 0 and Timer 1 in the AT89S52 operate the same way as Timer 0 and Timer 1 in the AT89C51 and AT89C52. For further information on the timers’ operation, refer to the ATMEL Web site (http://www.atmel.com). From the home page, select ‘Products’, then ‘8051-Architecture Flash Microcontroller’, then ‘Product Overview’.

Timer 2

Timer 2 is a 16-bit Timer/Counter that can operate as either a timer or an event counter. The type of operation is selected by bit C/$\overline{T2}$ in the SFR T2CON (shown in Table 2). Timer 2 has three operating modes: capture, auto-reload (up or down counting), and baud rate generator. The modes are selected by bits in T2CON, as shown in Table 3. Timer 2 consists of two 8-bit registers, TH2 and TL2. In the Timer function, the TL2 register is incremented every machine cycle. Since a machine cycle consists of 12 oscillator periods, the count rate is 1/12 of the oscillator frequency.

Table 3. Timer 2 Operating Modes

RCLK +TCLK	CP/$\overline{RL2}$	TR2	MODE
0	0	1	16-bit Auto-reload
0	1	1	16-bit Capture
1	X	1	Baud Rate Generator
X	X	0	(Off)

In the Counter function, the register is incremented in response to a 1-to-0 transition at its corresponding external input pin, T2. In this function, the external input is sampled during S5P2 of every machine cycle. When the samples show a high in one cycle and a low in the next cycle, the count is incremented. The new count value appears in the register during S3P1 of the cycle following the one in which the transition was detected. Since two machine cycles (24 oscillator periods) are required to recognize a 1-to-0 transition, the maximum count rate is 1/24 of the oscillator frequency. To ensure that a given level is sampled at least once before it changes, the level should be held for at least one full machine cycle.

Capture Mode

In the capture mode, two options are selected by bit EXEN2 in T2CON. If EXEN2 = 0, Timer 2 is a 16-bit timer or counter which upon overflow sets bit TF2 in T2CON. This bit can then be used to generate an interrupt. If EXEN2 = 1, Timer 2 performs the same operation, but a 1-to-0 transition at external input T2EX also causes the current value in TH2 and TL2 to be captured into RCAP2H and RCAP2L, respectively. In addition, the transition at T2EX causes bit EXF2 in T2CON to be set. The EXF2 bit, like TF2, can generate an interrupt. The capture mode is illustrated in Figure 5.

Auto-reload (Up or Down Counter)

Timer 2 can be programmed to count up or down when configured in its 16-bit auto-reload mode. This feature is invoked by the DCEN (Down Counter Enable) bit located in the SFR T2MOD (see Table 4). Upon reset, the DCEN bit is set to 0 so that timer 2 will default to count up. When DCEN is set, Timer 2 can count up or down, depending on the value of the T2EX pin.

Figure 5. Timer in Capture Mode

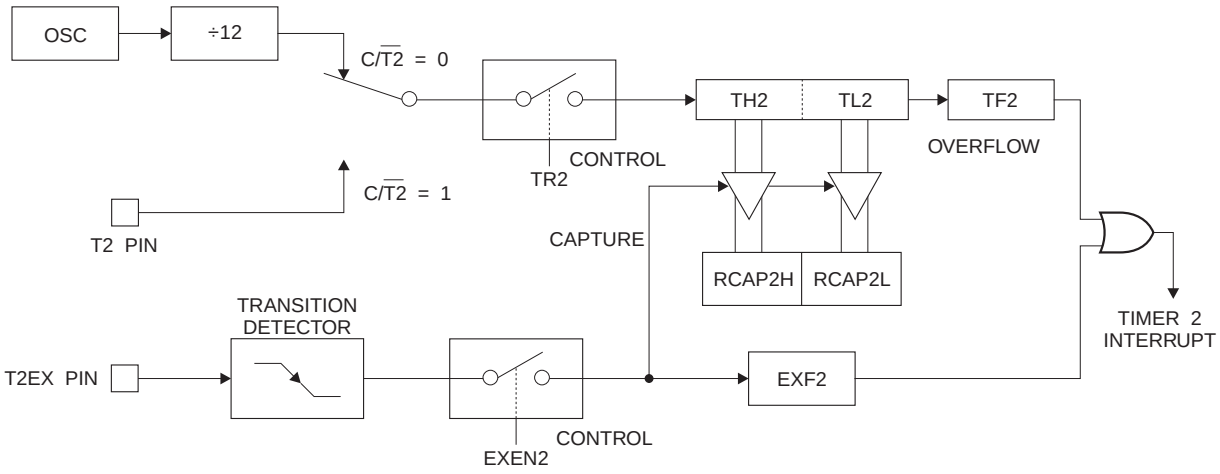


Figure 6 shows Timer 2 automatically counting up when DCEN=0. In this mode, two options are selected by bit EXEN2 in T2CON. If EXEN2 = 0, Timer 2 counts up to 0FFFFH and then sets the TF2 bit upon overflow. The overflow also causes the timer registers to be reloaded with the 16-bit value in RCAP2H and RCAP2L. The values in Timer in Capture ModeRCAP2H and RCAP2L are preset by software. If EXEN2 = 1, a 16-bit reload can be triggered either by an overflow or by a 1-to-0 transition at external input T2EX. This transition also sets the EXF2 bit. Both the TF2 and EXF2 bits can generate an interrupt if enabled.

Setting the DCEN bit enables Timer 2 to count up or down, as shown in Figure 6. In this mode, the T2EX pin controls the direction of the count. A logic 1 at T2EX makes Timer 2 count up. The timer will overflow at 0FFFFH and set the TF2 bit. This overflow also causes the 16-bit value in RCAP2H and RCAP2L to be reloaded into the timer registers, TH2 and TL2, respectively.

A logic 0 at T2EX makes Timer 2 count down. The timer underflows when TH2 and TL2 equal the values stored in RCAP2H and RCAP2L. The underflow sets the TF2 bit and causes 0FFFFH to be reloaded into the timer registers.

The EXF2 bit toggles whenever Timer 2 overflows or underflows and can be used as a 17th bit of resolution. In this operating mode, EXF2 does not flag an interrupt.

Figure 6. Timer 2 Auto Reload Mode (DCEN = 0)

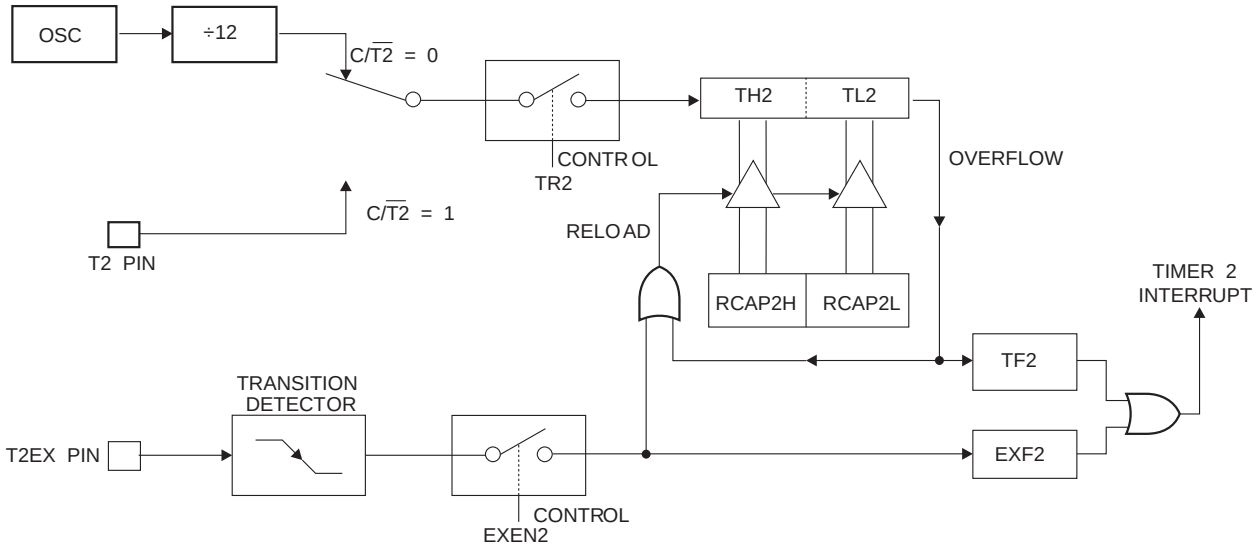


Table 4. T2MOD – Timer 2 Mode Control Register

T2MOD Address = 0C9H — Reset Value = XXXX XX00B

Not Bit Addressable

	–	–	–	–	–	–	T2OE	DCEN
Bit	7	6	5	4	3	2	1	0

Symbol	Function
–	Not implemented, reserved for future
T2OE	Timer 2 Output Enable bit
DCEN	When set, this bit allows Timer 2 to be configured as an up/down counter

Figure 7. Timer 2 Auto Reload Mode (DCEN = 1)

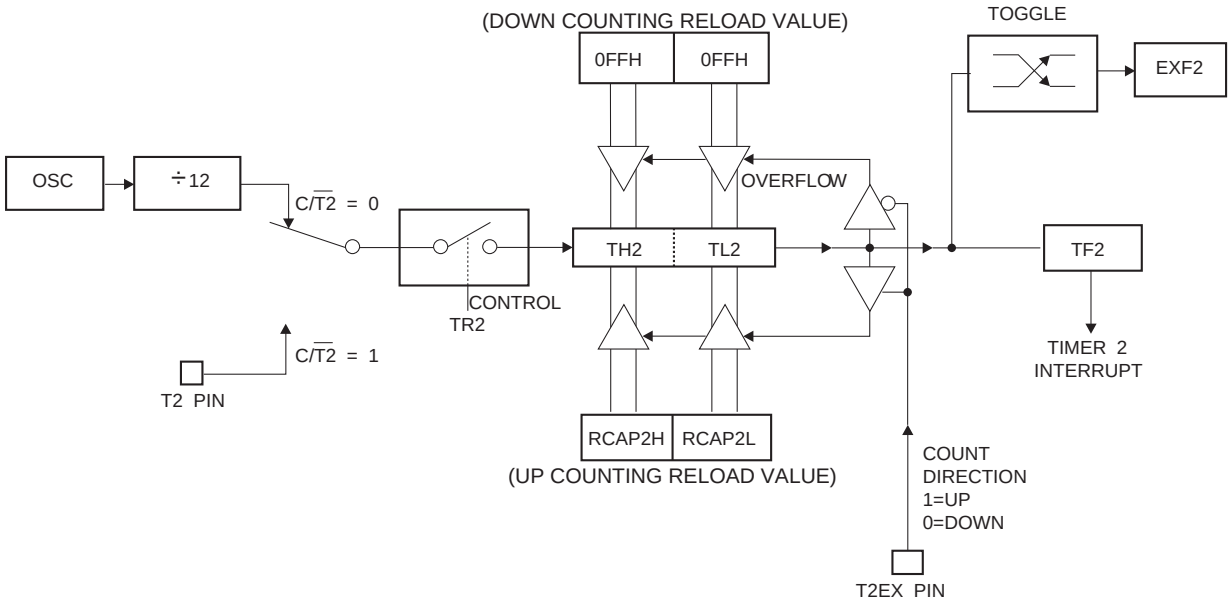
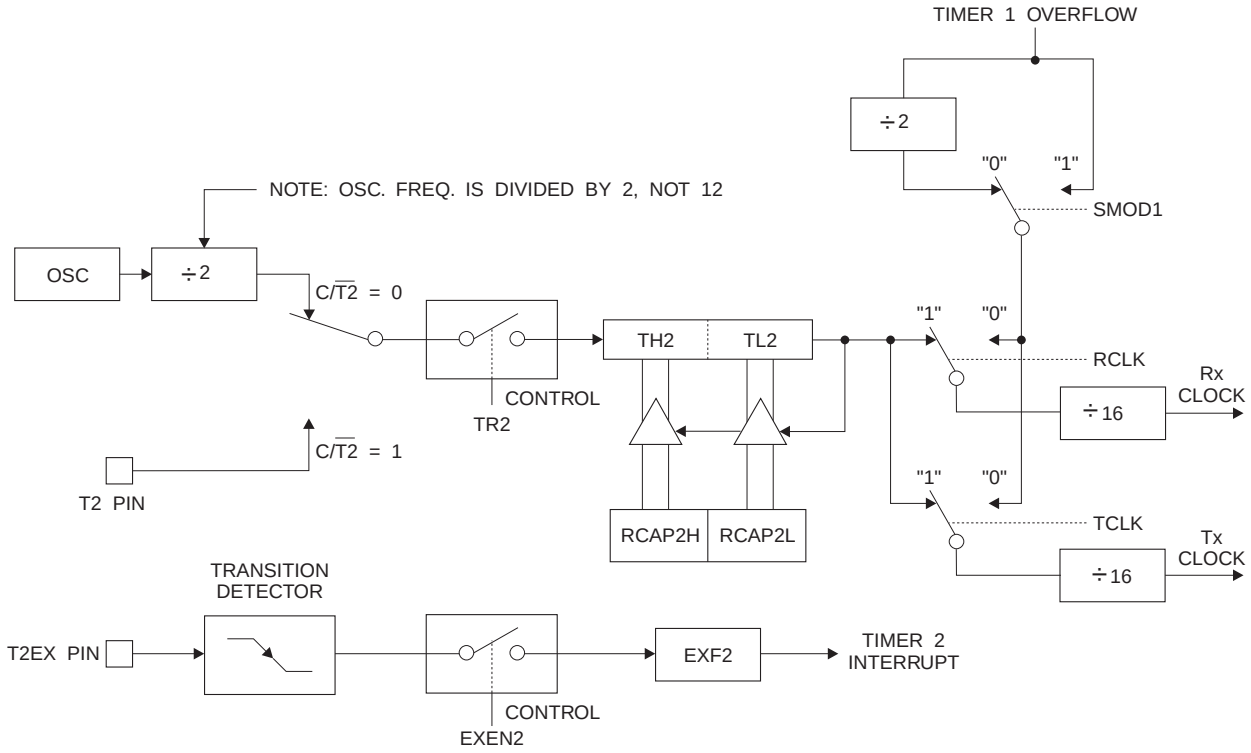


Figure 8. Timer 2 in Baud Rate Generator Mode



Baud Rate Generator

Timer 2 is selected as the baud rate generator by setting TCLK and/or RCLK in T2CON (Table 2). Note that the baud rates for transmit and receive can be different if Timer 2 is used for the receiver or transmitter and Timer 1 is used for the other function. Setting RCLK and/or TCLK puts Timer 2 into its baud rate generator mode, as shown in Figure 8.

The baud rate generator mode is similar to the auto-reload mode, in that a rollover in TH2 causes the Timer 2 registers to be reloaded with the 16-bit value in registers RCAP2H and RCAP2L, which are preset by software.

The baud rates in Modes 1 and 3 are determined by Timer 2's overflow rate according to the following equation.

$$\text{Modes 1 and 3 Baud Rates} = \frac{\text{Timer 2 Overflow Rate}}{16}$$

The Timer can be configured for either timer or counter operation. In most applications, it is configured for timer operation ($C/\overline{T2} = 0$). The timer operation is different for Timer 2 when it is used as a baud rate generator. Normally, as a timer, it increments every machine cycle (at 1/12 the oscillator frequency). As a baud rate generator, however, it increments every state time (at 1/2 the oscillator frequency). The baud rate formula is given below.

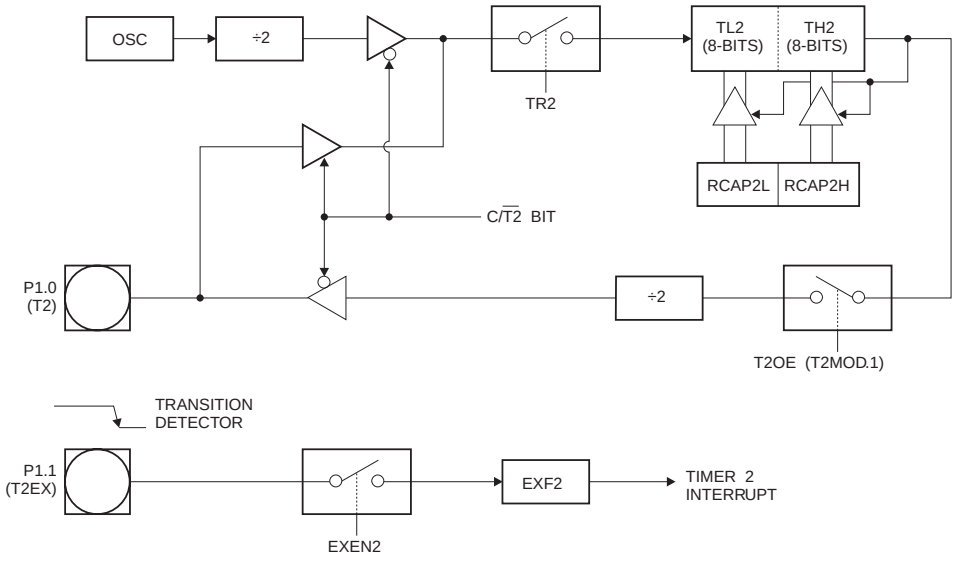
$$\frac{\text{Modes 1 and 3}}{\text{Baud Rate}} = \frac{\text{Oscillator Frequency}}{32 \times [65536 - \text{RCAP2H,RCAP2L})]}$$

where (RCAP2H, RCAP2L) is the content of RCAP2H and RCAP2L taken as a 16-bit unsigned integer.

Timer 2 as a baud rate generator is shown in Figure 8. This figure is valid only if RCLK or TCLK = 1 in T2CON. Note that a rollover in TH2 does not set TF2 and will not generate an interrupt. Note too, that if EXEN2 is set, a 1-to-0 transition in T2EX will set EXF2 but will not cause a reload from (RCAP2H, RCAP2L) to (TH2, TL2). Thus, when Timer 2 is in use as a baud rate generator, T2EX can be used as an extra external interrupt.

Note that when Timer 2 is running (TR2 = 1) as a timer in the baud rate generator mode, TH2 or TL2 should not be read from or written to. Under these conditions, the Timer is incremented every state time, and the results of a read or write may not be accurate. The RCAP2 registers may be read but should not be written to, because a write might overlap a reload and cause write and/or reload errors. The timer should be turned off (clear TR2) before accessing the Timer 2 or RCAP2 registers.

Figure 9. Timer 2 in Clock-Out Mode



Programmable Clock Out

A 50% duty cycle clock can be programmed to come out on P1.0, as shown in Figure 9. This pin, besides being a regular I/O pin, has two alternate functions. It can be programmed to input the external clock for Timer/Counter 2 or to output a 50% duty cycle clock ranging from 61 Hz to 4 MHz at a 16 MHz operating frequency.

To configure the Timer/Counter 2 as a clock generator, bit C/$\overline{T2}$ (T2CON.1) must be cleared and bit T2OE (T2MOD.1) must be set. Bit TR2 (T2CON.2) starts and stops the timer.

The clock-out frequency depends on the oscillator frequency and the reload value of Timer 2 capture registers (RCAP2H, RCAP2L), as shown in the following equation.

$$\text{Clock-Out Frequency} = \frac{\text{Oscillator Frequency}}{4 \times [65536-(\text{RCAP2H,RCAP2L})]}$$

In the clock-out mode, Timer 2 roll-overs will not generate an interrupt. This behavior is similar to when Timer 2 is used as a baud-rate generator. It is possible to use Timer 2 as a baud-rate generator and a clock generator simultaneously. Note, however, that the baud-rate and clock-out frequencies cannot be determined independently from one another since they both use RCAP2H and RCAP2L.

Interrupts

The AT89S52 has a total of six interrupt vectors: two external interrupts ($\overline{INT0}$ and $\overline{INT1}$), three timer interrupts (Timers 0, 1, and 2), and the serial port interrupt. These interrupts are all shown in Figure 10.

Each of these interrupt sources can be individually enabled or disabled by setting or clearing a bit in Special Function Register IE. IE also contains a global disable bit, EA, which disables all interrupts at once.

Note that Table 5 shows that bit position IE.6 is unimplemented. In the AT89S52, bit position IE.5 is also unimplemented. User software should not write 1s to these bit positions, since they may be used in future AT89 products.

Timer 2 interrupt is generated by the logical OR of bits TF2 and EXF2 in register T2CON. Neither of these flags is cleared by hardware when the service routine is vectored to. In fact, the service routine may have to determine whether it was TF2 or EXF2 that generated the interrupt, and that bit will have to be cleared in software.

The Timer 0 and Timer 1 flags, TF0 and TF1, are set at S5P2 of the cycle in which the timers overflow. The values are then polled by the circuitry in the next cycle. However, the Timer 2 flag, TF2, is set at S2P2 and is polled in the same cycle in which the timer overflows.

Table 5. Interrupt Enable (IE) Register

(MSB) (LSB)

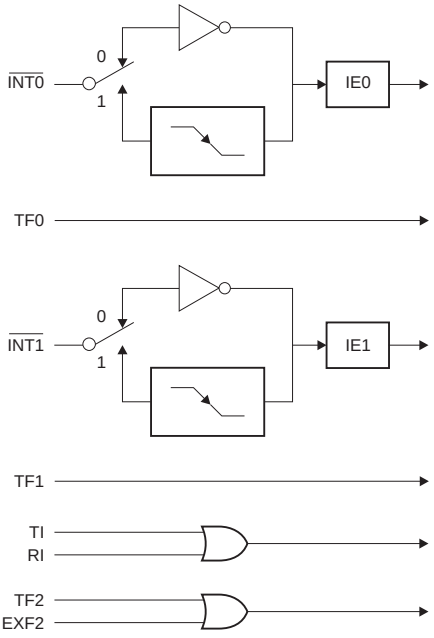
EA	–	ET2	ES	ET1	EX1	ET0	EX0

Enable Bit = 1 enables the interrupt.

Enable Bit = 0 disables the interrupt.

Symbol	Position	Function
EA	IE.7	Disables all interrupts. If EA = 0, no interrupt is acknowledged. If EA = 1, each interrupt source is individually enabled or disabled by setting or clearing its enable bit.
–	IE.6	Reserved.
ET2	IE.5	Timer 2 interrupt enable bit.
ES	IE.4	Serial Port interrupt enable bit.
ET1	IE.3	Timer 1 interrupt enable bit.
EX1	IE.2	External interrupt 1 enable bit.
ET0	IE.1	Timer 0 interrupt enable bit.
EX0	IE.0	External interrupt 0 enable bit.
User software should never write 1s to unimplemented bits, because they may be used in future AT89 products.		

Figure 10. Interrupt Sources



Oscillator Characteristics

XTAL1 and XTAL2 are the input and output, respectively, of an inverting amplifier that can be configured for use as an on-chip oscillator, as shown in Figure 11. Either a quartz crystal or ceramic resonator may be used. To drive the device from an external clock source, XTAL2 should be left unconnected while XTAL1 is driven, as shown in Figure 12. There are no requirements on the duty cycle of the external clock signal, since the input to the internal clocking circuitry is through a divide-by-two flip-flop, but minimum and maximum voltage high and low time specifications must be observed.

Idle Mode

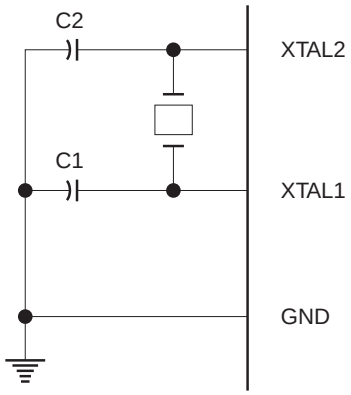
In idle mode, the CPU puts itself to sleep while all the on-chip peripherals remain active. The mode is invoked by software. The content of the on-chip RAM and all the special functions registers remain unchanged during this mode. The idle mode can be terminated by any enabled interrupt or by a hardware reset.

Note that when idle mode is terminated by a hardware reset, the device normally resumes program execution from where it left off, up to two machine cycles before the internal reset algorithm takes control. On-chip hardware inhibits access to internal RAM in this event, but access to the port pins is not inhibited. To eliminate the possibility of an unexpected write to a port pin when idle mode is terminated by a reset, the instruction following the one that invokes idle mode should not write to a port pin or to external memory.

Power-down Mode

In the Power-down mode, the oscillator is stopped, and the instruction that invokes Power-down is the last instruction executed. The on-chip RAM and Special Function Registers retain their values until the Power-down mode is terminated. Exit from Power-down mode can be initiated either by a hardware reset or by an enabled external interrupt. Reset redefines the SFRs but does not change the on-chip RAM. The reset should not be activated before V_{CC} is restored to its normal operating level and must be held active long enough to allow the oscillator to restart and stabilize.

Figure 11. Oscillator Connections



Note: C1, C2 = 30 pF ±10 pF for Crystals
= 40 pF ±10 pF for Ceramic Resonators

Figure 12. External Clock Drive Configuration

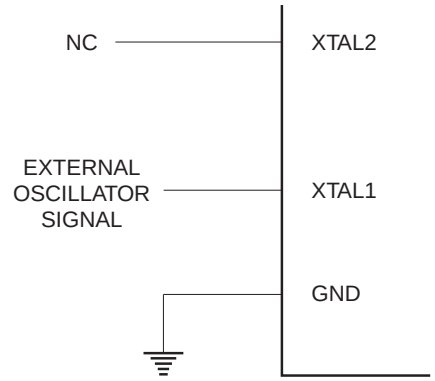


Table 6. Status of External Pins During Idle and Power-down Modes

Mode	Program Memory	ALE	$\overline{PSEN}$	PORT0	PORT1	PORT2	PORT3
Idle	Internal	1	1	Data	Data	Data	Data
Idle	External	1	1	Float	Data	Address	Data
Power-down	Internal	0	0	Data	Data	Data	Data
Power-down	External	0	0	Float	Data	Data	Data

Program Memory Lock Bits

The AT89S52 has three lock bits that can be left unprogrammed (U) or can be programmed (P) to obtain the additional features listed in the following table.

Table 7. Lock Bit Protection Modes

Program Lock Bits				
	LB1	LB2	LB3	Protection Type
1	U	U	U	No program lock features
2	P	U	U	MOVC instructions executed from external program memory are disabled from fetching code bytes from internal memory, $\overline{EA}$ is sampled and latched on reset, and further programming of the Flash memory is disabled
3	P	P	U	Same as mode 2, but verify is also disabled
4	P	P	P	Same as mode 3, but external execution is also disabled

When lock bit 1 is programmed, the logic level at the $\overline{EA}$ pin is sampled and latched during reset. If the device is powered up without a reset, the latch initializes to a random value and holds that value until reset is activated. The latched value of $\overline{EA}$ must agree with the current logic level at that pin in order for the device to function properly.

Programming the Flash – Parallel Mode

The AT89S52 is shipped with the on-chip Flash memory array ready to be programmed. The programming interface needs a high-voltage (12-volt) program enable signal and is compatible with conventional third-party Flash or EPROM programmers.

The AT89S52 code memory array is programmed byte-by-byte.

Programming Algorithm: Before programming the AT89S52, the address, data, and control signals should be set up according to the Flash programming mode table and Figures 13 and 14. To program the AT89S52, take the following steps:

1. Input the desired memory location on the address lines.
2. Input the appropriate data byte on the data lines.
3. Activate the correct combination of control signals.
4. Raise $\overline{EA}/V_{PP}$ to 12V.
5. Pulse ALE/$\overline{PROG}$ once to program a byte in the Flash array or the lock bits. The byte-write cycle is self-timed and typically takes no more than 50 µs.

   Repeat steps 1 through 5, changing the address and data for the entire array or until the end of the object file is reached.

$\overline{Data}$ Polling: The AT89S52 features $\overline{Data}$ Polling to indicate the end of a byte write cycle. During a write cycle, an attempted read of the last byte written will result in the complement of the written data on P0.7. Once the write cycle has been completed, true data is valid on all outputs, and the next cycle may begin. $\overline{Data}$ Polling may begin any time after a write cycle has been initiated.

Ready/$\overline{Busy}$: The progress of byte programming can also be monitored by the RDY/$\overline{BSY}$ output signal. P3.0 is pulled low after ALE goes high during programming to indicate $\overline{BUSY}$. P3.0 is pulled high again when programming is done to indicate READY.

Program Verify: If lock bits LB1 and LB2 have not been programmed, the programmed code data can be read back via the address and data lines for verification. The status of the individual lock bits can be verified directly by reading them back.

Reading the Signature Bytes: The signature bytes are read by the same procedure as a normal verification of locations 000H, 100H, and 200H, except that P3.6 and P3.7 must be pulled to a logic low. The values returned are as follows.

(000H) = 1EH indicates manufactured by Atmel
(100H) = 52H indicates 89S52
(200H) = 06H

Chip Erase: In the parallel programming mode, a chip erase operation is initiated by using the proper combination of control signals and by pulsing ALE/$\overline{PROG}$ low for a duration of 200 ns - 500 ns.

In the serial programming mode, a chip erase operation is initiated by issuing the Chip Erase instruction. In this mode, chip erase is self-timed and takes about 500 ms.

During chip erase, a serial read from any address location will return 00H at the data output.

Programming the Flash – Serial Mode

The Code memory array can be programmed using the serial ISP interface while RST is pulled to V_{CC}. The serial interface consists of pins SCK, MOSI (input) and MISO (output). After RST is set high, the Programming Enable instruction needs to be executed first before other operations can be executed. Before a reprogramming sequence can occur, a Chip Erase operation is required.

The Chip Erase operation turns the content of every memory location in the Code array into FFH.

Either an external system clock can be supplied at pin XTAL1 or a crystal needs to be connected across pins XTAL1 and XTAL2. The maximum serial clock (SCK)

frequency should be less than 1/16 of the crystal frequency. With a 33 MHz oscillator clock, the maximum SCK frequency is 2 MHz.

Serial Programming Algorithm

To program and verify the AT89S52 in the serial programming mode, the following sequence is recommended:

1. Power-up sequence:

   Apply power between VCC and GND pins.

   Set RST pin to “H”.

   If a crystal is not connected across pins XTAL1 and XTAL2, apply a 3 MHz to 33 MHz clock to XTAL1 pin and wait for at least 10 milliseconds.
2. Enable serial programming by sending the Programming Enable serial instruction to pin MOSI/P1.5. The frequency of the shift clock supplied at pin SCK/P1.7 needs to be less than the CPU clock at XTAL1 divided by 16.
3. The Code array is programmed one byte at a time by supplying the address and data together with the appropriate Write instruction. The write cycle is self-timed and typically takes less than 1 ms at 5V.
4. Any memory location can be verified by using the Read instruction which returns the content at the selected address at serial output MISO/P1.6.
5. At the end of a programming session, RST can be set low to commence normal device operation.

Power-off sequence (if needed):

Set XTAL1 to “L” (if a crystal is not used).

Set RST to “L”.

Turn V_{CC} power off.

$\overline{\text{Data}}$ Polling: The $\overline{\text{Data}}$ Polling feature is also available in the serial mode. In this mode, during a write cycle an attempted read of the last byte written will result in the complement of the MSB of the serial output byte on MISO.

Serial Programming Instruction Set

The Instruction Set for Serial Programming follows a 4-byte protocol and is shown in Table 10.

Programming Interface – Parallel Mode

Every code byte in the Flash array can be programmed by using the appropriate combination of control signals. The write operation cycle is self-timed and once initiated, will automatically time itself to completion.

All major programming vendors offer worldwide support for the Atmel microcontroller series. Please contact your local programming vendor for the appropriate software revision.

Table 8. Flash Programming Modes

Mode	V_{CC}	RST	$\overline{PSEN}$	ALE/ $\overline{PROG}$	$\overline{EA}$/ V_{PP}	P2.6	P2.7	P3.3	P3.6	P3.7	P0.7-0 Data	P2.4-0 Address	P1.7-0 Address
Write Code Data	5V	H	L	(2)	12V	L	H	H	H	H	D_{IN}	A12-8	A7-0
Read Code Data	5V	H	L	H	H	L	L	L	H	H	D_{OUT}	A12-8	A7-0
Write Lock Bit 1	5V	H	L	(3)	12V	H	H	H	H	H	X	X	X
Write Lock Bit 2	5V	H	L	(3)	12V	H	H	H	L	L	X	X	X
Write Lock Bit 3	5V	H	L	(3)	12V	H	L	H	H	L	X	X	X
Read Lock Bits 1, 2, 3	5V	H	L	H	H	H	H	L	H	L	P0.2, P0.3, P0.4	X	X
Chip Erase	5V	H	L	(1)	12V	H	L	H	L	L	X	X	X
Read Atmel ID	5V	H	L	H	H	L	L	L	L	L	1EH	X 0000	00H
Read Device ID	5V	H	L	H	H	L	L	L	L	L	52H	X 0001	00H
Read Device ID	5V	H	L	H	H	L	L	L	L	L	06H	X 0010	00H

Notes:
1. Each $\overline{PROG}$ pulse is 200 ns - 500 ns for Chip Erase.
2. Each $\overline{PROG}$ pulse is 200 ns - 500 ns for Write Code Data.
3. Each $\overline{PROG}$ pulse is 200 ns - 500 ns for Write Lock Bits.
4. RDY/$\overline{BSY}$ signal is output on P3.0 during programming.
5. X = don't care.

Figure 13. Programming the Flash Memory (Parallel Mode)

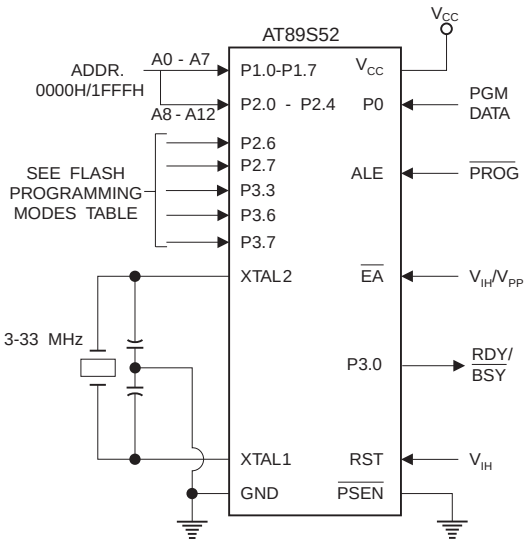
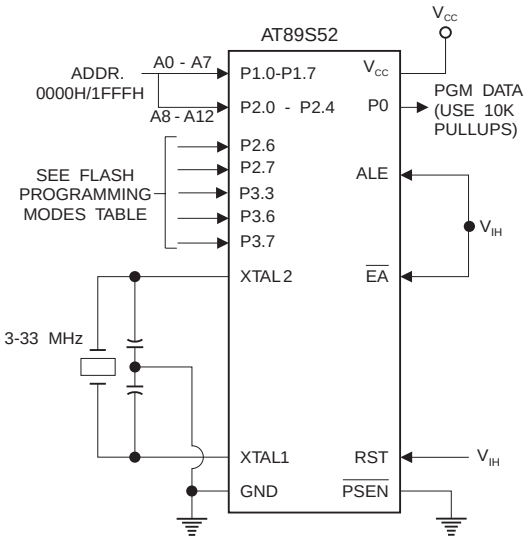


Figure 14. Verifying the Flash Memory (Parallel Mode)



Flash Programming and Verification Characteristics (Parallel Mode)

T_A = 20°C to 30°C, V_{CC} = 4.5 to 5.5V

Symbol	Parameter	Min	Max	Units
V_{PP}	Programming Supply Voltage	11.5	12.5	V
I_{PP}	Programming Supply Current		10	mA
I_{CC}	V_{CC} Supply Current		30	mA
$1/t_{CLCL}$	Oscillator Frequency	3	33	MHz
t_{AVGL}	Address Setup to $\overline{PROG}$ Low	$48t_{CLCL}$		
t_{GHAX}	Address Hold After $\overline{PROG}$	$48t_{CLCL}$		
t_{DVGL}	Data Setup to $\overline{PROG}$ Low	$48t_{CLCL}$		
t_{GHDX}	Data Hold After $\overline{PROG}$	$48t_{CLCL}$		
t_{EHSH}	P2.7 ($\overline{ENABLE}$) High to V_{PP}	$48t_{CLCL}$		
t_{SHGL}	V_{PP} Setup to $\overline{PROG}$ Low	10		µs
t_{GHSL}	V_{PP} Hold After $\overline{PROG}$	10		µs
t_{GLGH}	$\overline{PROG}$ Width	0.2	1	µs
t_{AVQV}	Address to Data Valid		$48t_{CLCL}$	
t_{ELQV}	$\overline{ENABLE}$ Low to Data Valid		$48t_{CLCL}$	
t_{EHQZ}	Data Float After $\overline{ENABLE}$	0	$48t_{CLCL}$	
t_{GHBL}	$\overline{PROG}$ High to $\overline{BUSY}$ Low		1.0	µs
t_{WC}	Byte Write Cycle Time		50	µs

Figure 15. Flash Programming and Verification Waveforms – Parallel Mode

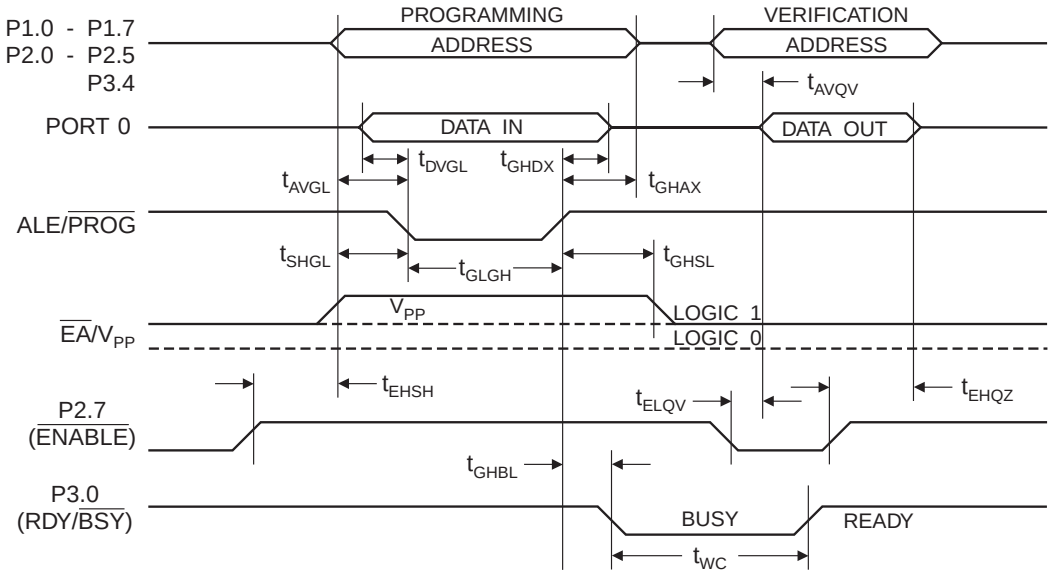
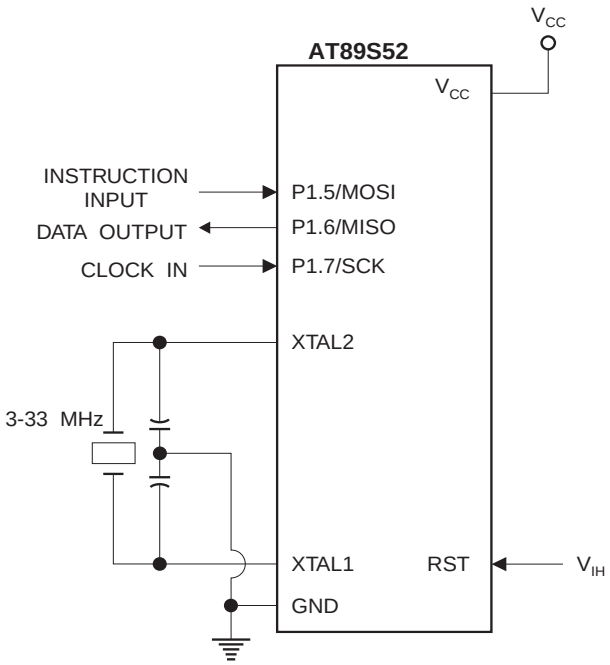


Figure 16. Flash Memory Serial Downloading



Flash Programming and Verification Waveforms – Serial Mode

Figure 17. Serial Programming Waveforms

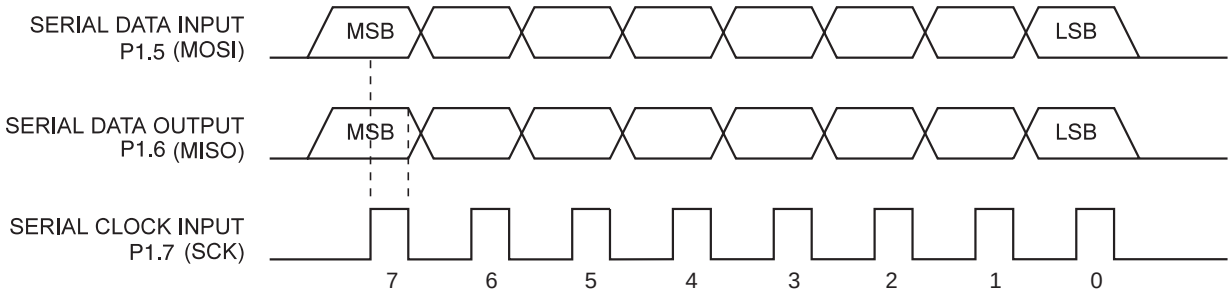


Table 9. Serial Programming Instruction Set

Instruction	Instruction Format Byte 1	Byte 2	Byte 3	Byte 4	Operation
Programming Enable	1010 1100	0101 0011	xxxx xxxx	xxxx xxxx 0110 1001 (Output)	Enable Serial Programming while RST is high
Chip Erase	1010 1100	100x xxxx	xxxx xxxx	xxxx xxxx	Chip Erase Flash memory array
Read Program Memory (Byte Mode)	0010 0000	xxx A12 A11 A10 A9 A8	A7 A6 A5 A4 A3 A2 A1 A0	D7 D6 D5 D4 D3 D2 D1 D0	Read data from Program memory in the byte mode
Write Program Memory (Byte Mode)	0100 0000	xxx A12 A11 A10 A9 A8	A7 A6 A5 A4 A3 A2 A1 A0	D7 D6 D5 D4 D3 D2 D1 D0	Write data to Program memory in the byte mode
Write Lock Bits[2]	1010 1100	1110 00 B1 B2	xxxx xxxx	xxxx xxxx	Write Lock bits. See Note (2).
Read Lock Bits	0010 0100	xxxx xxxx	xxxx xxxx	xx LB3 LB2 LB1 xx	Read back current status of the lock bits (a programmed lock bit reads back as a '1')
Read Signature Bytes[1]	0010 1000	xxx A5 A4 A3 A2 A1 A0	xxx xxxx	Signature Byte	Read Signature Byte
Read Program Memory (Page Mode)	0011 0000	xxx A12 A11 A10 A9 A8	Byte 0	Byte 1... Byte 255	Read data from Program memory in the Page Mode (256 bytes)
Write Program Memory (Page Mode)	0101 0000	xxx A12 A11 A10 A9 A8	Byte 0	Byte 1... Byte 255	Write data to Program memory in the Page Mode (256 bytes)

Notes: 1. The signature bytes are not readable in Lock Bit Modes 3 and 4.

2. B1 = 0, B2 = 0 ---> Mode 1, no lock protection
   B1 = 0, B2 = 1 ---> Mode 2, lock bit 1 activated
   B1 = 1, B2 = 0 ---> Mode 3, lock bit 2 activated
   B1 = 1, B1 = 1 ---> Mode 4, lock bit 3 activated

   } Each of the lock bits needs to be activated sequentially before Mode 4 can be executed.

After Reset signal is high, SCK should be low for at least 64 system clocks before it goes high to clock in the enable data bytes. No pulsing of Reset signal is necessary. SCK should be no faster than 1/16 of the system clock at XTAL1.

For Page Read/Write, the data always starts from byte 0 to 255. After the command byte and upper address byte are latched, each byte thereafter is treated as data until all 256 bytes are shifted in/out. Then the next instruction will be ready to be decoded.

Serial Programming Characteristics

Figure 18. Serial Programming Timing

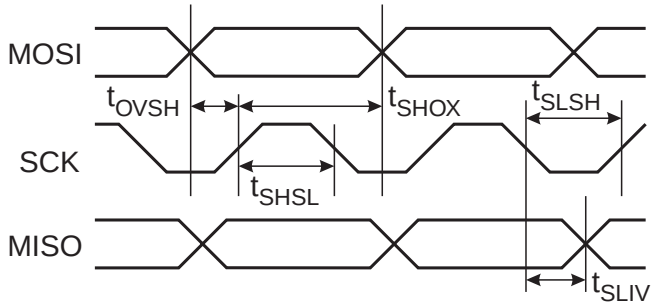


Table 10. Serial Programming Characteristics, T_A = -40° C to 85° C, V_{CC} = 4.0 - 5.5V (Unless otherwise noted)

Symbol	Parameter	Min	Typ	Max	Units
$1/t_{CLCL}$	Oscillator Frequency	0		33	MHz
t_{CLCL}	Oscillator Period	30			ns
t_{SHSL}	SCK Pulse Width High	2 t_{CLCL}			ns
t_{SLSH}	SCK Pulse Width Low	2 t_{CLCL}			ns
t_{OVSH}	MOSI Setup to SCK High	t_{CLCL}			ns
t_{SHOX}	MOSI Hold after SCK High	2 t_{CLCL}			ns
t_{SLIV}	SCK Low to MISO Valid	10	16	32	ns
t_{ERASE}	Chip Erase Instruction Cycle Time			500	ms
t_{SWC}	Serial Byte Write Cycle Time			64 t_{CLCL} + 400	µs

Absolute Maximum Ratings*

Operating Temperature	-55°C to +125°C
Storage Temperature	-65°C to +150°C
Voltage on Any Pin with Respect to Ground	-1.0V to +7.0V
Maximum Operating Voltage	6.6V
DC Output Current	15.0 mA

*NOTICE: Stresses beyond those listed under "Absolute Maximum Ratings" may cause permanent damage to the device. This is a stress rating only and functional operation of the device at these or any other conditions beyond those indicated in the operational sections of this specification is not implied. Exposure to absolute maximum rating conditions for extended periods may affect device reliability.

DC Characteristics

The values shown in this table are valid for T_A = -40°C to 85°C and V_{CC} = 4.0V to 5.5V, unless otherwise noted.

Symbol	Parameter	Condition	Min	Max	Units
V_{IL}	Input Low Voltage	(Except $\overline{EA}$)	-0.5	0.2 V_{CC}-0.1	V
V_{IL1}	Input Low Voltage ($\overline{EA}$)		-0.5	0.2 V_{CC}-0.3	V
V_{IH}	Input High Voltage	(Except XTAL1, RST)	0.2 V_{CC}+0.9	V_{CC}+0.5	V
V_{IH1}	Input High Voltage	(XTAL1, RST)	0.7 V_{CC}	V_{CC}+0.5	V
V_{OL}	Output Low Voltage(1) (Ports 1,2,3)	I_{OL} = 1.6 mA		0.45	V
V_{OL1}	Output Low Voltage(1) (Port 0, ALE, $\overline{PSEN}$)	I_{OL} = 3.2 mA		0.45	V
V_{OH}	Output High Voltage (Ports 1,2,3, ALE, $\overline{PSEN}$)	I_{OH} = -60 µA, V_{CC} = 5V ±10%	2.4		V
		I_{OH} = -25 µA	0.75 V_{CC}		V
		I_{OH} = -10 µA	0.9 V_{CC}		V
V_{OH1}	Output High Voltage (Port 0 in External Bus Mode)	I_{OH} = -800 µA, V_{CC} = 5V ±10%	2.4		V
		I_{OH} = -300 µA	0.75 V_{CC}		V
		I_{OH} = -80 µA	0.9 V_{CC}		V
I_{IL}	Logical 0 Input Current (Ports 1,2,3)	V_{IN} = 0.45V		-50	µA
I_{TL}	Logical 1 to 0 Transition Current (Ports 1,2,3)	V_{IN} = 2V, V_{CC} = 5V ±10%		-650	µA
I_{LI}	Input Leakage Current (Port 0, $\overline{EA}$)	0.45 < V_{IN} < V_{CC}		±10	µA
RRST	Reset Pulldown Resistor		10	30	KΩ
C_{IO}	Pin Capacitance	Test Freq. = 1 MHz, T_A = 25°C		10	pF
I_{CC}	Power Supply Current	Active Mode, 12 MHz		25	mA
		Idle Mode, 12 MHz		6.5	mA
	Power-down Mode(1)	V_{CC} = 5.5V		50	µA

Notes: 1. Under steady state (non-transient) conditions, I_{OL} must be externally limited as follows:
Maximum I_{OL} per port pin: 10 mA
Maximum I_{OL} per 8-bit port:
Port 0: 26 mA Ports 1, 2, 3: 15 mA
Maximum total I_{OL} for all output pins: 71 mA
If I_{OL} exceeds the test condition, V_{OL} may exceed the related specification. Pins are not guaranteed to sink current greater than the listed test conditions.

2. Minimum V_{CC} for Power-down is 2V.

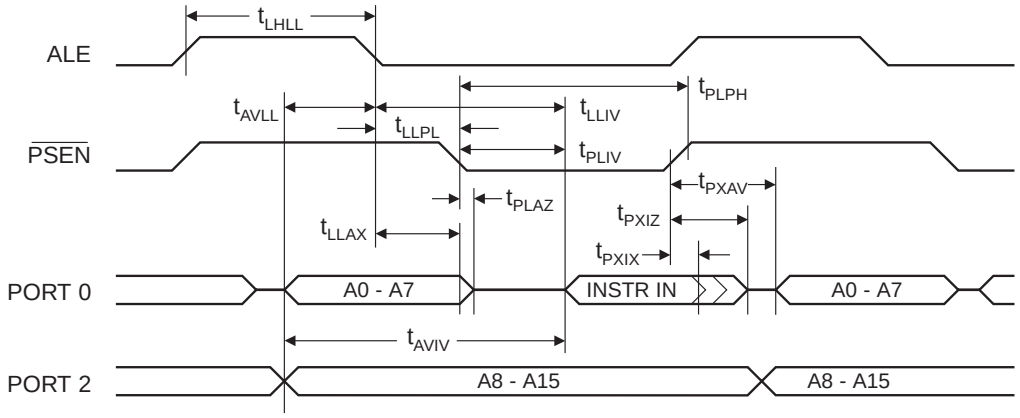
AC Characteristics

Under operating conditions, load capacitance for Port 0, ALE/$\overline{PROG}$, and $\overline{PSEN}$ = 100 pF; load capacitance for all other outputs = 80 pF.

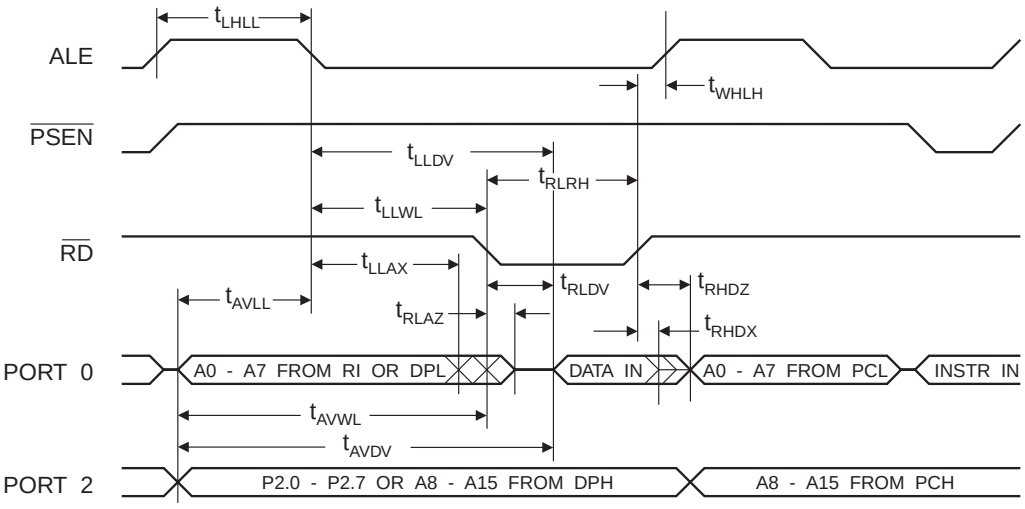
External Program and Data Memory Characteristics

Symbol	Parameter	12 MHz Oscillator		Variable Oscillator		Units
		Min	Max	Min	Max	
$1/t_{CLCL}$	Oscillator Frequency			0	33	MHz
t_{LHLL}	ALE Pulse Width	127		$2t_{CLCL}$-40		ns
t_{AVLL}	Address Valid to ALE Low	43		t_{CLCL}-25		ns
t_{LLAX}	Address Hold After ALE Low	48		t_{CLCL}-25		ns
t_{LLIV}	ALE Low to Valid Instruction In		233		$4t_{CLCL}$-65	ns
t_{LLPL}	ALE Low to $\overline{PSEN}$ Low	43		t_{CLCL}-25		ns
t_{PLPH}	$\overline{PSEN}$ Pulse Width	205		$3t_{CLCL}$-45		ns
t_{PLIV}	$\overline{PSEN}$ Low to Valid Instruction In		145		$3t_{CLCL}$-60	ns
t_{PXIX}	Input Instruction Hold After $\overline{PSEN}$	0		0		ns
t_{PXIZ}	Input Instruction Float After $\overline{PSEN}$		59		t_{CLCL}-25	ns
t_{PXAV}	$\overline{PSEN}$ to Address Valid	75		t_{CLCL}-8		ns
t_{AVIV}	Address to Valid Instruction In		312		$5t_{CLCL}$-80	ns
t_{PLAZ}	$\overline{PSEN}$ Low to Address Float		10		10	ns
t_{RLRH}	$\overline{RD}$ Pulse Width	400		$6t_{CLCL}$-100		ns
t_{WLWH}	$\overline{WR}$ Pulse Width	400		$6t_{CLCL}$-100		ns
t_{RLDV}	$\overline{RD}$ Low to Valid Data In		252		$5t_{CLCL}$-90	ns
t_{RHDX}	Data Hold After $\overline{RD}$	0		0		ns
t_{RHDZ}	Data Float After $\overline{RD}$		97		$2t_{CLCL}$-28	ns
t_{LLDV}	ALE Low to Valid Data In		517		$8t_{CLCL}$-150	ns
t_{AVDV}	Address to Valid Data In		585		$9t_{CLCL}$-165	ns
t_{LLWL}	ALE Low to $\overline{RD}$ or $\overline{WR}$ Low	200	300	$3t_{CLCL}$-50	$3t_{CLCL}$+50	ns
t_{AVWL}	Address to $\overline{RD}$ or $\overline{WR}$ Low	203		$4t_{CLCL}$-75		ns
t_{QVWX}	Data Valid to $\overline{WR}$ Transition	23		t_{CLCL}-30		ns
t_{QVWH}	Data Valid to $\overline{WR}$ High	433		$7t_{CLCL}$-130		ns
t_{WHQX}	Data Hold After $\overline{WR}$	33		t_{CLCL}-25		ns
t_{RLAZ}	$\overline{RD}$ Low to Address Float		0		0	ns
t_{WHLH}	$\overline{RD}$ or $\overline{WR}$ High to ALE High	43	123	t_{CLCL}-25	t_{CLCL}+25	ns

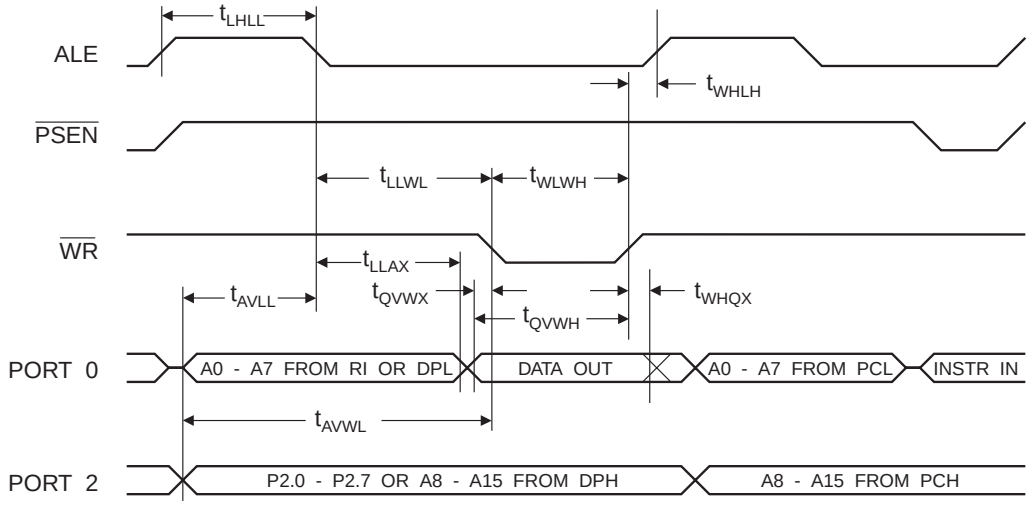
External Program Memory Read Cycle



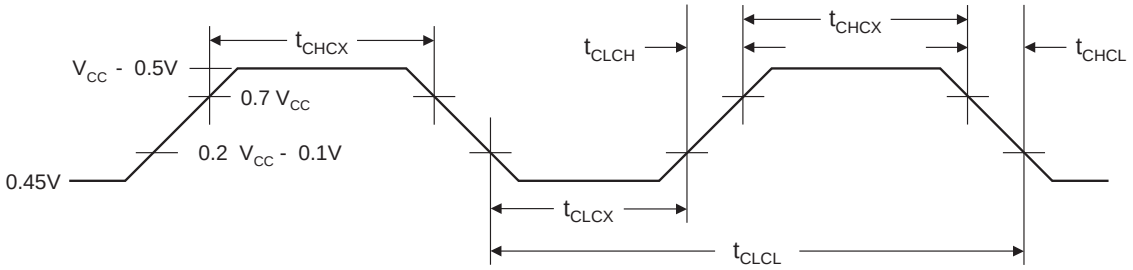
External Data Memory Read Cycle



External Data Memory Write Cycle



External Clock Drive Waveforms



External Clock Drive

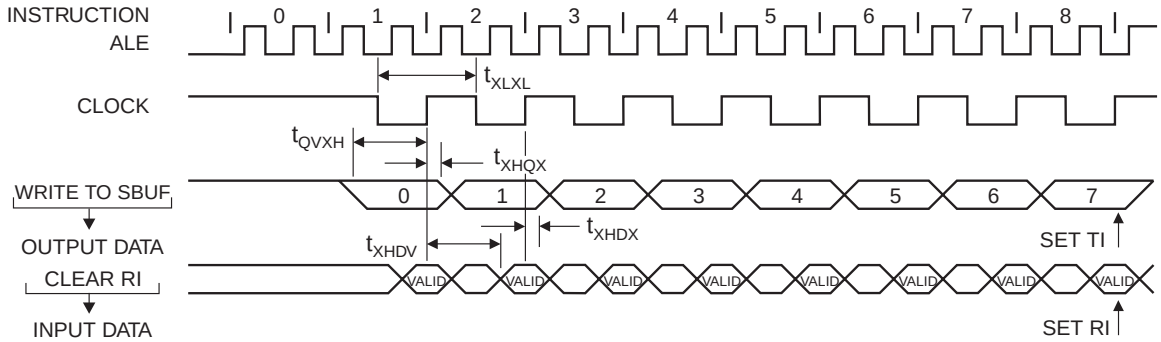
Symbol	Parameter	Min	Max	Units
$1/t_{CLCL}$	Oscillator Frequency	0	33	MHz
t_{CLCL}	Clock Period	30		ns
t_{CHCX}	High Time	12		ns
t_{CLCX}	Low Time	12		ns
t_{CLCH}	Rise Time		5	ns
t_{CHCL}	Fall Time		5	ns

Serial Port Timing: Shift Register Mode Test Conditions

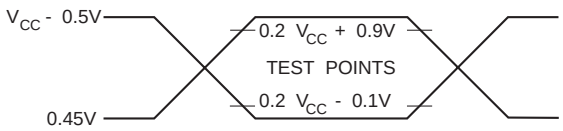
The values in this table are valid for V_{CC} = 4.0V to 5.5V and Load Capacitance = 80 pF.

Symbol	Parameter	12 MHz Osc		Variable Oscillator		Units
		Min	Max	Min	Max	
t_{XLXL}	Serial Port Clock Cycle Time	1.0		$12t_{CLCL}$		µs
t_{QVXH}	Output Data Setup to Clock Rising Edge	700		$10t_{CLCL}$-133		ns
t_{XHQX}	Output Data Hold After Clock Rising Edge	50		$2t_{CLCL}$-80		ns
t_{XHDX}	Input Data Hold After Clock Rising Edge	0		0		ns
t_{XHDV}	Clock Rising Edge to Input Data Valid		700		$10t_{CLCL}$-133	ns

Shift Register Mode Timing Waveforms

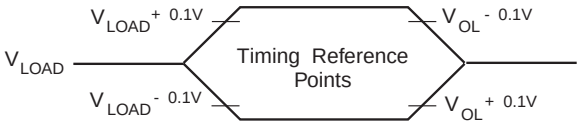


AC Testing Input/Output Waveforms(1)



Note: 1. AC Inputs during testing are driven at V_{CC} - 0.5V for a logic 1 and 0.45V for a logic 0. Timing measurements are made at V_{IH} min. for a logic 1 and V_{IL} max. for a logic 0.

Float Waveforms(1)



Note: 1. For timing purposes, a port pin is no longer floating when a 100 mV change from load voltage occurs. A port pin begins to float when a 100 mV change from the loaded V_{OH}/V_{OL} level occurs.

Ordering Information

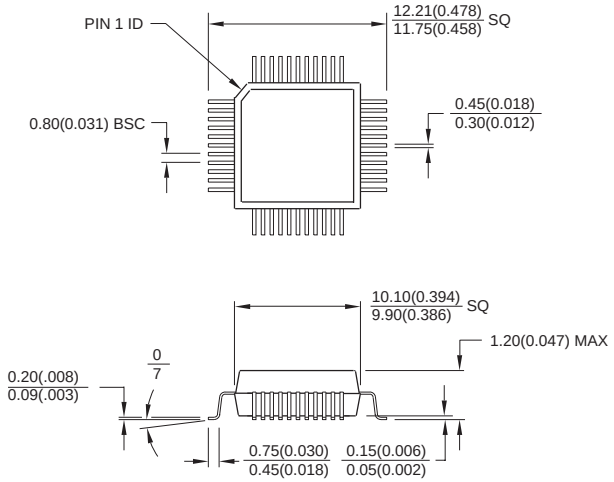
Speed (MHz)	Power Supply	Ordering Code	Package	Operation Range
24	4.0V to 5.5V	AT89S52-24AC AT89S52-24JC AT89S52-24PC	44A 44J 40P6	Commercial (0° C to 70° C)
		AT89S52-24AI AT89S52-24JI AT89S52-24PI	44A 44J 40P6	Industrial (-40° C to 85° C)
33	4.5V to 5.5V	AT89S52-33AC AT89S52-33JC AT89S52-33PC	44A 44J 40P6	Commercial (0° C to 70° C)

= Preliminary Availability

Package Type	
44A	44-lead, Thin Plastic Gull Wing Quad Flatpack (TQFP)
44J	44-lead, Plastic J-leaded Chip Carrier (PLCC)
40P6	40-pin, 0.600" Wide, Plastic Dual Inline Package (PDIP)

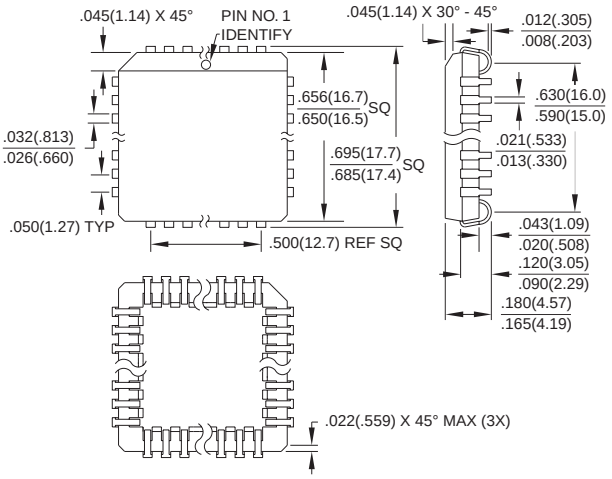
Packaging Information

44A, 44-lead, Thin (1.0 mm) Plastic Gull Wing Quad Flat Package (TQFP)
Dimensions in Millimeters and (Inches)*

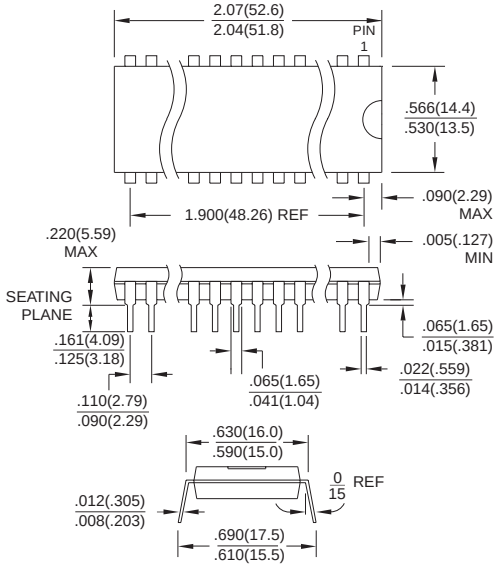


*Controlling dimension: millimeters

44J, 44-lead, Plastic J-leaded Chip Carrier (PLCC)
Dimensions in Inches and (Millimeters)



40P6, 40-pin, 0.600" Wide, Plastic Dual Inline Package (PDIP)
Dimensions in Inches and (Millimeters)
JEDEC STANDARD MS-011 AC



Atmel Headquarters

Corporate Headquarters
2325 Orchard Parkway
San Jose, CA 95131
TEL (408) 441-0311
FAX (408) 487-2600

Europe
Atmel SarL
Route des Arsenaux 41
Casa Postale 80
CH-1705 Fribourg
Switzerland
TEL (41) 26-426-5555
FAX (41) 26-426-5500

Asia
Atmel Asia, Ltd.
Room 1219
Chinachem Golden Plaza
77 Mody Road Tsimhatsui
East Kowloon
Hong Kong
TEL (852) 2721-9778
FAX (852) 2722-1369

Japan
Atmel Japan K.K.
9F, Tonetsu Shinkawa Bldg.
1-24-8 Shinkawa
Chuo-ku, Tokyo 104-0033
Japan
TEL (81) 3-3523-3551
FAX (81) 3-3523-7581

Atmel Product Operations

Atmel Colorado Springs
1150 E. Cheyenne Mtn. Blvd.
Colorado Springs, CO 80906
TEL (719) 576-3300
FAX (719) 540-1759

Atmel Grenoble
Avenue de Rochepleine
BP 123
38521 Saint-Egreve Cedex, France
TEL (33) 4-7658-3000
FAX (33) 4-7658-3480

Atmel Heilbronn
Theresienstrasse 2
POB 3535
D-74025 Heilbronn, Germany
TEL (49) 71 31 67 25 94
FAX (49) 71 31 67 24 23

Atmel Nantes
La Chantrerie
BP 70602
44306 Nantes Cedex 3, France
TEL (33) 0 2 40 18 18 18
FAX (33) 0 2 40 18 19 60

Atmel Rousset
Zone Industrielle
13106 Rousset Cedex, France
TEL (33) 4-4253-6000
FAX (33) 4-4253-6001

Atmel Smart Card ICs
Scottish Enterprise Technology Park
East Kilbride, Scotland G75 0QR
TEL (44) 1355-357-000
FAX (44) 1355-242-743

Fax-on-Demand
North America:
1-(800) 292-8635
International:
1-(408) 441-0732

e-mail
literature@atmel.com

Web Site
http://www.atmel.com

BBS
1-(408) 436-4309

Rev.1919A-07/01/xM

Microcontroller Instruction Set

For interrupt response time information, refer to the hardware description chapter.

Instructions that Affect Flag Settings(1)

Instruction	Flag			Instruction	Flag		
	C	OV	AC		C	OV	AC
ADD	X	X	X	CLR C	O		
ADDC	X	X	X	CPL C	X		
SUBB	X	X	X	ANL C,bit	X		
MUL	O	X		ANL C,/bit	X		
DIV	O	X		ORL C,bit	X		
DA	X			ORL C,/bit	X		
RRC	X			MOV C,bit	X		
RLC	X			CJNE	X		
SETB C	1						

Note: 1. Operations on SFR byte address 208 or bit addresses 209-215 (that is, the PSW or bits in the PSW) also affect flag settings.

The Instruction Set and Addressing Modes

R_n	Register R7-R0 of the currently selected Register Bank.
direct	8-bit internal data location's address. This could be an Internal Data RAM location (0-127) or a SFR [i.e., I/O port, control register, status register, etc. (128-255)].
@R_i	8-bit internal data RAM location (0-255) addressed indirectly through register R1or R0.
#data	8-bit constant included in instruction.
#data 16	16-bit constant included in instruction.
addr 16	16-bit destination address. Used by LCALL and LJMP. A branch can be anywhere within the 64K byte Program Memory address space.
addr 11	11-bit destination address. Used by ACALL and AJMP. The branch will be within the same 2K byte page of program memory as the first byte of the following instruction.
rel	Signed (two's complement) 8-bit offset byte. Used by SJMP and all conditional jumps. Range is -128 to +127 bytes relative to first byte of the following instruction.
bit	Direct Addressed bit in Internal Data RAM or Special Function Register.

0509B-B–12/97

Instruction Set Summary

	0	1	2	3	4	5	6	7
0	NOP	JBC bit,rel [3B, 2C]	JB bit, rel [3B, 2C]	JNB bit, rel [3B, 2C]	JC rel [2B, 2C]	JNC rel [2B, 2C]	JZ rel [2B, 2C]	JNZ rel [2B, 2C]
1	AJMP (P0) [2B, 2C]	ACALL (P0) [2B, 2C]	AJMP (P1) [2B, 2C]	ACALL (P1) [2B, 2C]	AJMP (P2) [2B, 2C]	ACALL (P2) [2B, 2C]	AJMP (P3) [2B, 2C]	ACALL (P3) [2B, 2C]
2	LJMP addr16 [3B, 2C]	LCALL addr16 [3B, 2C]	RET [2C]	RETI [2C]	ORL dir, A [2B]	ANL dir, A [2B]	XRL dir, a [2B]	ORL C, bit [2B, 2C]
3	RR A	RRC A	RL A	RLC A	ORL dir, #data [3B, 2C]	ANL dir, #data [3B, 2C]	XRL dir, #data [3B, 2C]	JMP @A + DPTR [2C]
4	INC A	DEC A	ADD A, #data [2B]	ADDC A, #data [2B]	ORL A, #data [2B]	ANL A, #data [2B]	XRL A, #data [2B]	MOV A, #data [2B]
5	INC dir [2B]	DEC dir [2B]	ADD A, dir [2B]	ADDC A, dir [2B]	ORL A, dir [2B]	ANL A, dir [2B]	XRL A, dir [2B]	MOV dir, #data [3B, 2C]
6	INC @R0	DEC @R0	ADD A, @R0	ADDC A, @R0	ORL A, @R0	ANL A, @R0	XRL A, @R0	MOV @R0, @data [2B]
7	INC @R1	DEC @R1	ADD A, @R1	ADDC A, @R1	ORL A, @R1	ANL A, @R1	XRL A, @R1	MOV @R1, #data [2B]
8	INC R0	DEC R0	ADD A, R0	ADDC A, R0	ORL A, R0	ANL A, R0	XRL A, R0	MOV R0, #data [2B]
9	INC R1	DEC R1	ADD A, R1	ADDC A, R1	ORL A, R1	ANL A, R1	XRL A, R1	MOV R1, #data [2B]
A	INC R2	DEC R2	ADD A, R2	ADDC A, R2	ORL A, R2	ANL A, R2	XRL A, R2	MOV R2, #data [2B]
B	INC R3	DEC R3	ADD A, R3	ADDC A, R3	ORL A, R3	ANL A, R3	XRL A, R3	MOV R3, #data [2B]
C	INC R4	DEC R4	ADD A, R4	ADDC A, R4	ORL A, R4	ANL A, R4	XRL A, R4	MOV R4, #data [2B]
D	INC R5	DEC R5	ADD A, R5	ADDC A, R5	ORL A, R5	ANL A, R5	XRL A, R5	MOV R5, #data [2B]
E	INC R6	DEC R6	ADD A, R6	ADDC A, R6	ORL A, R6	ANL A, R6	XRL A, R6	MOV R6, #data [2B]
F	INC R7	DEC R7	ADD A, R7	ADDC A, R7	ORL A, R7	ANL A, R7	XRL A, R7	MOV R7, #data [2B]

Note: Key: [2B] = 2 Byte, [3B] = 3 Byte, [2C] = 2 Cycle, [4C] = 4 Cycle, Blank = 1 byte/1 cycle

Instruction Set Summary (Continued)

	8	9	A	B	C	D	E	F
0	SJMP REL [2B, 2C]	MOV DPTR,# data 16 [3B, 2C]	ORL C, /bit [2B, 2C]	ANL C, /bit [2B, 2C]	PUSH dir [2B, 2C]	POP dir [2B, 2C]	MOVX A, @DPTR [2C]	MOVX @DPTR, A [2C]
1	AJMP (P4) [2B, 2C]	ACALL (P4) [2B, 2C]	AJMP (P5) [2B, 2C]	ACALL (P5) [2B, 2C]	AJMP (P6) [2B, 2C]	ACALL (P6) [2B, 2C]	AJMP (P7) [2B, 2C]	ACALL (P7) [2B, 2C]
2	ANL C, bit [2B, 2C]	MOV bit, C [2B, 2C]	MOV C, bit [2B]	CPL bit [2B]	CLR bit [2B]	SETB bit [2B]	MOVX A, @R0 [2C]	MOVX wR0, A [2C]
3	MOVC A, @A + PC [2C]	MOVC A, @A + DPTR [2C]	INC DPTR [2C]	CPL C	CLR C	SETB C	MOVX A, @RI [2C]	MOVX @RI, A [2C]
4	DIV AB [2B, 4C]	SUBB A, #data [2B]	MUL AB [4C]	CJNE A, #data, rel [3B, 2C]	SWAP A	DA A	CLR A	CPL A
5	MOV dir, dir [3B, 2C]	SUBB A, dir [2B]		CJNE A, dir, rel [3B, 2C]	XCH A, dir [2B]	DJNZ dir, rel [3B, 2C]	MOV A, dir [2B]	MOV dir, A [2B]
6	MOV dir, @R0 [2B, 2C]	SUBB A, @R0	MOV @R0, dir [2B, 2C]	CJNE @R0, #data, rel [3B, 2C]	XCH A, @R0	XCHD A, @R0	MOV A, @R0	MOV @R0, A
7	MOV dir, @R1 [2B, 2C]	SUBB A, @R1	MOV @R1, dir [2B, 2C]	CJNE @R1, #data, rel [3B, 2C]	XCH A, @R1	XCHD A, @R1	MOV A, @R1	MOV @R1, A
8	MOV dir, R0 [2B, 2C]	SUBB A, R0	MOV R0, dir [2B, 2C]	CJNE R0, #data, rel [3B, 2C]	XCH A, R0	DJNZ R0, rel [2B, 2C]	MOV A, R0	MOV R0, A
9	MOV dir, R1 [2B, 2C]	SUBB A, R1	MOV R1, dir [2B, 2C]	CJNE R1, #data, rel [3B, 2C]	XCH A, R1	DJNZ R1, rel [2B, 2C]	MOV A, R1	MOV R1, A
A	MOV dir, R2 [2B, 2C]	SUBB A, R2	MOV R2, dir [2B, 2C]	CJNE R2, #data, rel [3B, 2C]	XCH A, R2	DJNZ R2, rel [2B, 2C]	MOV A, R2	MOV R2, A
B	MOV dir, R3 [2B, 2C]	SUBB A, R3	MOV R3, dir [2B, 2C]	CJNE R3, #data, rel [3B, 2C]	XCH A, R3	DJNZ R3, rel [2B, 2C]	MOV A, R3	MOV R3, A
C	MOV dir, R4 [2B, 2C]	SUBB A, R4	MOV R4, dir [2B, 2C]	CJNE R4, #data, rel [3B, 2C]	XCH A, R4	DJNZ R4, rel [2B, 2C]	MOV A, R4	MOV R4, A
D	MOV dir, R5 [2B, 2C]	SUBB A, R5	MOV R5, dir [2B, 2C]	CJNE R5, #data, rel [3B, 2C]	XCH A, R5	DJNZ R5, rel [2B, 2C]	MOV A, R5	MOV R5, A
E	MOV dir, R6 [2B, 2C]	SUBB A, R6	MOV R6, dir [2B, 2C]	CJNE R6, #data, rel [3B, 2C]	XCH A, R6	DJNZ R6, rel [2B, 2C]	MOV A, R6	MOV R6. A
F	MOV dir, R7 [2B, 2C]	SUBB A, R7	MOV R7, dir [2B, 2C]	CJNE R7, #data, rel [3B, 2C]	XCH A, R7	DJNZ R7, rel [2B, 2C]	MOV A, R7	MOV R7, A

Note: Key: [2B] = 2 Byte, [3B] = 3 Byte, [2C] = 2 Cycle, [4C] = 4 Cycle, Blank = 1 byte/1 cycle

Table 1. AT89 Instruction Set Summary[1]

Mnemonic		Description	Byte	Oscillator Period
ARITHMETIC OPERATIONS				
ADD	A,R_n	Add register to Accumulator	1	12
ADD	A,direct	Add direct byte to Accumulator	2	12
ADD	A,@R_i	Add indirect RAM to Accumulator	1	12
ADD	A,#data	Add immediate data to Accumulator	2	12
ADDC	A,R_n	Add register to Accumulator with Carry	1	12
ADDC	A,direct	Add direct byte to Accumulator with Carry	2	12
ADDC	A,@R_i	Add indirect RAM to Accumulator with Carry	1	12
ADDC	A,#data	Add immediate data to Acc with Carry	2	12
SUBB	A,R_n	Subtract Register from Acc with borrow	1	12
SUBB	A,direct	Subtract direct byte from Acc with borrow	2	12
SUBB	A,@R_i	Subtract indirect RAM from ACC with borrow	1	12
SUBB	A,#data	Subtract immediate data from Acc with borrow	2	12
INC	A	Increment Accumulator	1	12
INC	R_n	Increment register	1	12
INC	direct	Increment direct byte	2	12
INC	@R_i	Increment direct RAM	1	12
DEC	A	Decrement Accumulator	1	12
DEC	R_n	Decrement Register	1	12
DEC	direct	Decrement direct byte	2	12
DEC	@R_i	Decrement indirect RAM	1	12
INC	DPTR	Increment Data Pointer	1	24
MUL	AB	Multiply A & B	1	48
DIV	AB	Divide A by B	1	48
DA	A	Decimal Adjust Accumulator	1	12

Note: 1. All mnemonics copyrighted © Intel Corp., 1980.

Mnemonic		Description	Byte	Oscillator Period
LOGICAL OPERATIONS				
ANL	A,R_n	AND Register to Accumulator	1	12
ANL	A,direct	AND direct byte to Accumulator	2	12
ANL	A,@R_i	AND indirect RAM to Accumulator	1	12
ANL	A,#data	AND immediate data to Accumulator	2	12
ANL	direct,A	AND Accumulator to direct byte	2	12
ANL	direct,#data	AND immediate data to direct byte	3	24
ORL	A,R_n	OR register to Accumulator	1	12
ORL	A,direct	OR direct byte to Accumulator	2	12
ORL	A,@R_i	OR indirect RAM to Accumulator	1	12
ORL	A,#data	OR immediate data to Accumulator	2	12
ORL	direct,A	OR Accumulator to direct byte	2	12
ORL	direct,#data	OR immediate data to direct byte	3	24
XRL	A,R_n	Exclusive-OR register to Accumulator	1	12
XRL	A,direct	Exclusive-OR direct byte to Accumulator	2	12
XRL	A,@R_i	Exclusive-OR indirect RAM to Accumulator	1	12
XRL	A,#data	Exclusive-OR immediate data to Accumulator	2	12
XRL	direct,A	Exclusive-OR Accumulator to direct byte	2	12
XRL	direct,#data	Exclusive-OR immediate data to direct byte	3	24
CLR	A	Clear Accumulator	1	12
CPL	A	Complement Accumulator	1	12
RL	A	Rotate Accumulator Left	1	12
RLC	A	Rotate Accumulator Left through the Carry	1	12
LOGICAL OPERATIONS (continued)				

Mnemonic		Description	Byte	Oscillator Period
RR	A	Rotate Accumulator Right	1	12
RRC	A	Rotate Accumulator Right through the Carry	1	12
SWAP	A	Swap nibbles within the Accumulator	1	12
DATA TRANSFER				
MOV	A,R_n	Move register to Accumulator	1	12
MOV	A,direct	Move direct byte to Accumulator	2	12
MOV	A,@R_i	Move indirect RAM to Accumulator	1	12
MOV	A,#data	Move immediate data to Accumulator	2	12
MOV	R_n,A	Move Accumulator to register	1	12
MOV	R_n,direct	Move direct byte to register	2	24
MOV	R_n,#data	Move immediate data to register	2	12
MOV	direct,A	Move Accumulator to direct byte	2	12
MOV	direct,R_n	Move register to direct byte	2	24
MOV	direct,direct	Move direct byte to direct	3	24
MOV	direct,@R_i	Move indirect RAM to direct byte	2	24
MOV	direct,#data	Move immediate data to direct byte	3	24
MOV	@R_i,A	Move Accumulator to indirect RAM	1	12
MOV	@R_i,direct	Move direct byte to indirect RAM	2	24
MOV	@R_i,#data	Move immediate data to indirect RAM	2	12
MOV	DPTR,#data16	Load Data Pointer with a 16-bit constant	3	24
MOVC	A,@A+DPTR	Move Code byte relative to DPTR to Acc	1	24
MOVC	A,@A+PC	Move Code byte relative to PC to Acc	1	24
MOVX	A,@R_i	Move External RAM (8-bit addr) to Acc	1	24
DATA TRANSFER (continued)				
MOVX	A,@DPTR	Move Exernal RAM (16-bit addr) to Acc	1	24
MOVX	@R_i,A	Move Acc to External RAM (8-bit addr)	1	24
MOVX	@DPTR,A	Move Acc to External RAM (16-bit addr)	1	24
PUSH	direct	Push direct byte onto stack	2	24
POP	direct	Pop direct byte from stack	2	24
XCH	A,R_n	Exchange register with Accumulator	1	12
XCH	A,direct	Exchange direct byte with Accumulator	2	12
XCH	A,@R_i	Exchange indirect RAM with Accumulator	1	12
XCHD	A,@R_i	Exchange low-order Digit indirect RAM with Acc	1	12
BOOLEAN VARIABLE MANIPULATION				
CLR	C	Clear Carry	1	12
CLR	bit	Clear direct bit	2	12
SETB	C	Set Carry	1	12
SETB	bit	Set direct bit	2	12
CPL	C	Complement Carry	1	12
CPL	bit	Complement direct bit	2	12
ANL	C,bit	AND direct bit to CARRY	2	24
ANL	C,/bit	AND complement of direct bit to Carry	2	24
ORL	C,bit	OR direct bit to Carry	2	24
ORL	C,/bit	OR complement of direct bit to Carry	2	24
MOV	C,bit	Move direct bit to Carry	2	12
MOV	bit,C	Move Carry to direct bit	2	24
JC	rel	Jump if Carry is set	2	24
JNC	rel	Jump if Carry not set	2	24
JB	bit,rel	Jump if direct Bit is set	3	24
JNB	bit,rel	Jump if direct Bit is Not set	3	24
JBC	bit,rel	Jump if direct Bit is set & clear bit	3	24
PROGRAM BRANCHING				

Mnemonic		Description	Byte	Oscillator Period
ACALL	addr11	Absolute Subroutine Call	2	24
LCALL	addr16	Long Subroutine Call	3	24
RET		Return from Subroutine	1	24
RETI		Return from interrupt	1	24
AJMP	addr11	Absolute Jump	2	24
LJMP	addr16	Long Jump	3	24
SJMP	rel	Short Jump (relative addr)	2	24
JMP	@A+DPTR	Jump indirect relative to the DPTR	1	24
JZ	rel	Jump if Accumulator is Zero	2	24
JNZ	rel	Jump if Accumulator is Not Zero	2	24
CJNE	A,direct,rel	Compare direct byte to Acc and Jump if Not Equal	3	24
CJNE	A,#data,rel	Compare immediate to Acc and Jump if Not Equal	3	24
CJNE	R_n,#data,rel	Compare immediate to register and Jump if Not Equal	3	24
CJNE	@R_i,#data,rel	Compare immediate to indirect and Jump if Not Equal	3	24
DJNZ	R_n,rel	Decrement register and Jump if Not Zero	2	24
DJNZ	direct,rel	Decrement direct byte and Jump if Not Zero	3	24
NOP		No Operation	1	12

Table 2. Instruction Opcodes in Hexadecimal Order

Hex Code	Number of Bytes	Mnemonic	Operands
00	1	NOP	
01	2	AJMP	code addr
02	3	LJMP	code addr
03	1	RR	A
04	1	INC	A
05	2	INC	data addr
06	1	INC	@R0
07	1	INC	@R1
08	1	INC	R0
09	1	INC	R1
0A	1	INC	R2
0B	1	INC	R3
0C	1	INC	R4
0D	1	INC	R5
0E	1	INC	R6
0F	1	INC	R7
10	3	JBC	bit addr,code addr
11	2	ACALL	code addr
12	3	LCALL	code addr
13	1	RRC	A
14	1	DEC	A
15	2	DEC	data addr
16	1	DEC	@R0
17	1	DEC	@R1
18	1	DEC	R0
19	1	DEC	R1
1A	1	DEC	R2
1B	1	DEC	R3
1C	1	DEC	R4
1D	1	DEC	R5
1E	1	DEC	R6
1F	1	DEC	R7
20	3	JB	bit addr,code addr
21	2	AJMP	code addr
22	1	RET	
23	1	RL	A
24	2	ADD	A,#data
25	2	ADD	A,data addr
26	1	ADD	A,@R0
27	1	ADD	A,@R1
28	1	ADD	A,R0
29	1	ADD	A,R1
2A	1	ADD	A,R2
2B	1	ADD	A,R3
2C	1	ADD	A,R4
2D	1	ADD	A,R5
2E	1	ADD	A,R6
2F	1	ADD	A,R7
30	3	JNB	bit addr,code addr
31	2	ACALL	code addr
32	1	RETI	
33	1	RLC	A
34	2	ADDC	A,#data
35	2	ADDC	A,data addr
36	1	ADDC	A,@R0
37	1	ADDC	A,@R1
38	1	ADDC	A,R0
39	1	ADDC	A,R1
3A	1	ADDC	A,R2
3B	1	ADDC	A,R3
3C	1	ADDC	A,R4
3D	1	ADDC	A,R5
3E	1	ADDC	A,R6
3F	1	ADDC	A,R7
40	2	JC	code addr
41	2	AJMP	code addr
42	2	ORL	data addr,A
43	3	ORL	data addr,#data
44	2	ORL	A,#data
45	2	ORL	A,data addr
46	1	ORL	A,@R0
47	1	ORL	A,@R1
48	1	ORL	A,R0
49	1	ORL	A,R1
4A	1	ORL	A,R2

Hex Code	Number of Bytes	Mnemonic	Operands
4B	1	ORL	A,R3
4C	1	ORL	A,R4
4D	1	ORL	A,R5
4E	1	ORL	A,R6
4F	1	ORL	A,R7
50	2	JNC	code addr
51	2	ACALL	code addr
52	2	ANL	data addr,A
53	3	ANL	data addr,#data
54	2	ANL	A,#data
55	2	ANL	A,data addr
56	1	ANL	A,@R0
57	1	ANL	A,@R1
58	1	ANL	A,R0
59	1	ANL	A,R1
5A	1	ANL	A,R2
5B	1	ANL	A,R3
5C	1	ANL	A,R4
5D	1	ANL	A,R5
5E	1	ANL	A,R6
5F	1	ANL	A,R7
60	2	JZ	code addr
61	2	AJMP	code addr
62	2	XRL	data addr,A
63	3	XRL	data addr,#data
64	2	XRL	A,#data
65	2	XRL	A,data addr
66	1	XRL	A,@R0
67	1	XRL	A,@R1
68	1	XRL	A,R0
69	1	XRL	A,R1
6A	1	XRL	A,R2
6B	1	XRL	A,R3
6C	1	XRL	A,R4
6D	1	XRL	A,R5
6E	1	XRL	A,R6
6F	1	XRL	A,R7
70	2	JNZ	code addr

Hex Code	Number of Bytes	Mnemonic	Operands
71	2	ACALL	code addr
72	2	ORL	C,bit addr
73	1	JMP	@A+DPTR
74	2	MOV	A,#data
75	3	MOV	data addr,#data
76	2	MOV	@R0,#data
77	2	MOV	@R1,#data
78	2	MOV	R0,#data
79	2	MOV	R1,#data
7A	2	MOV	R2,#data
7B	2	MOV	R3,#data
7C	2	MOV	R4,#data
7D	2	MOV	R5,#data
7E	2	MOV	R6,#data
7F	2	MOV	R7,#data
80	2	SJMP	code addr
81	2	AJMP	code addr
82	2	ANL	C,bit addr
83	1	MOVC	A,@A+PC
84	1	DIV	AB
85	3	MOV	data addr,data addr
86	2	MOV	data addr,@R0
87	2	MOV	data addr,@R1
88	2	MOV	data addr,R0
89	2	MOV	data addr,R1
8A	2	MOV	data addr,R2
8B	2	MOV	data addr,R3
8C	2	MOV	data addr,R4
8D	2	MOV	data addr,R5
8E	2	MOV	data addr,R6
8F	2	MOV	data addr,R7
90	3	MOV	DPTR,#data
91	2	ACALL	code addr
92	2	MOV	bit addr,C
93	1	MOVC	A,@A+DPTR
94	2	SUBB	A,#data
95	2	SUBB	A,data addr
96	1	SUBB	A,@R0

Hex Code	Number of Bytes	Mnemonic	Operands
97	1	SUBB	A,@R1
98	1	SUBB	A,R0
99	1	SUBB	A,R1
9A	1	SUBB	A,R2
9B	1	SUBB	A,R3
9C	1	SUBB	A,R4
9D	1	SUBB	A,R5
9E	1	SUBB	A,R6
9F	1	SUBB	A,R7
A0	2	ORL	C,/bit addr
A1	2	AJMP	code addr
A2	2	MOV	C,bit addr
A3	1	INC	DPTR
A4	1	MUL	AB
A5		reserved	
A6	2	MOV	@R0,data addr
A7	2	MOV	@R1,data addr
A8	2	MOV	R0,data addr
A9	2	MOV	R1,data addr
AA	2	MOV	R2,data addr
AB	2	MOV	R3,data addr
AC	2	MOV	R4,data addr
AD	2	MOV	R5,data addr
AE	2	MOV	R6,data addr
AF	2	MOV	R7,data addr
B0	2	ANL	C,/bit addr
B1	2	ACALL	code addr
B2	2	CPL	bit addr
B3	1	CPL	C
B4	3	CJNE	A,#data,code addr
B5	3	CJNE	A,data addr,code addr
B6	3	CJNE	@R0,#data,code addr
B7	3	CJNE	@R1,#data,code addr
B8	3	CJNE	R0,#data,code addr
B9	3	CJNE	R1,#data,code addr
BA	3	CJNE	R2,#data,code addr
BB	3	CJNE	R3,#data,code addr
BC	3	CJNE	R4,#data,code addr

Hex Code	Number of Bytes	Mnemonic	Operands
BD	3	CJNE	R5,#data,code addr
BE	3	CJNE	R6,#data,code addr
BF	3	CJNE	R7,#data,code addr
C0	2	PUSH	data addr
C1	2	AJMP	code addr
C2	2	CLR	bit addr
C3	1	CLR	C
C4	1	SWAP	A
C5	2	XCH	A,data addr
C6	1	XCH	A,@R0
C7	1	XCH	A,@R1
C8	1	XCH	A,R0
C9	1	XCH	A,R1
CA	1	XCH	A,R2
CB	1	XCH	A,R3
CC	1	XCH	A,R4
CD	1	XCH	A,R5
CE	1	XCH	A,R6
CF	1	XCH	A,R7
D0	2	POP	data addr
D1	2	ACALL	code addr
D2	2	SETB	bit addr
D3	1	SETB	C
D4	1	DA	A
D5	3	DJNZ	data addr,code addr
D6	1	XCHD	A,@R0
D7	1	XCHD	A,@R1
D8	2	DJNZ	R0,code addr
D9	2	DJNZ	R1,code addr
DA	2	DJNZ	R2,code addr
DB	2	DJNZ	R3,code addr
DC	2	DJNZ	R4,code addr
DD	2	DJNZ	R5,code addr
DE	2	DJNZ	R6,code addr
DF	2	DJNZ	R7,code addr
E0	1	MOVX	A,@DPTR
E1	2	AJMP	code addr
E2	1	MOVX	A,@R0

Hex Code	Number of Bytes	Mnemonic	Operands
E3	1	MOVX	A,@R1
E4	1	CLR	A
E5	2	MOV	A,data addr
E6	1	MOV	A,@R0
E7	1	MOV	A,@R1
E8	1	MOV	A,R0
E9	1	MOV	A,R1
EA	1	MOV	A,R2
EB	1	MOV	A,R3
EC	1	MOV	A,R4
ED	1	MOV	A,R5
EE	1	MOV	A,R6
EF	1	MOV	A,R7
F0	1	MOVX	@DPTR,A
F1	2	ACALL	code addr
F2	1	MOVX	@R0,A
F3	1	MOVX	@R1,A
F4	1	CPL	A
F5	2	MOV	data addr,A
F6	1	MOV	@R0,A
F7	1	MOV	@R1,A
F8	1	MOV	R0,A
F9	1	MOV	R1,A
FA	1	MOV	R2,A
FB	1	MOV	R3,A
FC	1	MOV	R4,A
FD	1	MOV	R5,A
FE	1	MOV	R6,A
FF	1	MOV	R7,A

Instruction Definitions

ACALL addr11

Function: Absolute Call

Description: ACALL unconditionally calls a subroutine located at the indicated address. The instruction increments the PC twice to obtain the address of the following instruction, then pushes the 16-bit result onto the stack (low-order byte first) and increments the Stack Pointer twice. The destination address is obtained by successively concatenating the five high-order bits of the incremented PC, opcode bits 7 through 5, and the second byte of the instruction. The subroutine called must therefore start within the same 2 K block of the program memory as the first byte of the instruction following ACALL. No flags are affected.

Example: Initially SP equals 07H. The label SUBRTN is at program memory location 0345 H. After executing the following instruction,

```
ACALL     SUBRTN
```

at location 0123H, SP contains 09H, internal RAM locations 08H and 09H will contain 25H and 01H, respectively, and the PC contains 0345H.

Bytes: 2

Cycles: 2

Encoding:

a10	a9	a8	1	0	0	0	1	a7	a6	a5	a4	a3	a2	a1	a0

Operation: ACALL
(PC) ← (PC) + 2
(SP) ← (SP) + 1
((SP)) ← (PC_{7-0})
(SP) ← (SP) + 1
((SP)) ← (PC_{15-8})
(PC_{10-0}) ← page address

ADD A,<src-byte>

Function: Add

Description: ADD adds the byte variable indicated to the Accumulator, leaving the result in the Accumulator. The carry and auxiliary-carry flags are set, respectively, if there is a carry-out from bit 7 or bit 3, and cleared otherwise. When adding unsigned integers, the carry flag indicates an overflow occurred.

OV is set if there is a carry-out of bit 6 but not out of bit 7, or a carry-out of bit 7 but not bit 6; otherwise, OV is cleared. When adding signed integers, OV indicates a negative number produced as the sum of two positive operands, or a positive sum from two negative operands.

Four source operand addressing modes are allowed: register, direct, register-indirect, or immediate.

Example: The Accumulator holds 0C3H (1100001lB), and register 0 holds 0AAH (10101010B). The following instruction,

```
ADD         A,R0
```

leaves 6DH (01101101B) in the Accumulator with the AC flag cleared and both the carry flag and OV set to 1.

ADD A,R_n

Bytes: 1

Cycles: 1

Encoding:

0	0	1	0	1	r	r	r

Operation: ADD
(A) ← (A) + (R_n)

ADD A,direct

Bytes: 2

Cycles: 1

Encoding:

0	0	1	0	0	1	0	1	direct address

Operation: ADD
(A) ← (A) + (direct)

ADD A,@R_i

Bytes: 1

Cycles: 1

Encoding:

0	0	1	0	0	1	1	i

Operation: ADD
(A) ← (A) + ((R_i))

ADD A,#data

Bytes: 2

Cycles: 1

Encoding:

0	0	1	0	0	1	0	0	immediate data

Operation: ADD
(A) ← (A) + #data

ADDC A, <src-byte>

Function: Add with Carry

Description: ADDC simultaneously adds the byte variable indicated, the carry flag and the Accumulator contents, leaving the result in the Accumulator. The carry and auxiliary-carry flags are set respectively, if there is a carry-out from bit 7 or bit 3, and cleared otherwise. When adding unsigned integers, the carry flag indicates an overflow occurred.

OV is set if there is a carry-out of bit 6 but not out of bit 7, or a carry-out of bit 7 but not out of bit 6; otherwise OV is cleared. When adding signed integers, OV indicates a negative number produced as the sum of two positive operands or a positive sum from two negative operands.

Four source operand addressing modes are allowed: register, direct, register-indirect, or immediate.

Example: The Accumulator holds 0C3H (11000011B) and register 0 holds 0AAH (10101010B) with the carry flag set. The following instruction,

```
ADDC    A,R0
```

leaves 6EH (01101110B) in the Accumulator with AC cleared and both the Carry flag and OV set to 1.

ADDC A,R_n

Bytes: 1

Cycles: 1

Encoding:

0	0	1	1	1	r	r	r

Operation: ADDC
(A) ← (A) + (C) + (R_n)

ADDC A,direct

Bytes: 2

Cycles: 1

Encoding:

0	0	1	1	0	1	0	1	direct address

Operation: ADDC
(A) ← (A) + (C) + (direct)

ADDC A,@R_i

Bytes: 1

Cycles: 1

Encoding:

0	0	1	1	0	1	1	i

Operation: ADDC
(A) ← (A) + (C) + ((R_i))

ADDC A,#data

Bytes: 2

Cycles: 1

Encoding:

0	0	1	1	0	1	0	0	immediate data

Operation: ADDC
(A) ← (A) + (C) + #data

AJMP addr11

Function: Absolute Jump

Description: AJMP transfers program execution to the indicated address, which is formed at run-time by concatenating the high-order five bits of the PC (after incrementing the PC twice), opcode bits 7 through 5, and the second byte of the instruction. The destination must therfore be within the same 2 K block of program memory as the first byte of the instruction following AJMP.

Example: The label JMPADR is at program memory location 0123H. The following instruction,

AJMP JMPADR

is at location 0345H and loads the PC with 0123H.

Bytes: 2

Cycles: 2

Encoding:

a10	a9	a8	0	0	0	0	1	a7	a6	a5	a4	a3	a2	a1	a0

Operation: AJMP
(PC) ← (PC) + 2
(PC_{10-0}) ← page address

ANL <dest-byte>,<src-byte>

Function: Logical-AND for byte variables

Description: ANL performs the bitwise logical-AND operation between the variables indicated and stores the results in the destination variable. No flags are affected.

The two operands allow six addressing mode combinations. When the destination is the Accumulator, the source can use register, direct, register-indirect, or immediate addressing; when the destination is a direct address, the source can be the Accumulator or immediate data.

Note: When this instruction is used to modify an output port, the value used as the original port data will be read from the output data latch, *not* the input pins.

Example: If the Accumulator holds 0C3H (1100001lB), and register 0 holds 55H (01010101B), then the following instruction,

ANL A,R0

leaves 41H (01000001B) in the Accumulator.

When the destination is a directly addressed byte, this instruction clears combinations of bits in any RAM location or hardware register. The mask byte determining the pattern of bits to be cleared would either be a constant contained in the instruction or a value computed in the Accumulator at run-time. The following instruction,

ANL P1,#01110011B

clears bits 7, 3, and 2 of output port 1.

ANL A,R_n

Bytes: 1

Cycles: 1

Encoding:

0	1	0	1	1	r	r	r

Operation: ANL
(A) ← (A) ∧ (R_n)

ANL A,direct

Bytes: 2

Cycles: 1

Encoding:

0	1	0	1	0	1	0	1	direct address

Operation: ANL
(A) ← (A) ∧ (direct)

ANL A,@R_i

Bytes: 1

Cycles: 1

Encoding:

0	1	0	1	0	1	1	i

Operation: ANL
(A) ← (A) ∧ ((R_i))

ANL A,#data

Bytes: 2

Cycles: 1

Encoding:

0	1	0	1	0	1	0	0	immediate data

Operation: ANL
(A) ← (A) ∧ #data

ANL direct,A

Bytes: 2

Cycles: 1

Encoding:

0	1	0	1	0	0	1	0	direct address

Operation: ANL
(direct) ← (direct) ∧ (A)

ANL direct,#data

Bytes: 3

Cycles: 2

Encoding:

0	1	0	1	0	0	1	1	direct address	immediate data

Operation: ANL
(direct) ← (direct) ∧ #data

ANL C,<src-bit>

Function: Logical-AND for bit variables

Description: If the Boolean value of the source bit is a logical 0, then ANL C clears the carry flag; otherwise, this instruction leaves the carry flag in its current state. A slash (/) preceding the operand in the assembly language indicates that the logical complement of the addressed bit is used as the source value, *but the source bit itself is not affected.* No other flags are affected.

Only direct addressing is allowed for the source operand.

Example: Set the carry flag if, and only if, P1.0 = 1, ACC.7 = 1, and OV = 0:

```
MOV     C,P1.0      ;LOAD CARRY WITH INPUT PIN STATE
ANL     C,ACC.7     ;AND CARRY WITH ACCUM. BIT 7
ANL     C,/OV       ;AND WITH INVERSE OF OVERFLOW FLAG
```

ANL C,bit

Bytes: 2

Cycles: 2

Encoding:

1	0	0	0	0	0	1	0	bit address

Operation: ANL
(C) ← (C) ∧ (bit)

ANL C,/bit

Bytes: 2

Cycles: 2

Encoding:

1	0	1	1	0	0	0	0	bit address

Operation: ANL
(C) ← (C) ∧ ¬ (bit)

CJNE <dest-byte>,<src-byte>, rel

Function: Compare and Jump if Not Equal.

Description: CJNE compares the magnitudes of the first two operands and branches if their values are not equal. The branch destination is computed by adding the signed relative-displacement in the last instruction byte to the PC, after incrementing the PC to the start of the next instruction. The carry flag is set if the unsigned integer value of <dest-byte> is less than the unsigned integer value of <src-byte>; otherwise, the carry is cleared. Neither operand is affected.

The first two operands allow four addressing mode combinations: the Accumulator may be compared with any directly addressed byte or immediate data, and any indirect RAM location or working register can be compared with an immediate constant.

Example: The Accumulator contains 34H. Register 7 contains 56H. The first instruction in the sequence,

```
                CJNE    R7, # 60H, NOT_EQ
;               . . .   . . . . .       ;R7 = 60H.
NOT_EQ:         JC      REQ_LOW         ;IF R7 < 60H.
;               . . .   . . . . .       ;R7 > 60H.
```

sets the carry flag and branches to the instruction at label NOT_EQ. By testing the carry flag, this instruction determines whether R7 is greater or less than 60H.

If the data being presented to Port 1 is also 34H, then the following instruction,

```
WAIT:   CJNE    A, P1,WAIT
```

clears the carry flag and continues with the next instruction in sequence, since the Accumulator does equal the data read from P1. (If some other value was being input on P1, the program loops at this point until the P1 data changes to 34H.)

CJNE A,direct,rel

Bytes: 3

Cycles: 2

Encoding:

1	0	1	1	0	1	0	1	direct address	rel. address

Operation:

```
(PC) ← (PC) + 3
IF (A) < > (direct)
THEN
    (PC) ← (PC) + relative offset
IF (A) < (direct)
THEN
    (C) ← 1
ELSE
    (C) ← 0
```

CJNE A,#data,rel

Bytes: 3

Cycles: 2

Encoding:

1	0	1	1	0	1	0	0	immediate data	rel. address

Operation: (PC) ← (PC) + 3
IF (A) < > *data*
THEN
    (PC) ← (PC) + *relative offset*
IF (A) < *data*
THEN
    (C) ← 1
ELSE
    (C) ← 0

CJNE R_n,#data,rel

Bytes: 3

Cycles: 2

Encoding:

1	0	1	1	1	r	r	r	immediate data	rel. address

Operation: (PC) ← (PC) + 3
IF (R_n) < > *data*
THEN
    (PC) ← (PC) + *relative offset*
IF (R_n) < *data*
THEN
    (C) ← 1
ELSE
    (C) ← 0

CJNE @R_i,data,rel

Bytes: 3

Cycles: 2

Encoding:

1	0	1	1	0	1	1	i	immediate data	rel. address

Operation: (PC) ← (PC) + 3
IF ((R_i)) < > *data*
THEN
    (PC) ← (PC) + *relative offset*
IF ((R_i)) < *data*
THEN
    (C) ← 1
ELSE
    (C) ← 0

CLR A

Function: Clear Accumulator

Description: CLR A clears the Accumulator (all bits set to 0). No flags are affected

Example: The Accumulator contains 5CH (01011100B). The following instruction,CLR Aleaves the Accumulator set to 00H (00000000B).

Bytes: 1

Cycles: 1

Encoding:

1	1	1	0	0	1	0	0

Operation: CLR
(A) ← 0

CLR bit

Function: Clear bit

Description: CLR bit clears the indicated bit (reset to 0). No other flags are affected. CLR can operate on the carry flag or any directly addressable bit.

Example: Port 1 has previously been written with 5DH (01011101B). The following instruction,CLR P1.2 leaves the port set to 59H (01011001B).

CLR C

Bytes: 1

Cycles: 1

Encoding:

1	1	0	0	0	0	1	1

Operation: CLR
(C) ← 0

CLR bit

Bytes: 2

Cycles: 1

Encoding:

1	1	0	0	0	0	1	0	bit address

Operation: CLR
(bit) ← 0

CPL A

Function: Complement Accumulator

Description: CPLA logically complements each bit of the Accumulator (one's complement). Bits which previously contained a 1 are changed to a 0 and vice-versa. No flags are affected.

Example: The Accumulator contains 5CH (01011100B). The following instruction,

```
CPL     A
```

leaves the Accumulator set to 0A3H (10100011B).

Bytes: 1

Cycles: 1

Encoding:

1	1	1	1	0	1	0	0

Operation: CPL
(A) ← ¬ (A)

CPL bit

Function: Complement bit

Description: CPL bit complements the bit variable specified. A bit that had been a 1 is changed to 0 and vice-versa. No other flags are affected. CLR can operate on the carry or any directly addressable bit.

Note: When this instruction is used to modify an output pin, the value used as the original data is read from the output data latch, *not* the input pin.

Example: Port 1 has previously been written with 5BH (01011101B). The following instruction sequence,CPL P1.1CPL P1.2 leaves the port set to 5BH (01011011B).

CPL C

Bytes: 1

Cycles: 1

Encoding:

1	0	1	1	0	0	1	1

Operation: CPL
(C) ← ¬ (C)

CPL bit

Bytes: 2

Cycles: 1

Encoding:

1	0	1	1	0	0	1	0	bit address

Operation: CPL
(bit) ← ¬ (bit)

DA A

Function: Decimal-adjust Accumulator for Addition

Description: DA A adjusts the eight-bit value in the Accumulator resulting from the earlier addition of two variables (each in packed-BCD format), producing two four-bit digits. Any ADD or ADDC instruction may have been used to perform the addition.

If Accumulator bits 3 through 0 are greater than nine (xxxx1010-xxxx1111), or if the AC flag is one, six is added to the Accumulator producing the proper BCD digit in the low-order nibble. This internal addition sets the carry flag if a carry-out of the low-order four-bit field propagates through all high-order bits, but it does not clear the carry flag otherwise.

If the carry flag is now set, or if the four high-order bits now exceed nine (1010xxxx-1111xxxx), these high-order bits are incremented by six, producing the proper BCD digit in the high-order nibble. Again, this sets the carry flag if there is a carry-out of the high-order bits, but does not clear the carry. The carry flag thus indicates if the sum of the original two BCD variables is greater than 100, allowing multiple precision decimal addition. OV is not affected.

All of this occurs during the one instruction cycle. Essentially, this instruction performs the decimal conversion by adding 00H, 06H, 60H, or 66H to the Accumulator, depending on initial Accumulator and PSW conditions.

Note: DA A *cannot* simply convert a hexadecimal number in the Accumulator to BCD notation, nor does DAA apply to decimal subtraction.

Example: The Accumulator holds the value 56H (01010110B), representing the packed BCD digits of the decimal number 56. Register 3 contains the value 67H (01100111B), representing the packed BCD digits of the decimal number 67. The carry flag is set. The following instruction sequence

```
ADDC    A,R3
DA      A
```

first performs a standard two's-complement binary addition, resulting in the value 0BEH (10111110) in the Accumulator. The carry and auxiliary carry flags are cleared.

The Decimal Adjust instruction then alters the Accumulator to the value 24H (00100100B), indicating the packed BCD digits of the decimal number 24, the low-order two digits of the decimal sum of 56, 67, and the carry-in. The carry flag is set by the Decimal Adjust instruction, indicating that a decimal overflow occurred. The true sum of 56, 67, and 1 is 124.

BCD variables can be incremented or decremented by adding 01H or 99H. If the Accumulator initially holds 30H (representing the digits of 30 decimal), then the following instruction sequence,

```
ADD     A, # 99H
DA      A
```

leaves the carry set and 29H in the Accumulator, since 30 + 99 = 129. The low-order byte of the sum can be interpreted to mean 30 - 1 = 29.

Bytes: 1

Cycles: 1

Encoding:

1	1	0	1	0	1	0	0

Operation: DA
-contents of Accumulator are BCD
IF [[(A_{3-0}) > 9] $\vee$ [(AC) = 1]]
THEN $(A_{3-0}) \leftarrow (A_{3-0}) + 6$
AND
IF [[(A_{7-4}) > 9] $\vee$ [(C) = 1]]
THEN $(A_{7-4}) \leftarrow (A_{7-4}) + 6$

DEC byte

Function: Decrement

Description: DEC byte decrements the variable indicated by 1. An original value of 00H underflows to 0FFH. No flags are affected. Four operand addressing modes are allowed: accumulator, register, direct, or register-indirect.

Note: When this instruction is used to modify an output port, the value used as the original port data will be read from the output data latch, *not* the input pins.

Example: Register 0 contains 7FH (01111111B). Internal RAM locations 7EH and 7FH contain 00H and 40H, respectively. The following instruction sequence,

```
DEC     @R0
DEC     R0
DEC     @R0
```

leaves register 0 set to 7EH and internal RAM locations 7EH and 7FH set to 0FFH and 3FH.

DEC A

Bytes: 1

Cycles: 1

Encoding:

0	0	0	1	0	1	0	0

Operation: DEC
(A) ← (A) - 1

DEC R_n

Bytes: 1

Cycles: 1

Encoding:

0	0	0	1	1	r	r	r

Operation: DEC
(R_n) ← (R_n) - 1

DEC direct

Bytes: 2

Cycles: 1

Encoding:

0	0	0	1	0	1	0	1	direct address

Operation: DEC
(direct) ← (direct) - 1

DEC @R_i

Bytes: 1

Cycles: 1

Encoding:

0	0	0	1	0	1	1	i

Operation: DEC
((R_i)) ← ((R_i)) - 1

DIV AB

Function: Divide

Description: DIV AB divides the unsigned eight-bit integer in the Accumulator by the unsigned eight-bit integer in register B. The Accumulator receives the integer part of the quotient; register B receives the integer remainder. The carry and OV flags are cleared.

Exception: if B had originally contained 00H, the values returned in the Accumulator and B-register are undefined and the overflow flag are set. The carry flag is cleared in any case.

Example: The Accumulator contains 251 (0FBH or 11111011B) and B contains 18 (12H or 00010010B). The following instruction,

DIV AB

leaves 13 in the Accumulator (0DH or 00001101B) and the value 17 (11H or 00010001B) in B, since 251 = (13 x 18) + 17. Carry and OV are both cleared.

Bytes: 1

Cycles: 4

Encoding:

1	0	0	0	0	1	0	0

Operation: DIV

$(A)_{15-8} \leftarrow (A)/(B)$

$(B)_{7-0}$

DJNZ <byte>,<rel-addr>

Function: Decrement and Jump if Not Zero

Description: DJNZ decrements the location indicated by 1, and branches to the address indicated by the second operand if the resulting value is not zero. An original value of 00H underflows to 0FFH. No flags are affected. The branch destination is computed by adding the signed relative-displacement value in the last instruction byte to the PC, after incrementing the PC to the first byte of the following instruction.

The location decremented may be a register or directly addressed byte.

Note: When this instruction is used to modify an output port, the value used as the original port data will be read from the output data latch, *not* the input pins.

Example: Internal RAM locations 40H, 50H, and 60H contain the values 01H, 70H, and 15H, respectively. The following instruction sequence,

```
DJNZ    40H,LABEL_1
DJNZ    50H,LABEL_2
DJNZ    60H,LABEL_3
```

causes a jump to the instruction at label LABEL_2 with the values 00H, 6FH, and 15H in the three RAM locations. The first jump was *not* taken because the result was zero.

This instruction provides a simple way to execute a program loop a given number of times or for adding a moderate time delay (from 2 to 512 machine cycles) with a single instruction. The following instruction sequence,

```
        MOV     R2, # 8
TOGGLE: CPL     P1.7
        DJNZ    R2,TOGGLE
```

toggles P1.7 eight times, causing four output pulses to appear at bit 7 of output Port 1. Each pulse lasts three machine cycles; two for DJNZ and one to alter the pin.

DJNZ R_n,rel

Bytes: 2

Cycles: 2

Encoding:

1	1	0	1	1	r	r	r	rel. address	

Operation: DJNZ
(PC) ← (PC) + 2
(R_n) ← (R_n) - 1
IF (R_n) > 0 or (R_n) < 0
THEN
(PC) ← (PC) + rel

DJNZ direct,rel

Bytes: 3

Cycles: 2

Encoding:

1	1	0	1	0	1	0	1	direct address	rel. address

Operation: DJNZ
(PC) ← (PC) + 2
(direct) ← (direct) - 1
IF (direct) > 0 or (direct) < 0
THEN
(PC) ← (PC) + rel

INC <byte>

Function: Increment

Description: INC increments the indicated variable by 1. An original value of 0FFH overflows to 00H. No flags are affected. Three addressing modes are allowed: register, direct, or register-indirect.

Note: When this instruction is used to modify an output port, the value used as the original port data will be read from the output data latch, *not* the input pins.

Example: Register 0 contains 7EH (011111110B). Internal RAM locations 7EH and 7FH contain 0FFH and 40H, respectively. The following instruction sequence,

```
INC     @R0
INC     R0
INC     @R0
```

leaves register 0 set to 7FH and internal RAM locations 7EH and 7FH holding 00H and 41H, respectively.

INC A

Bytes: 1

Cycles: 1

Encoding:

0	0	0	0	0	1	0	0

Operation: INC
(A) ← (A) + 1

INC R_n

Bytes: 1

Cycles: 1

Encoding:

0	0	0	0	1	r	r	r

Operation: INC
(R_n) ← (R_n) + 1

INC direct

Bytes: 2

Cycles: 1

Encoding:

0	0	0	0	0	1	0	1	direct address

Operation: INC
(direct) ← (direct) + 1

INC @R_i

Bytes: 1

Cycles: 1

Encoding:

0	0	0	0	0	1	1	i

Operation: INC
((R_i)) ← ((R_i)) + 1

INC DPTR

Function: Increment Data Pointer

Description: INC DPTR increments the 16-bit data pointer by 1. A 16-bit increment (modulo 2^{16}) is performed, and an overflow of the low-order byte of the data pointer (DPL) from 0FFH to 00H increments the high-order byte (DPH). No flags are affected.

This is the only 16-bit register which can be incremented.

Example: Registers DPH and DPL contain 12H and 0FEH, respectively. The following instruction sequence,

```
INC     DPTR
INC     DPTR
INC     DPTR
```

changes DPH and DPL to 13H and 01H.

Bytes: 1

Cycles: 2

Encoding:

1	0	1	0	0	0	1	1

Operation: INC

(DPTR) ← (DPTR) + 1

JB blt,rel

Function: Jump if Bit set

Description: If the indicated bit is a one, JB jump to the address indicated; otherwise, it proceeds with the next instruction. The branch destination is computed by adding the signed relative-displacement in the third instruction byte to the PC, after incrementing the PC to the first byte of the next instruction. The bit tested is not modified. No flags are affected.

Example: The data present at input port 1 is 11001010B. The Accumulator holds 56 (01010110B). The following instruction sequence,

```
JB      P1.2,LABEL1
JB      ACC. 2,LABEL2
```

causes program execution to branch to the instruction at label LABEL2.

Bytes: 3

Cycles: 2

Encoding:

0	0	1	0	0	0	0	0	bit address	rel. address

Operation: JB

(PC) ← (PC) + 3

IF (bit) = 1

THEN

(PC) ← (PC) + rel

JBC bit,rel

Function: Jump if Bit is set and Clear bit

Description: If the indicated bit is one, JBC branches to the address indicated; otherwise, it proceeds with the next instruction. *The bit will not be cleared if it is already a zero*. The branch destination is computed by adding the signed relative-displacement in the third instruction byte to the PC, after incrementing the PC to the first byte of the next instruction. No flags are affected.

Note: When this instruction is used to test an output pin, the value used as the original data will be read from the output data latch, *not* the input pin.

Example: The Accumulator holds 56H (01010110B). The following instruction sequence,

```
JBC     ACC.3,LABEL1
JBC     ACC.2,LABEL2
```

causes program execution to continue at the instruction identified by the label LABEL2, with the Accumulator modified to 52H (01010010B).

Bytes: 3

Cycles: 2

Encoding:

0	0	0	1	0	0	0	0	bit address	rel. address

Operation:

```
JBC
(PC) ← (PC) + 3
IF   (bit) = 1
     THEN
          (bit) ← 0
          (PC) ← (PC) +rel
```

JC rel

Function: Jump if Carry is set

Description: If the carry flag is set, JC branches to the address indicated; otherwise, it proceeds with the next instruction. The branch destination is computed by adding the signed relative-displacement in the second instruction byte to the PC, after incrementing the PC twice. No flags are affected.

Example: The carry flag is cleared. The following instruction sequence,

```
JC      LABEL1
CPL     C
 JC     LABEL 2
```

sets the carry and causes program execution to continue at the instruction identified by the label LABEL2.

Bytes: 2

Cycles: 2

Encoding:

0	1	0	0	0	0	0	0	rel. address

Operation:

```
JC
(PC) ← (PC) + 2
IF   (C) = 1
     THEN
          (PC) ← (PC) + rel
```

JMP @A+DPTR

Function: Jump indirect

Description: JMP @A+DPTR adds the eight-bit unsigned contents of the Accumulator with the 16-bit data pointer and loads the resulting sum to the program counter. This is the address for subsequent instruction fetches. Sixteen-bit addition is performed (modulo 2^{16}): a carry-out from the low-order eight bits propagates through the higher-order bits. Neither the Accumulator nor the Data Pointer is altered. No flags are affected.

Example: An even number from 0 to 6 is in the Accumulator. The following sequence of instructions branches to one of four AJMP instructions in a jump table starting at JMP_TBL.

```
        MOV     DPTR, # JMP_TBL
        JMP     @A + DPTR
JMP_TBL: AJMP   LABEL0
        AJMP    LABEL1
        AJMP    LABEL2
        AJMP    LABEL3
```

If the Accumulator equals 04H when starting this sequence, execution jumps to label LABEL2. Because AJMP is a 2-byte instruction, the jump instructions start at every other address.

Bytes: 1

Cycles: 2

Encoding:

0	1	1	1	0	0	1	1

Operation: JMP
(PC) ← (A) + (DPTR)

JNB bit,rel

Function: Jump if Bit Not set

Description: If the indicated bit is a 0, JNB branches to the indicated address; otherwise, it proceeds with the next instruction. The branch destination is computed by adding the signed relative-displacement in the third instruction byte to the PC, after incrementing the PC to the first byte of the next instruction. *The bit tested is not modified.* No flags are affected.

Example: The data present at input port 1 is 11001010B. The Accumulator holds 56H (01010110B). The following instruction sequence,

```
JNB     P1.3,LABEL1
JNB     ACC.3,LABEL2
```

causes program execution to continue at the instruction at label LABEL2.

Bytes: 3

Cycles: 2

Encoding:

0	0	1	1	0	0	0	0	bit address	rel. address

Operation: JNB
(PC) ← (PC) + 3
IF (bit) = 0
THEN (PC) ← (PC) + rel

JNC rel

Function: Jump if Carry not set

Description: If the carry flag is a 0, JNC branches to the address indicated; otherwise, it proceeds with the next instruction. The branch destination is computed by adding the signal relative-displacement in the second instruction byte to the PC, after incrementing the PC twice to point to the next instruction. The carry flag is not modified.

Example: The carry flag is set. The following instruction sequence,

```
JNC     LABEL1
CPL     C
JNC     LABEL2
```

clears the carry and causes program execution to continue at the instruction identified by the label LABEL2.

Bytes: 2

Cycles: 2

Encoding:

0	1	0	1	0	0	0	0	rel. address

Operation: JNC
(PC) ← (PC) + 2
IF (C) = 0
THEN (PC) ← (PC) + rel

JNZ rel

Function: Jump if Accumulator Not Zero

Description: If any bit of the Accumulator is a one, JNZ branches to the indicated address; otherwise, it proceeds with the next instruction. The branch destination is computed by adding the signed relative-displacement in the second instruction byte to the PC, after incrementing the PC twice. The Accumulator is not modified. No flags are affected.

Example: The Accumulator originally holds 00H. The following instruction sequence,

```
JNZ     LABEL1
INC     A
JNZ     LABEL2
```

sets the Accumulator to 01H and continues at label LABEL2.

Bytes: 2

Cycles: 2

Encoding:

0	1	1	1	0	0	0	0	rel. address

Operation: JNZ
(PC) ← (PC) + 2
IF (A) ≠ 0
THEN (PC) ← (PC) + rel

JZ rel

Function: Jump if Accumulator Zero

Description: If all bits of the Accumulator are 0, JZ branches to the address indicated; otherwise, it proceeds with the next instruction. The branch destination is computed by adding the signed relative-displacement in the second instruction byte to the PC, after incrementing the PC twice. The Accumulator is not modified. No flags are affected.

Example: The Accumulator originally contains 01H. The following instruction sequence,

```
JZ      LABEL1
DEC     A
JZ      LABEL2
```

changes the Accumulator to 00H and causes program execution to continue at the instruction identified by the label LABEL2.

Bytes: 2

Cycles: 2

Encoding:

0	1	1	0	0	0	0	0	rel. address

Operation: JZ
(PC) ← (PC) + 2
IF (A) = 0
THEN (PC) ← (PC) + rel

LCALL addr16

Function: Long call

Description: LCALL calls a subroutine located at the indicated address. The instruction adds three to the program counter to generate the address of the next instruction and then pushes the 16-bit result onto the stack (low byte first), incrementing the Stack Pointer by two. The high-order and low-order bytes of the PC are then loaded, respectively, with the second and third bytes of the LCALL instruction. Program execution continues with the instruction at this address. The subroutine may therefore begin anywhere in the full 64K byte program memory address space. No flags are affected.

Example: Initially the Stack Pointer equals 07H. The label SUBRTN is assigned to program memory location 1234H. After executing the instruction,

```
LCALL    SUBRTN
```

at location 0123H, the Stack Pointer will contain 09H, internal RAM locations 08H and 09H will contain 26H and 01H, and the PC will contain 1234H.

Bytes: 3

Cycles: 2

Encoding:

0	0	0	1	0	0	1	0	addr15-addr8	addr7-addr0

Operation: LCALL
(PC) ← (PC) + 3
(SP) ← (SP) + 1
((SP)) ← (PC_{7-0})
(SP) ← (SP) + 1
((SP)) ← (PC_{15-8})
(PC) ← $addr_{15-0}$

LJMP addr16

Function: Long Jump

Description: LJMP causes an unconditional branch to the indicated address, by loading the high-order and low-order bytes of the PC (respectively) with the second and third instruction bytes. The destination may therefore be anywhere in the full 64K program memory address space. No flags are affected.

Example: The label JMPADR is assigned to the instruction at program memory location 1234H. The instruction,

```
LJMP     JMPADR
```

at location 0123H will load the program counter with 1234H.

Bytes: 3

Cycles: 2

Encoding:

0	0	0	0	0	0	1	0	addr15-addr8	addr7-addr0

Operation: LJMP
(PC) ← $addr_{15-0}$

MOV <dest-byte>,<src-byte>

Function: Move byte variable

Description: The byte variable indicated by the second operand is copied into the location specified by the first operand. The source byte is not affected. No other register or flag is affected.

This is by far the most flexible operation. Fifteen combinations of source and destination addressing modes are allowed.

Example: Internal RAM location 30H holds 40H. The value of RAM location 40H is 10H. The data present at input port 1 is 11001010B (0CAH).

```
MOV     R0,#30H     ;R0 < = 30H
MOV     A,@R0       ;A < = 40H
MOV     R1,A        ;R1 < = 40H
MOV     B,@R1       ;B < = 10H
MOV     @R1,P1      ;RAM (40H) < = 0CAH
MOV     P2,P1       ;P2 #0CAH
```

leaves the value 30H in register 0, 40H in both the Accumulator and register 1, 10H in register B, and 0CAH (11001010B) both in RAM location 40H and output on port 2.

MOV A,R_n

Bytes: 1

Cycles: 1

Encoding:

1	1	1	0	1	r	r	r

Operation: MOV
(A) ← (R_n)

*MOV A,direct

Bytes: 2

Cycles: 1

Encoding:

1	1	1	0	0	1	0	1	direct address

Operation: MOV
(A) ← (direct)

* MOV A,ACC is not a valid Instruction.

MOV A,@R_i

Bytes: 1

Cycles: 1

Encoding:

1	1	1	0	0	1	1	i

Operation: MOV
(A) ← ((R_i))

MOV A,#data

Bytes: 2

Cycles: 1

Encoding:

0	1	1	1	0	1	0	0	immediate data

Operation: MOV
(A) ← #data

MOV R_n,A

Bytes: 1

Cycles: 1

Encoding:

1	1	1	1	1	r	r	r

Operation: MOV
(R_n) ← (A)

MOV R_n,direct

Bytes: 2

Cycles: 2

Encoding:

1	0	1	0	1	r	r	r	direct addr.

Operation: MOV
(R_n) ← (direct)

MOV R_n,#data

Bytes: 2

Cycles: 1

Encoding:

0	1	1	1	1	r	r	r	immediate data

Operation: MOV
(R_n) ← #data

MOV direct,A

Bytes: 2

Cycles: 1

Encoding:

1	1	1	1	0	1	0	1	direct address

Operation: MOV
(direct) ← (A)

MOV direct,R_n

Bytes: 2

Cycles: 2

Encoding:

1	0	0	0	1	r	r	r	direct address

Operation: MOV
(direct) ← (R_n)

MOV direct,direct

Bytes: 3

Cycles: 2

Encoding:

1	0	0	0	0	1	0	1	dir. addr. (scr)	dir. addr. (dest)

Operation: MOV
(direct) ← (direct)

MOV direct,@R_i

Bytes: 2

Cycles: 2

Encoding:

1	0	0	0	0	1	1	i	direct addr.

Operation: MOV
(direct) ← ((R_i))

MOV direct,#data

Bytes: 3

Cycles: 2

Encoding:

0	1	1	1	0	1	0	1	direct address	immediate data

Operation: MOV
(direct) ← #data

MOV @R_i,A

Bytes: 1

Cycles: 1

Encoding:

1	1	1	1	0	1	1	i

Operation: MOV
((R_i)) ← (A)

MOV @R_i,direct

Bytes: 2

Cycles: 2

Encoding:

1	0	1	0	0	1	1	i	direct addr.

Operation: MOV
((R_i)) ← (direct)

MOV @R_i,#data

Bytes: 2

Cycles: 1

Encoding:

0	1	1	1	0	1	1	i	immediate data

Operation: MOV
((R_i)) ← #data

MOV <dest-bit>,<src-bit>

Function: Move bit data

Description: MOV <dest-bit>,<src-bit> copies the Boolean variable indicated by the second operand into the location specified by the first operand. One of the operands must be the carry flag; the other may be any directly addressable bit. No other register or flag is affected.

Example: The carry flag is originally set. The data present at input Port 3 is 11000101B. The data previously written to output Port 1 is 35H (00110101B).

```
MOV     P1.3,C
MOV     C,P3.3
MOV     P1.2,C
```

leaves the carry cleared and changes Port 1 to 39H (00111001B).

MOV C,bit

Bytes: 2

Cycles: 1

Encoding:

1	0	1	0	0	0	1	0	bit address	

Operation: MOV
(C) ← (bit)

MOV bit,C

Bytes: 2

Cycles: 2

Encoding:

1	0	0	1	0	0	1	0	bit address	

Operation: MOV
(bit) ← (C)

MOV DPTR,#data16

Function: Load Data Pointer with a 16-bit constant

Description: MOV DPTR,#data16 loads the Data Pointer with the 16-bit constant indicated. The 16-bit constant is loaded into the second and third bytes of the instruction. The second byte (DPH) is the high-order byte, while the third byte (DPL) holds the lower-order byte. No flags are affected.

This is the only instruction which moves 16 bits of data at once.

Example: The instruction,

```
MOV     DPTR, # 1234H
```

loads the value 1234H into the Data Pointer: DPH holds 12H, and DPL holds 34H.

Bytes: 3

Cycles: 2

Encoding:

1	0	0	1	0	0	0	0	immed. data15-8	immed. data7-0

Operation: MOV
(DPTR) ← #data$_{15\text{-}0}$
DPH ← DPL ← #data$_{15\text{-}8}$ ← #data$_{7\text{-}0}$

MOVC A,@A+ <base-reg>

Function: Move Code byte

Description: The MOVC instructions load the Accumulator with a code byte or constant from program memory. The address of the byte fetched is the sum of the original unsigned 8-bit Accumulator contents and the contents of a 16-bit base register, which may be either the Data Pointer or the PC. In the latter case, the PC is incremented to the address of the following instruction before being added with the Accumulator; otherwise the base register is not altered. Sixteen-bit addition is performed so a carry-out from the low-order eight bits may propagate through higher-order bits. No flags are affected.

Example: A value between 0 and 3 is in the Accumulator. The following instructions will translate the value in the Accumulator to one of four values defined by the DB (define byte) directive.

```
REL_PC:  INC   A
         MOVC  A,@A+PC
         RET
         DB    66H
         DB    77H
         DB    88H
         DB    99H
```

If the subroutine is called with the Accumulator equal to 01H, it returns with 77H in the Accumulator. The INC A before the MOVC instruction is needed to “get around” the RET instruction above the table. If several bytes of code separate the MOVC from the table, the corresponding number is added to the Accumulator instead.

MOVC A,@A+DPTR

Bytes: 1

Cycles: 2

Encoding:

1	0	0	1	0	0	1	1

Operation: MOVC
(A) ← ((A) + (DPTR))

MOVC A,@A+PC

Bytes: 1

Cycles: 2

Encoding:

1	0	0	0	0	0	1	1

Operation: MOVC
(PC) ← (PC) + 1
(A) ← ((A) + (PC))

MOVX <dest-byte>,<src-byte>

Function: Move External

Description: The MOVX instructions transfer data between the Accumulator and a byte of external data memory, which is why "X" is appended to MOV. There are two types of instructions, differing in whether they provide an 8-bit or 16-bit indirect address to the external data RAM.

In the first type, the contents of R0 or R1 in the current register bank provide an 8-bit address multiplexed with data on P0. Eight bits are sufficient for external I/O expansion decoding or for a relatively small RAM array. For somewhat larger arrays, any output port pins can be used to output higher-order address bits. These pins are controlled by an output instruction preceding the MOVX.

In the second type of MOVX instruction, the Data Pointer generates a 16-bit address. P2 outputs the high-order eight address bits (the contents of DPH), while P0 multiplexes the low-order eight bits (DPL) with data. The P2 Special Function Register retains its previous contents, while the P2 output buffers emit the contents of DPH. This form of MOVX is faster and more efficient when accessing very large data arrays (up to 64K bytes), since no additional instructions are needed to set up the output ports.

It is possible to use both MOVX types in some situations. A large RAM array with its high-order address lines driven by P2 can be addressed via the Data Pointer, or with code to output high-order address bits to P2, followed by a MOVX instruction using R0 or R1.

Example: An external 256 byte RAM using multiplexed address/data lines is connected to the 8051 Port 0. Port 3 provides control lines for the external RAM. Ports 1 and 2 are used for normal I/O. Registers 0 and 1 contain 12H and 34H. Location 34H of the external RAM holds the value 56H. The instruction sequence,

```
MOVX    A,@R1
MOVX    @R0,A
```

copies the value 56H into both the Accumulator and external RAM location 12H.

MOVX A,@R_i

Bytes: 1

Cycles: 2

Encoding:

1	1	1	0	0	0	1	i

Operation: MOVX
(A) ← ((R_i))

MOVX A,@DPTR

Bytes: 1

Cycles: 2

Encoding:

1	1	1	0	0	0	0	0

Operation: MOVX
(A) ← ((DPTR))

MOVX @R_i,A

Bytes: 1

Cycles: 2

Encoding:

1	1	1	1	0	0	1	i

Operation: MOVX
$((R_i)) \leftarrow (A)$

MOVX @DPTR,A

Bytes: 1

Cycles: 2

Encoding:

1	1	1	1	0	0	0	0

Operation: MOVX
(DPTR) ← (A)

MUL AB

Function: Multiply

Description: MUL AB multiplies the unsigned 8-bit integers in the Accumulator and register B. The low-order byte of the 16-bit product is left in the Accumulator, and the high-order byte in B. If the product is greater than 255 (0FFH), the overflow flag is set; otherwise it is cleared. The carry flag is always cleared.

Example: Originally the Accumulator holds the value 80 (50H). Register B holds the value 160 (0A0H). The instruction,

MUL AB

will give the product 12,800 (3200H), so B is changed to 32H (00110010B) and the Accumulator is cleared. The overflow flag is set, carry is cleared.

Bytes: 1

Cycles: 4

Encoding:

1	0	1	0	0	1	0	0

Operation: MUL
$(A)_{7-0} \leftarrow (A) \times (B)$
$(B)_{15-8}$

NOP

Function: No Operation

Description: Execution continues at the following instruction. Other than the PC, no registers or flags are affected.

Example: A low-going output pulse on bit 7 of Port 2 must last exactly 5 cycles. A simple SETB/CLR sequence generates a one-cycle pulse, so four additional cycles must be inserted. This may be done (assuming no interrupts are enabled) with the following instruction sequence,

```
CLR     P2.7
NOP
NOP
NOP
NOP
SETB    P2.7
```

Bytes: 1

Cycles: 1

Encoding:

0	0	0	0	0	0	0	0

Operation: NOP
(PC) ← (PC) + 1

ORL <dest-byte> <src-byte>

Function: Logical-OR for byte variables

Description: ORL performs the bitwise logical-OR operation between the indicated variables, storing the results in the destination byte. No flags are affected.

The two operands allow six addressing mode combinations. When the destination is the Accumulator, the source can use register, direct, register-indirect, or immediate addressing; when the destination is a direct address, the source can be the Accumulator or immediate data.

Note: When this instruction is used to modify an output port, the value used as the original port data is read from the output data latch, *not* the input pins.

Example: If the Accumulator holds 0C3H (11000011B) and R0 holds 55H (01010101B) then the following instruction,

```
ORL     A,R0
```

leaves the Accumulator holding the value 0D7H (1101011lB).When the destination is a directly addressed byte, the instruction can set combinations of bits in any RAM location or hardware register. The pattern of bits to be set is determined by a mask byte, which may be either a constant data value in the instruction or a variable computed in the Accumulator at run-time. The instruction,

```
ORL     P1,#00110010B
```

sets bits 5, 4, and 1 of output Port 1.

ORL A,R_n

Bytes: 1

Cycles: 1

Encoding:

0	1	0	0	1	r	r	r

Operation: ORL
(A) ← (A) ∨ (R_n)

ORL A,direct

Bytes: 2

Cycles: 1

Encoding:	0	1	0	0	0	1	0	1	direct address

Operation: ORL
(A) ← (A) ∨ (direct)

ORL A,@R_i

Bytes: 1

Cycles: 1

Encoding:	0	1	0	0	0	1	1	i

Operation: ORL
(A) ← (A) ∨((R$_i$))

ORL A,#data

Bytes: 2

Cycles: 1

Encoding:	0	1	0	0	0	1	0	0	immediate data

Operation: ORL
(A) ← (A) ∨ #data

ORL direct,A

Bytes: 2

Cycles: 1

Encoding:	0	1	0	0	0	0	1	0	direct address

Operation: ORL
(direct) ← (direct) ∨ (A)

ORL direct,#data

Bytes: 3

Cycles: 2

Encoding:	0	1	0	0	0	0	1	1	direct addr.	immediate data

Operation: ORL
(direct) ← (direct) ∨ #data

ORL C,<src-bit>

Function: Logical-OR for bit variables

Description: Set the carry flag if the Boolean value is a logical 1; leave the carry in its current state otherwise. A slash (/) preceding the operand in the assembly language indicates that the logical complement of the addressed bit is used as the source value, but the source bit itself is not affected. No other flags are affected.

Example: Set the carry flag if and only if P1.0 = 1, ACC. 7 = 1, or OV = 0:

```
MOV     C,P1.0      ;LOAD CARRY WITH INPUT PIN P10
ORL     C,ACC.7     ;OR CARRY WITH THE ACC. BIT 7
ORL     C,/OV       ;OR CARRY WITH THE INVERSE OF OV.
```

ORL C,bit

Bytes: 2

Cycles: 2

Encoding:

0	1	1	1	0	0	1	0	bit address

Operation: ORL
(C) ← (C) ∨ (bit)

ORL C,/bit

Bytes: 2

Cycles: 2

Encoding:

1	0	1	0	0	0	0	0	bit address

Operation: ORL
(C) ← (C) ∨ ($\overline{\text{bit}}$)

POP direct

Function: Pop from stack.

Description: The contents of the internal RAM location addressed by the Stack Pointer is read, and the Stack Pointer is decremented by one. The value read is then transferred to the directly addressed byte indicated. No flags are affected.

Example: The Stack Pointer originally contains the value 32H, and internal RAM locations 30H through 32H contain the values 20H, 23H, and 01H, respectively. The following instruction sequence,

```
POP     DPH
POP     DPL
```

leaves the Stack Pointer equal to the value 30H and sets the Data Pointer to 0123H. At this point, the following instruction,

```
POP     SP
```

leaves the Stack Pointer set to 20H. In this special case, the Stack Pointer was decremented to 2FH before being loaded with the value popped (20H).

Bytes: 2

Cycles: 2

Encoding:

1	1	0	1	0	0	0	0	direct address

Operation: POP
(direct) ← ((SP))
(SP) ← (SP) - 1

PUSH direct

Function: Push onto stack

Description: The Stack Pointer is incremented by one. The contents of the indicated variable is then copied into the internal RAM location addressed by the Stack Pointer. Otherwise no flags are affected.

Example: On entering an interrupt routine, the Stack Pointer contains 09H. The Data Pointer holds the value 0123H. The following instruction sequence,

```
PUSH    DPL
PUSH    DPH
```

leaves the Stack Pointer set to 0BH and stores 23H and 01H in internal RAM locations 0AH and 0BH, respectively.

Bytes: 2

Cycles: 2

Encoding:

1	1	0	0	0	0	0	0	direct address

Operation: PUSH
(SP) ← (SP) + 1
((SP)) ← (direct)

RET

Function: Return from subroutine

Description: RET pops the high- and low-order bytes of the PC successively from the stack, decrementing the Stack Pointer by two. Program execution continues at the resulting address, generally the instruction immediately following an ACALL or LCALL. No flags are affected.

Example: The Stack Pointer originally contains the value 0BH. Internal RAM locations 0AH and 0BH contain the values 23H and 01H, respectively. The following instruction,

```
RET
```

leaves the Stack Pointer equal to the value 09H. Program execution continues at location 0123H.

Bytes: 1

Cycles: 2

Encoding:

0	0	1	0	0	0	1	0

Operation: RET
(PC_{15-8}) ← ((SP))
(SP) ← (SP) - 1
(PC_{7-0}) ← ((SP))
(SP) ← (SP) - 1

RETI

Function: Return from interrupt

Description: RETI pops the high- and low-order bytes of the PC successively from the stack and restores the interrupt logic to accept additional interrupts at the same priority level as the one just processed. The Stack Pointer is left decremented by two. No other registers are affected; the PSW is *not* automatically restored to its pre-interrupt status. Program execution continues at the resulting address, which is generally the instruction immediately after the point at which the interrupt request was detected. If a lower- or same-level interrupt was pending when the RETI instruction is executed, that one instruction is executed before the pending interrupt is processed.

Example: The Stack Pointer originally contains the value 0BH. An interrupt was detected during the instruction ending at location 0122H. Internal RAM locations 0AH and 0BH contain the values 23H and 01H, respectively. The following instruction,

RETI

leaves the Stack Pointer equal to 09H and returns program execution to location 0123H.

Bytes: 1

Cycles: 2

Encoding:

0	0	1	1	0	0	1	0

Operation: RETI
$(PC_{15-8}) \leftarrow ((SP))$
$(SP) \leftarrow (SP) - 1$
$(PC_{7-0}) \leftarrow ((SP))$
$(SP) \leftarrow (SP) - 1$

RL A

Function: Rotate Accumulator Left

Description: The eight bits in the Accumulator are rotated one bit to the left. Bit 7 is rotated into the bit 0 position. No flags are affected.

Example: The Accumulator holds the value 0C5H (11000101B). The following instruction,

RL A

leaves the Accumulator holding the value 8BH (10001011B) with the carry unaffected.

Bytes: 1

Cycles: 1

Encoding:

0	0	1	0	0	0	1	1

Operation: RL
$(A_n + 1) \leftarrow (A_n)$ n = 0 - 6
$(A_0) \leftarrow (A_7)$

RLC A

Function: Rotate Accumulator Left through the Carry flag

Description: The eight bits in the Accumulator and the carry flag are together rotated one bit to the left. Bit 7 moves into the carry flag; the original state of the carry flag moves into the bit 0 position. No other flags are affected.

Example: The Accumulator holds the value 0C5H(11000101B), and the carry is zero. The following instruction,

```
RLC     A
```

leaves the Accumulator holding the value 8BH (10001010B) with the carry set.

Bytes: 1

Cycles: 1

Encoding:

0	0	1	1	0	0	1	1

Operation: RLC
$(A_n + 1) \leftarrow (A_n)$ n = 0 - 6
$(A_0) \leftarrow (C)$
$(C) \leftarrow (A_7)$

RR A

Function: Rotate Accumulator Right

Description: The eight bits in the Accumulator are rotated one bit to the right. Bit 0 is rotated into the bit 7 position. No flags are affected.

Example: The Accumulator holds the value 0C5H (11000101B). The following instruction,

```
RR      A
```

leaves the Accumulator holding the value 0E2H (11100010B) with the carry unaffected.

Bytes: 1

Cycles: 1

Encoding:

0	0	0	0	0	0	1	1

Operation: RR
$(A_n) \leftarrow (A_n + 1)$ n = 0 - 6
$(A_7) \leftarrow (A_0)$

RRC A

Function: Rotate Accumulator Right through Carry flag

Description: The eight bits in the Accumulator and the carry flag are together rotated one bit to the right. Bit 0 moves into the carry flag; the original value of the carry flag moves into the bit 7 position. No other flags are affected.

Example: The Accumulator holds the value 0C5H (11000101B), the carry is zero. The following instruction,

```
RRC     A
```

leaves the Accumulator holding the value 62 (01100010B) with the carry set.

Bytes: 1

Cycles: 1

Encoding:

0	0	0	1	0	0	1	1

Operation: RRC
$(A_n) \leftarrow (A_n + 1)$ n = 0 - 6
$(A_7) \leftarrow (C)$
$(C) \leftarrow (A_0)$

SETB <bit>

Function: Set Bit

Description: SETB sets the indicated bit to one. SETB can operate on the carry flag or any directly addressable bit. No other flags are affected.

Example: The carry flag is cleared. Output Port 1 has been written with the value 34H (00110100B). The following instructions,

```
SETB    C
SETB    P1.0
```

sets the carry flag to 1 and changes the data output on Port 1 to 35H (00110101B).

SETB C

Bytes: 1

Cycles: 1

Encoding:

1	1	0	1	0	0	1	1

Operation: SETB
(C) ← 1

SETB bit

Bytes: 2

Cycles: 1

Encoding:

1	1	0	1	0	0	1	0	bit address

Operation: SETB
(bit) ← 1

SJMP rel

Function: Short Jump

Description: Program control branches unconditionally to the address indicated. The branch destination is computed by adding the signed displacement in the second instruction byte to the PC, after incrementing the PC twice. Therefore, the range of destinations allowed is from 128 bytes preceding this instruction 127 bytes following it.

Example: The label RELADR is assigned to an instruction at program memory location 0123H. The following instruction,

```
SJMP    RELADR
```

assembles into location 0100H. After the instruction is executed, the PC contains the value 0123H.

Note: Under the above conditions the instruction following SJMP is at 102H. Therefore, the displacement byte of the instruction is the relative offset (0123H-0102H) = 21H. Put another way, an SJMP with a displacement of 0FEH is a one-instruction infinite loop.

Bytes: 2

Cycles: 2

Encoding:

1	0	0	0	0	0	0	0	rel. address

Operation: SJMP
(PC) ← (PC) + 2
(PC) ← (PC) + rel

SUBB A,<src-byte>

Function: Subtract with borrow

Description: SUBB subtracts the indicated variable and the carry flag together from the Accumulator, leaving the result in the Accumulator. SUBB sets the carry (borrow) flag if a borrow is needed for bit 7 and clears C otherwise. (If C was set *before* executing a SUBB instruction, this indicates that a borrow was needed for the previous step in a multiple-precision subtraction, so the carry is subtracted from the Accumulator along with the source operand.) AC is set if a borrow is needed for bit 3 and cleared otherwise. OV is set if a borrow is needed into bit 6, but not into bit 7, or into bit 7, but not bit 6.

When subtracting signed integers, OV indicates a negative number produced when a negative value is subtracted from a positive value, or a positive result when a positive number is subtracted from a negative number.

The source operand allows four addressing modes: register, direct, register-indirect, or immediate.

Example: The Accumulator holds 0C9H (11001001B), register 2 holds 54H (01010100B), and the carry flag is set. The instruction,

```
SUBB    A,R2
```

will leave the value 74H (01110100B) in the accumulator, with the carry flag and AC cleared but OV set.

Notice that 0C9H minus 54H is 75H. The difference between this and the above result is due to the carry (borrow) flag being set before the operation. If the state of the carry is not known before starting a single or multiple-precision subtraction, it should be explicitly cleared by CLR C instruction.

SUBB A,R_n

Bytes: 1

Cycles: 1

Encoding:

1	0	0	1	1	r	r	r

Operation: SUBB
(A) ← (A) - (C) - (R_n)

SUBB A,direct

Bytes: 2

Cycles: 1

Encoding:

1	0	0	1	0	1	0	1	direct address

Operation: SUBB
(A) ← (A) - (C) - (direct)

SUBB A,@R_i

Bytes: 1

Cycles: 1

Encoding:

1	0	0	1	0	1	1	i

Operation: SUBB
(A) ← (A) - (C) - ((R_i))

SUBB A,#data

Bytes: 2

Cycles: 1

Encoding:

1	0	0	1	0	1	0	0	immediate data

Operation: SUBB
(A) ← (A) - (C) - #data

SWAP A

Function: Swap nibbles within the Accumulator

Description: SWAP A interchanges the low- and high-order nibbles (four-bit fields) of the Accumulator (bits 3 through 0 and bits 7 through 4). The operation can also be thought of as a 4-bit rotate instruction. No flags are affected.

Example: The Accumulator holds the value 0C5H (11000101B). The instruction,

```
SWAP    A
```

leaves the Accumulator holding the value 5CH (01011100B).

Bytes: 1

Cycles: 1

Encoding:

1	1	0	0	0	1	0	0

Operation: SWAP
(A_{3-0}) D (A_{7-4})

XCH A,<byte>

Function: Exchange Accumulator with byte variable

Description: XCH loads the Accumulator with the contents of the indicated variable, at the same time writing the original Accumulator contents to the indicated variable. The source/destination operand can use register, direct, or register-indirect addressing.

Example: R0 contains the address 20H. The Accumulator holds the value 3FH (0011111lB). Internal RAM location 20H holds the value 75H (01110101B). The following instruction,

```
XCH     A,@R0
```

leaves RAM location 20H holding the values 3FH (00111111B) and 75H (01110101B) in the accumulator.

XCH A,R_n

Bytes: 1

Cycles: 1

Encoding:

1	1	0	0	1	r	r	r

Operation: XCH
(A) D $((R_n)$

XCH A,direct

Bytes: 2

Cycles: 1

Encoding:

1	1	0	0	0	1	0	1	direct address

Operation: XCH
(A) D (direct)

XCH A,@R_i

Bytes: 1

Cycles: 1

Encoding:

1	1	0	0	0	1	1	i

Operation: XCH
(A) D $((R_i))$

XCHD A,@R_i

Function: Exchange Digit

Description: XCHD exchanges the low-order nibble of the Accumulator (bits 3 through 0), generally representing a hexadecimal or BCD digit, with that of the internal RAM location indirectly addressed by the specified register. The high-order nibbles (bits 7-4) of each register are not affected. No flags are affected.

Example: R0 contains the address 20H. The Accumulator holds the value 36H (00110110B). Internal RAM location 20H holds the value 75H (01110101B). The following instruction,

```
XCHD    A,@R0
```

leaves RAM location 20H holding the value 76H (01110110B) and 35H (00110101B) in the Accumulator.

Bytes: 1

Cycles: 1

Encoding:

1	1	0	1	0	1	1	i

Operation: XCHD
(A_{3-0}) D $((R_{i3-0}))$

XRL <dest-byte>,<src-byte>

Function: Logical Exclusive-OR for byte variables

Description: XRL performs the bitwise logical Exclusive-OR operation between the indicated variables, storing the results in the destination. No flags are affected.

The two operands allow six addressing mode combinations. When the destination is the Accumulator, the source can use register, direct, register-indirect, or immediate addressing; when the destination is a direct address, the source can be the Accumulator or immediate data.

Note: When this instruction is used to modify an output port, the value used as the original port data is read from the output data latch, *not* the input pins.

Example: If the Accumulator holds 0C3H (1100001lB) and register 0 holds 0AAH (10101010B) then the instruction,

```
XRL     A,R0
```

leaves the Accumulator holding the value 69H (01101001B).

When the destination is a directly addressed byte, this instruction can complement combinations of bits in any RAM location or hardware register. The pattern of bits to be complemented is then determined by a mask byte, either a constant contained in the instruction or a variable computed in the Accumulator at run-time. The following instruction,

```
XRL     P1,#00110001B
```

complements bits 5, 4, and 0 of output Port 1.

XRL A,R_n

Bytes: 1

Cycles: 1

Encoding:

0	1	1	0	1	r	r	r

Operation: XRL
(A) ← (A) ∀ (R_n)

XRL A,direct

Bytes: 2

Cycles: 1

Encoding:

0	1	1	0	0	1	0	1	direct address

Operation: XRL
(A) ← (A) ∀ (direct)

XRL A,@R_i

Bytes: 1

Cycles: 1

Encoding:

0	1	1	0	0	1	1	i

Operation: XRL
(A) ← (A) ∀ ((R_i))

XRL A,#data

Bytes: 2

Cycles: 1

Encoding:

0	1	1	0	0	1	0	0	immediate data

Operation: XRL
(A) ← (A) ∀ #data

XRL direct,A

Bytes: 2

Cycles: 1

Encoding:

0	1	1	0	0	0	1	0	direct address

Operation: XRL
(direct) ← (direct) ∀ (A)

XRL direct,#data

Bytes: 3

Cycles: 2

Encoding:

0	1	1	0	0	0	1	1	direct address	immediate data

Operation: XRL
(direct) ← (direct) ∀ #data

LCD MOUDULE SPECIFICATION FOR APPROVAL	DATE	10/12/03
	VER.	1.0
JHD12864E	PAGE	1

CONTENTS

1. FEATURES

- •Display construction…… 128*64 DOTS
- •Display mode …… STN / Yellow Green
- •Display type …… Positive Tranflective
- •Viewing direction…… 6 o' clock
- •Operating temperature…… Indoor
- •Driving voltage…… Single power
- •Driving method……1/64 duty, 1/9 bias
- •Type……COB (Chip On Board)
- •Number of data line…… 8-bit parallel
- •Connector…… Pin

2. MECHANICAL DATA

ITEM		WIDTH	HEIGHT	THICKNESS	UNIT
Module size		93.0	70.0	12.7(MAX)	mm
Viewing area		70.7	38.8	-	mm
Dot	Size	0.48	0.48	-	mm
	Pitch	0.52	0.52	-	mm
Diameter of mounting hole		2.7			mm
Weight		About 50			g

3. ABSOLUTE MAXIMUM RATINGS

Characteristic	Symbol	Value	Unit	Note
Operating voltage	V_{DD}	-0.3 to +7.0	V	(1)
Supply voltage	V_{EE}	V_{DD}-19.0 to V_{DD}+0.3	V	(4)
Driver supply voltage	V_B	-0.3 to V_{DD}+0.3	V	(1), (3)
	V_{LCD}	V_{EE}-0.3 to V_{DD}+0.3	V	(2)
Operating temperature	T_{OPR}	-30 to +85	°C	
Storage temperature	T_{STG}	-55 to +125	°C	

4. ELECTRICAL CHARACTERISTICS

(V_{DD} = +5V ± 10%, V_{SS} = 0V, V_{DD}-V_{EE} = 8 to 17V, Ta =-30 to +85°C)

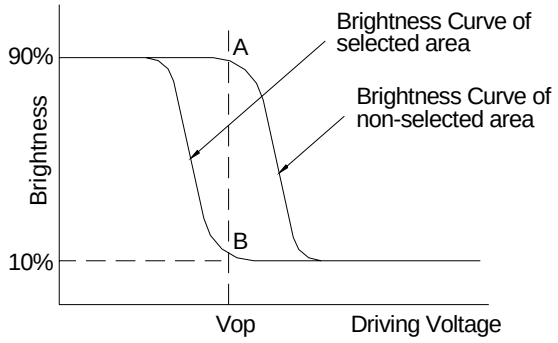
Characteristic	Symbol	Condition	Min	Typ	Max	Unit	Note
Input high voltage	V_{IH1}	–	0.7V_{DD}	–	V_{DD}	V	(1)
	V_{IH2}	–	2.0	–	V_{DD}	V	(2)
Input low voltage	V_{IL1}	–	0	–	0.3V_{DD}	V	(1)
	V_{IL2}	–	0	–	0.8	V	(2)
Output high voltage	V_{OH}	I_{OH} = -200μA	2.4	–	–	V	(3)
Output low voltage	V_{OL}	I_{OL} = 1.6mA	–	–	0.4	V	(3)
Input leakage current	I_{LKG}	V_{IN} = V_{SS} - V_{DD}	-1.0	–	1.0	μA	(4)
Three-state(off) input current	I_{TSL}	V_{IN} = V_{SS} - V_{DD}	-5.0	–	5.0	μA	(5)
Driver input leakage current	I_{DIL}	V_{IN} = V_{EE} - V_{DD}	-2.0	–	2.0	μA	(6)
Operating current	I_{DD1}	During display	–	–	100	μA	(7)
	I_{DD2}	During access Access cycle = 1MHz	–	–	500	μA	(7)
On resistance	R_{ON}	V_{DD}-V_{EE} = 15V I_{LOAD} = ± 0.1mA	–	–	7.5	KΩ	(8)

5. ELECTRO-OPTICAL CHARACTERISTICS

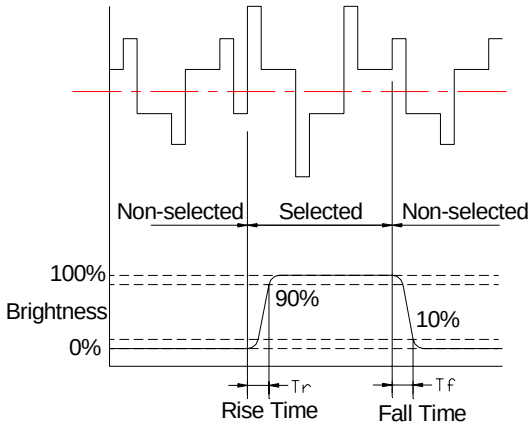
ITEM	SYMBOL	CONDITION	MIN.	TYP.	MAX.	UNIT	NOTE
Contrast ratio	K	Φ=0	1.4	4	-	-	1
Response time (rise)	Tr	Φ=1	-	130	-	ms	2
Response time (fall)	Tf	Φ=2		130	-	ms	2
Viewing angle	Φ	K ≥1.4	10 -- +30			deg.	3
	θ		-30 -- +30				

Note 1: Definition of Contrast Ratio "K"

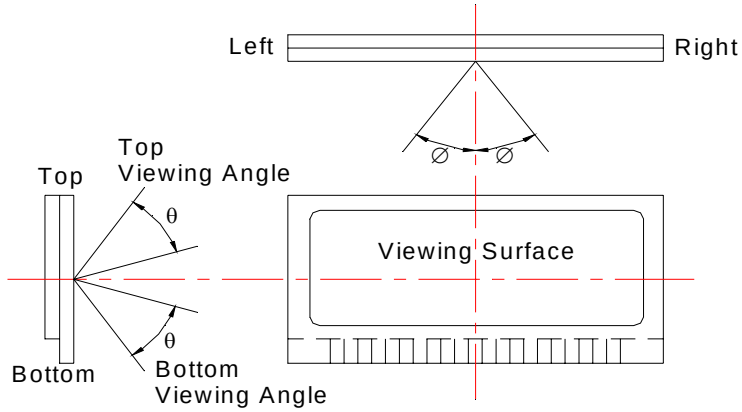
$$K = \frac{\text{Brightness of non-selected segment(A)}}{\text{Brightness of selected segment(B)}}$$



Note 2: Definition of Optical Response Time

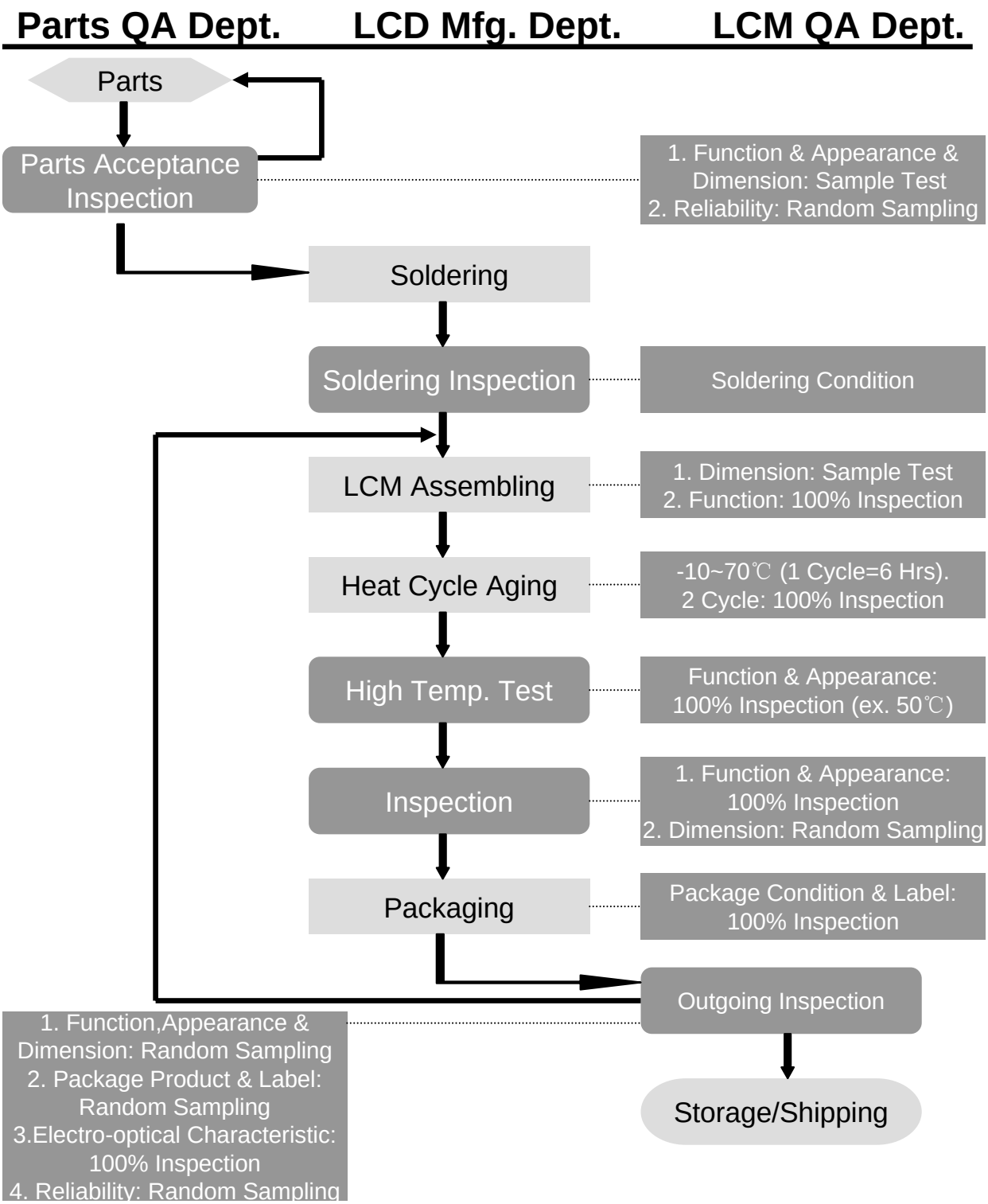


Note 3: Definition of Viewing Angle



Please select either top or bottom viewing angle

6. QC/QA PROCEDURE



7. RELIABILITY

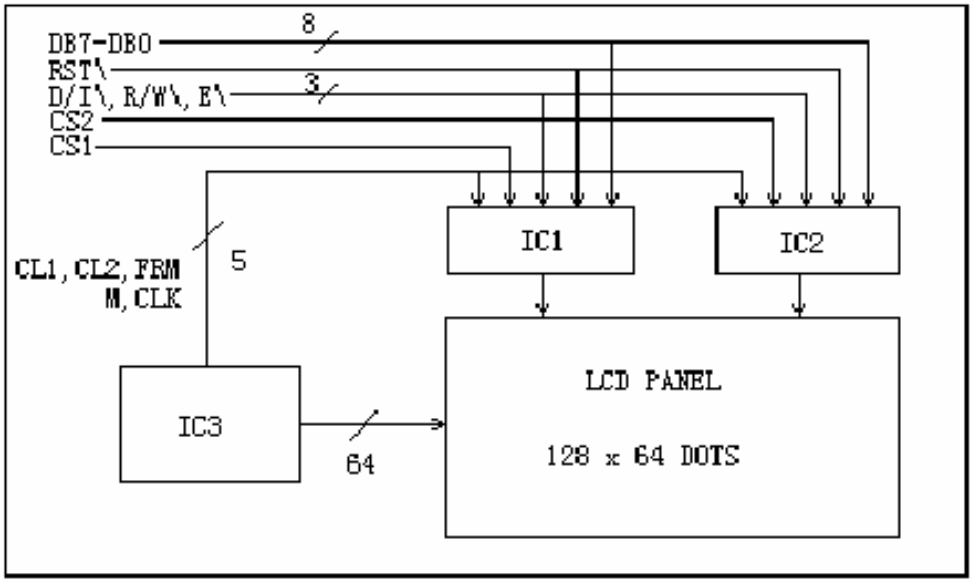
•Operating life time:

Longer than 50000 hours (at room temperature without direct irradiation of sunlight)

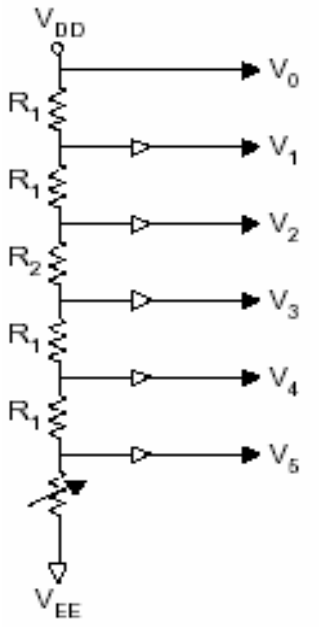
•Reliability Characteristics:

Item	Test	Criterion
High temp	70℃ / 200 Hrs	■Total current consumption should be below double of initial value ■Contrast ratio should be within initial value±50% ■No defect in cosmetic and operational function is allowable
Low temp.	-20℃ / 200 Hrs	
High humidity	40℃ * 90%RH / 200 Hrs	
Thermal shock	-20℃→25℃→70℃→25℃ /5 Cycles (30min) (5min) (30min) (5min)	
Vibration	1.Operating time: Thirty minutes exposure in each direction (x, y, z) 2.Sweep Frequency (1min):10Hz→ 55Hz →10Hz 3.Amplitude: 0.75mm double amplitude	

8. BLOCK DIAGRAM

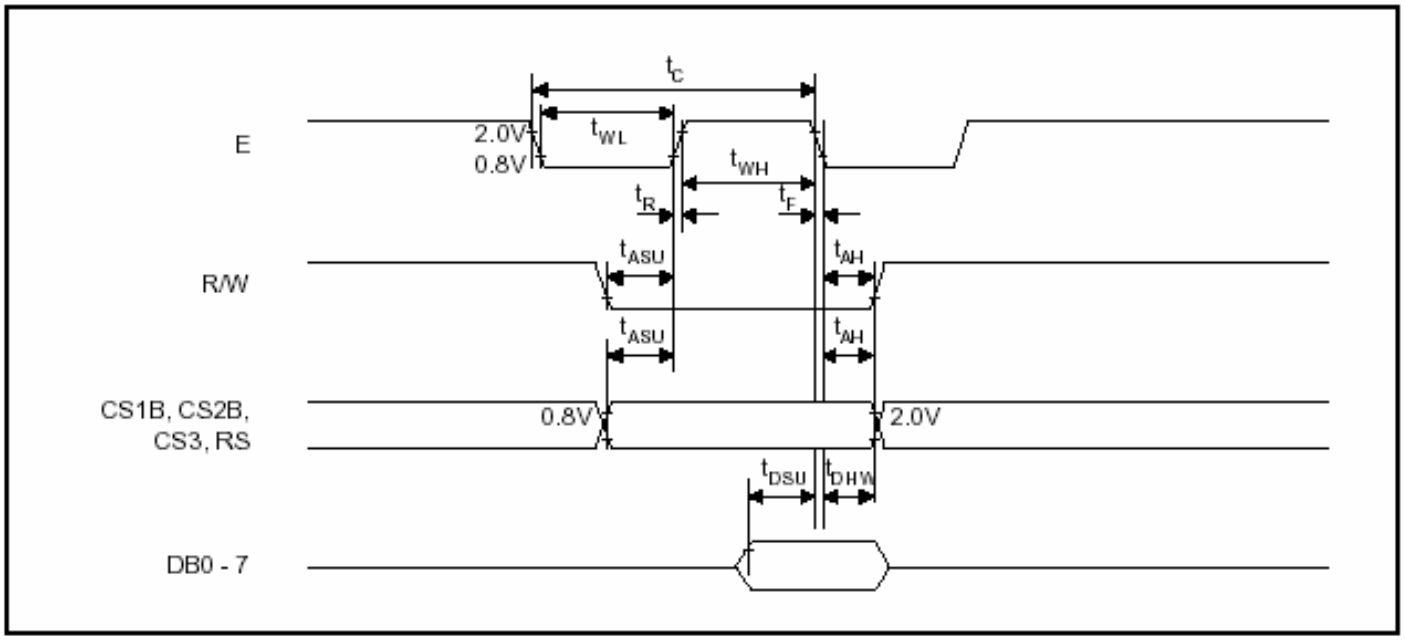


9. POWER SUPPLY

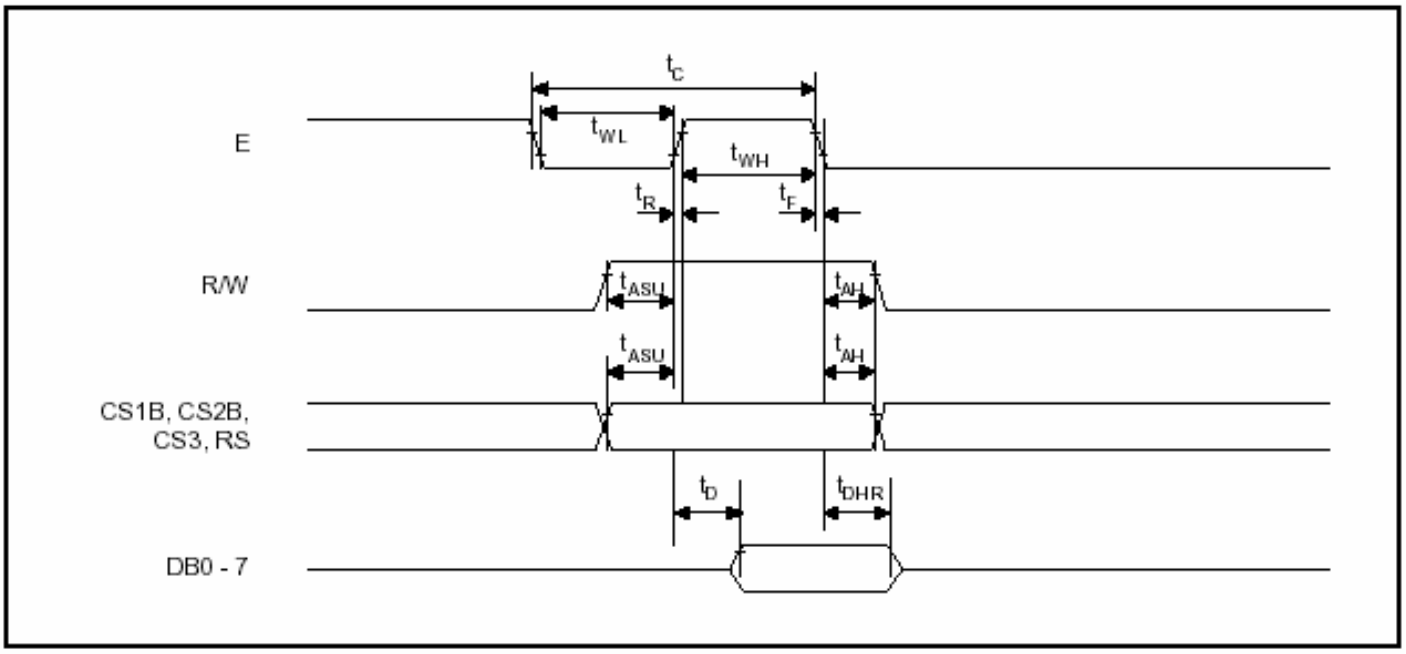


10. TIMING DIAGRAM

- WRITE OPERATION



- READ OPERATION



11. AC CHARACTERISTICS

MPU Interface

Characteristic	Symbol	Min	Typ	Max	Unit
E cycle	t_C	1000	–	–	ns
E high level width	t_{WH}	450	–	–	ns
E low level width	t_{WL}	450	–	–	ns
E rise time	t_R	–	–	25	ns
E fall time	t_F	–	–	25	ns
Address set-up time	t_{ASU}	140	–	–	ns
Address hold time	t_{AH}	10	–	–	ns
Data set-up time	t_{DSU}	200	–	–	ns
Data delay time	t_D	–	–	320	ns
Data hold time (write)	t_{DHW}	10	–	–	ns
Data hold time (read)	t_{DHR}	20	–	–	ns

12. INSTRUCTION SET

DISPLAY CONTROL INSTRUCTION

The display control instructions control the internal state of the S6B0108. Instruction is received from MPU to S6B0108 for the display control. The following table shows various instructions.

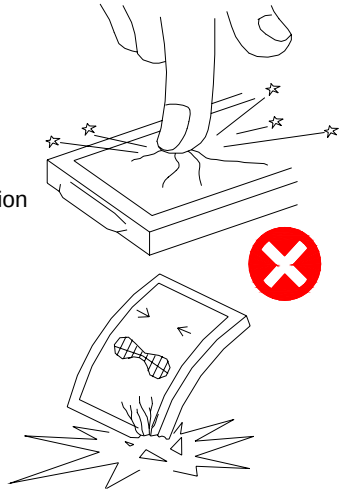
Instruction	RS	R/W	DB7	DB6	DB5	DB4	DB3	DB2	DB1	DB0	Function
Display on/off	L	L	L	L	H	H	H	H	H	L/H	Controls the display on or off. Internal status and display RAM data is not affected. L: OFF, H: ON
Set address (Y address)	L	L	L	H	Y address (0 - 63)						Sets the Y address in the Y address counter.
Set page (X address)	L	L	H	L	H	H	H	Page (0 - 7)			Sets the X address at the X address register.
Display start line (Z address)	L	L	H	H	Display start line (0 - 63)						Indicates the display data RAM displayed at the top of the screen.
Status read	L	H	Busy	L	On/ Off	Reset	L	L	L	L	Read status. BUSY L: Ready H: In operation ON/OFF L: Display ON H: Display OFF RESET L: Normal H: Reset
Write display data	H	L	Write data								Writes data (DB0:7) into display data RAM. After writing instruction, Y address is increased by 1 automatically.
Read display data	H	H	Read data								Reads data (DB0:7) from display data RAM to the data bus.

13. Handling Precautions

1. Limitation of Application:
Optrex products are designed for use in ordinary electronic devices such as business machines, telecommunications equipment,measurement devices and etc. Please handle the products with care. (see below)
Optrex products are not designed,intended ,or authorized for use in any application which the failure of the product could result in a situation where personal injury or death may occur . these applications include, but are not limited to . life-sustaining equipment,nuclear control devices , aerospace equipment , devices related to hazardous or flammable materials , etc.[If Buyer intends to purchase or use the Optrex Products for such unintended or unauthorized applications , Buyer must secure prior written consent to such use by a responsible officer of Optrex Corporation.]Should Buyer purchase or use Optrex Products for any such unintended or unauthorized application [without such consent].Buyer shall indemnify and hold Optrex and its officers. employees. subsidiaries, affiliates and distributors harmless against all claims, costs, damages and expenses , and reasonable attorney's fees,arising out of , directly or indirectly, any claim of personal injury or death associated with such unintended or unauthorized use, even if such claim alleges that Optrex was negligent regarding the design or manufacture of the part.
2.Industrial Rights and Patents
Optrex shall not be responsible for any infringement of industrial property rights of third parties in any country arising out of the application or use of Optrex products, except which directly concern the structure or production of such products.

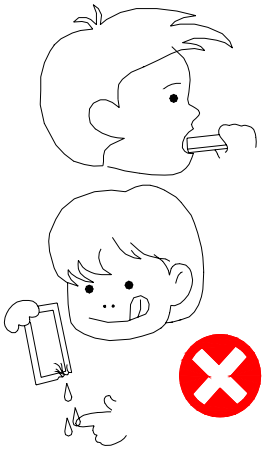
No Press and Shock!

If pressure to LCD, orientation may be disturbed.
LCD will broken by shock!



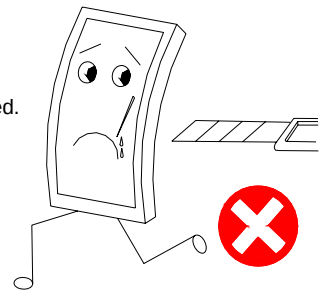
Don't Swallow or Touch Liquid Crystal!

Liquid Crystal may be leaked when display is broked.
If it accidentally gets your hands, wash then with water!



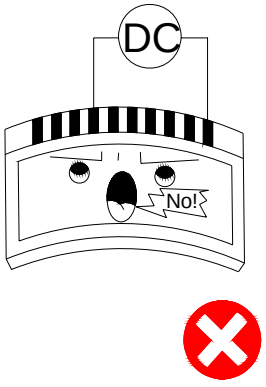
Don't not Scratch!

Polarizer is a soft material and can easily be scratched.



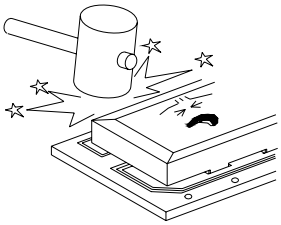
No DC Voltage to LCD!

DC volrage or driveing higher than the specified voltage will reduce the lifetime of the LCD.

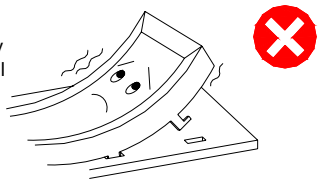


Don't Press the Metallic Frame and Disassemble the LCM

Pressure on the metallic frame and PCB may deform the conductive rubber or break the liquid crystal cell and back light, which will cause defects.

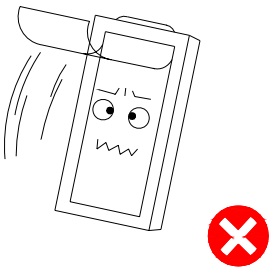


LCD may be shifted or conductive rubber may be reshaped, which will cause defects.



Slowly Peel Off Protective Film!

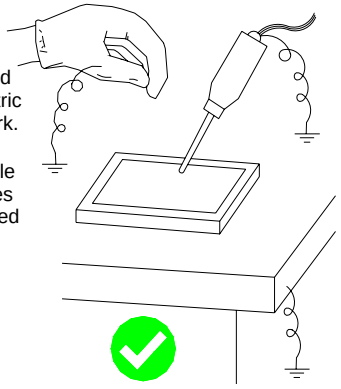
Avoid static electricity.



Avoid Static Electricity!

Please be sure to ground human body and electric apploances during work.

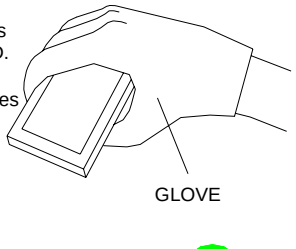
It is preferable to use conductive mat on table and wear cotton clothes or conduction processed fiber. Synthetic fiber is not recommended.



Wear Gloves While Handing!

It is preferable to wear gloves to avoid damaging the LCD.

Please do not touch electrodes with bare hands or make them dirty.

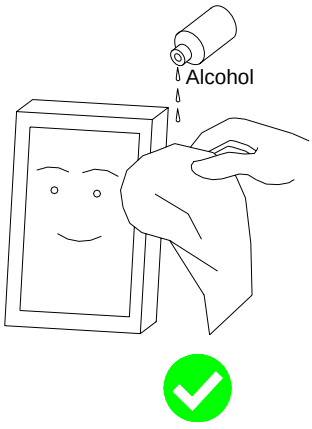


Keep Away From Extreme Heat and Humidity!

LCD deteriorates.

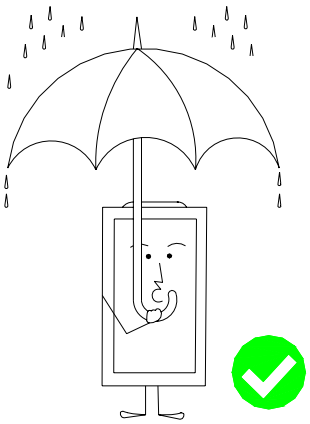
Use Alcohol to Clean Terminals!

When attaching with the heat seal or anisontropically conductive film, wipe off with alcohol before use.



Don't Drop Water on LCD!

Note that the presence of waterdrops or dew in the LCD panel may deteriorate the polarizer or corrade electrode.



Precaution in Soldering LCD Module

Basic instructions: Solder I/O terminals only.
Use soldering iron without leakage.
(1)Soldering condition to I/O terminals
Temperature at tip of the iron: 280±10℃
Soldering time: 3~4 sec.
Type of solder: Eutectic solder (containing colophony-flux)
*Please do not use flux because it may soak into LCD Module or contaminate it.
*It is preferable to peel off protective film on display surface after soldering I/O terminals is finished.
(2)Remove connector or cable
*When you remove connector or cable soldered to I/O terminals, please confirm that solder is fully melted. If you remove by force, electrodes at I/O terminals may be damaged(or stripped off).
*It is recommended to use solder suction machine.

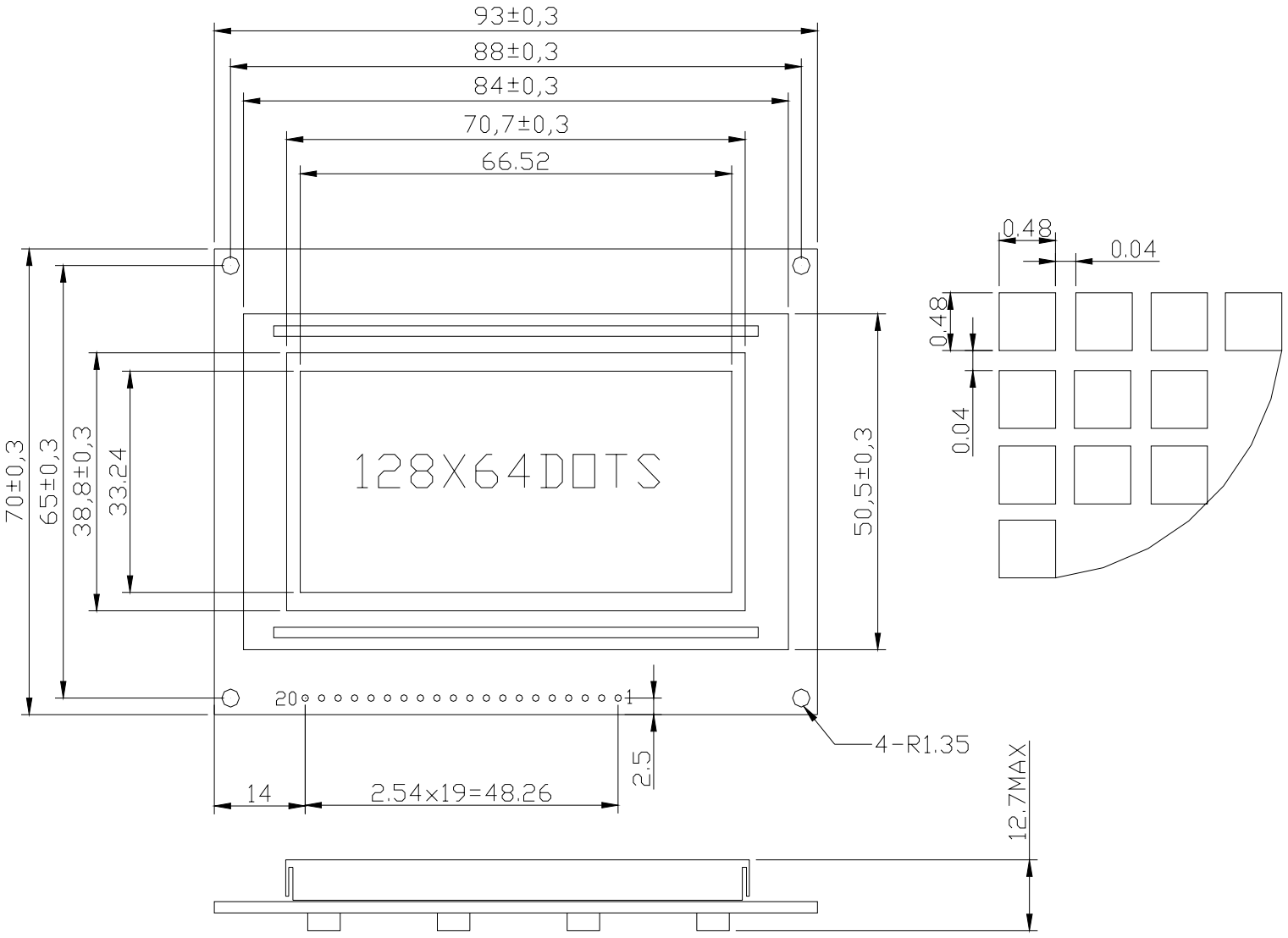
Long-term Storage

If it is necessary to store LCD modules for a long time, please comply with the following procedures.
If storage condition is not satisfactory, display(especially polarizer) may be deteriorated or soldering I/O terminals may become difficult(some oxide is generated at I/O terminals plating).
1.Store as delivered by Optrex
2.If you store as unpacked,put in anti-static bag,seal its opening and store where it is not subjected to direct sunshine nor fluorescent lamp.
3.Store at temperature 0 to +35℃ and at low humidity.Please refer to our specification sheets for storage temperature range and humidity condition.

Long-term Storage

Please use power supply with built-in surge protection circuit.

14. EXTERNAL DIMENSION



1	2	3	4	5	6	7	8	9	10
VSS	VDD	V0	D/I	R/W	E	DB0	DB1	DB2	DB3
11	12	13	14	15	16	17	18	19	20
DB4	DB5	DB6	DB7	CS1	CS2	RST	VEE	LED+	LED-

15. INTERFACE

PIN NO.	SYMBOL	DESCRIPTION	FUNCTION
1	VSS	GROUND	0V (GND)
2	VDD	POWER SUPPLY FOR LOGIC CIRCUIT	+5V
3	V0	LCD CONTRAST ADJUSTMENT	
4	RS	INSTRUCTION/DATA REGISTER SELECTION	RS = 0 : INSTRUCTION REGISTER RS = 1 : DATA REGISTER
5	R/W	READ/WRITE SELECTION	R/W = 0 : REGISTER WRITE R/W = 1 : REGISTER READ
6	E	ENABLE SIGNAL	
7	DB0	DATA INPUT/OUTPUT LINES	
8	DB1		
9	DB2		
10	DB3		8 BIT: DB0-DB7
11	DB4		
12	DB5		
13	DB6		
14	DB7		
15	CS1	CHIP SELECTION	CS1=1,CHIP SELECT SIGNAL FOR IC1
16	CS2	CHIP SELECTION	CS2=1,CHIP SELECT SIGNAL FOR IC2
17	RST	RESET SIGNAL	RSTB=0,DISPLAY OFF,DISPLAY FROM LINE 0.
18	VEE	NEGATIVE VOLTAGE FOR LCD DRIVING	-10V
19	LED+	SUPPLY VOLTAGE FOR LED+	+5V
20	LED-	SUPPLY VOLTAGE FOR LED-	0V

TEXAS INSTRUMENTS
Data sheet acquired from Harris Semiconductor
SCHS053C – Revised September 2003

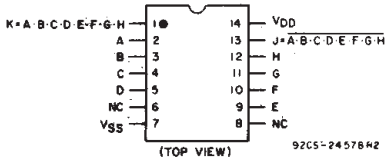
CD4068B Types

CMOS 8-Input NAND/AND Gate

High-Voltage Types (20-Volt Rating)

■ CD4068B NAND/AND gate provides the system designer with direct implementation of the positive-logic 8-input NAND and AND functions and supplements the existing family of CMOS gates.

The CD4068B types are supplied in 14-lead hermetic dual-in-line ceramic packages (F3A suffix), 14-lead dual-in-line plastic packages (E suffix), 14-lead small-outline packages (M, MT, M96, and NSR suffixes), and 14-lead thin shrink small-outline packages (PW and PWR suffixes).

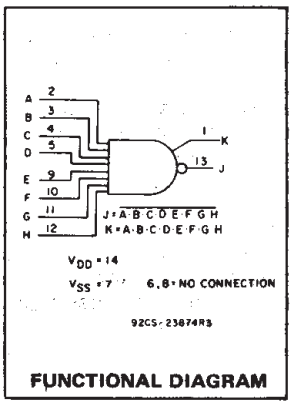


NC = NO CONNECTION

TERMINAL ASSIGNMENT

Features:

- Medium-Speed Operation: t_{PHL}, t_{PLH} = 75 ns (typ.) at V_{DD} = 10 V
- Buffered inputs and outputs
- 5-V, 10-V, and 15-V parametric ratings
- Standardized symmetrical output characteristics
- 100% tested for quiescent current at 20 V
- Maximum input current of 1 μA at 18 V over full package-temperature range; 100 nA at 18 V and 25°C
- Noise margin (over full package-temperature range): 1 V at V_{DD} = 5 V; 2 V at V_{DD} = 10 V; 2.5 V at V_{DD} = 15 V
- Meets all requirements of JEDEC Tentative **Standard No. 13B, "Standard Specifications for Description of 'B' Series CMOS Devices"**



FUNCTIONAL DIAGRAM

RECOMMENDED OPERATING CONDITIONS

For maximum reliability, nominal operating conditions should be selected so that operation is always within the following ranges:

CHARACTERISTIC	Min.	Max.	Units
Supply-Voltage Range (For T_A = Full Package Temperature Range)	3	18	V

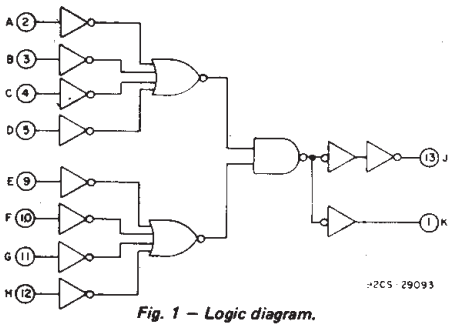


Fig. 1 – Logic diagram.

STATIC ELECTRICAL CHARACTERISTICS

CHARACTERISTIC	CONDITIONS			LIMITS AT INDICATED TEMPERATURES (°C)							UNITS
								+25			
	V_O (V)	V_{IN} (V)	V_{DD} (V)	−55	−40	+85	+125	Min.	Typ.	Max.	
Quiescent Device Current, I_{DD} Max.	–	0,5	5	0.25	0.25	7.5	7.5	–	0.01	0.25	μA
	–	0,10	10	0.5	0.5	15	15	–	0.01	0.5	
	–	0,15	15	1	1	30	30	–	0.01	1	
	–	0,20	20	5	5	150	150	–	0.02	5	
Output Low (Sink) Current I_{OL} Min.	0.4	0,5	5	0.64	0.61	0.42	0.36	0.51	1	–	mA
	0.5	0,10	10	1.6	1.5	1.1	0.9	1.3	2.6	–	
	1.5	0,15	15	4.2	4	2.8	2.4	3.4	6.8	–	
Output High (Source) Current, I_{OH} Min.	4.6	0,5	5	−0.64	−0.61	−0.42	−0.36	−0.51	−1	–	
	2.5	0,5	5	−2	−1.8	−1.3	−1.15	−1.6	−3.2	–	
	9.5	0,10	10	−1.6	−1.5	−1.1	−0.9	−1.3	−2.6	–	
	13.5	0,15	15	−4.2	−4	−2.8	−2.4	−3.4	−6.8	–	
Output Voltage: Low-Level, V_{OL} Max.	–	0,5	5	0.05				–	0	0.05	V
	–	0,10	10	0.05				–	0	0.05	
	–	0,15	15	0.05				–	0	0.05	
Output Voltage: High-Level, V_{OH} Min.	–	0,5	5	4.95				4.95	5	–	
	–	0,10	10	9.95				9.95	10	–	
	–	0,15	15	14.95				14.95	15	–	
Input Low Voltage, V_{IL} Max.	0.5,4.5	–	5	1.5				–	–	1.5	V
	1,9	–	10	3				–	–	3	
	1.5,13.5	–	15	4				–	–	4	
Input High Voltage, V_{IH} Min.	0.5,4.5	–	5	3.5				3.5	–	–	
	1,9	–	10	7				7	–	–	
	1.5,13.5	–	15	11				11	–	–	
Input Current I_{IN} Max.		0,18	18	±0.1	±0.1	±1	±1	–	$±10^{-5}$	±0.1	μA

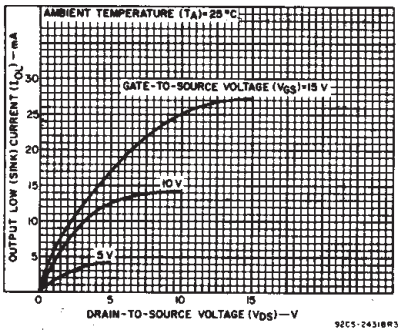


Fig. 2 – Typical output low (sink) current characteristics.

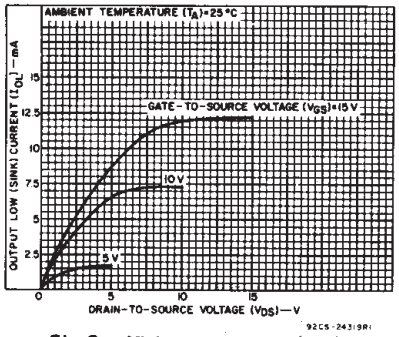


Fig. 3 – Minimum output low (sink) current characteristics.

3 COMMERCIAL CMOS HIGH VOLTAGE ICs

MAXIMUM RATINGS, *Absolute-Maximum Values:*

DC SUPPLY-VOLTAGE RANGE, (V_{DD})
(Voltages referenced to V_{SS} Terminal) ..... -0.5V to +20V
INPUT VOLTAGE RANGE, ALL INPUTS ..... -0.5V to V_{DD} +0.5V
DC INPUT CURRENT, ANY ONE INPUT ..... ±10mA
POWER DISSIPATION PER PACKAGE (P_D):
For T_A = -55°C to +100°C 500mW
For T_A = +100°C to +125°C ..... Derate Linearity at 12mW/°C to 200mW
DEVICE DISSIPATION PER OUTPUT TRANSISTOR
FOR T_A = FULL PACKAGE-TEMPERATURE RANGE (All Package Types) 100mW
OPERATING-TEMPERATURE RANGE (T_A) ..... -55°C to +125°C
STORAGE TEMPERATURE RANGE (T_{stg}) ..... -65°C to +150°C
LEAD TEMPERATURE (DURING SOLDERING):
At distance 1/16 ± 1/32 inch (1.59 ± 0.79mm) from case for 10s max ..... +265°C

DYNAMIC ELECTRICAL CHARACTERISTICS

At T_A = 25°C; Input t_r, t_f = 20 ns, C_L = 50 pF, R_L = 200kΩ

CHARACTERISTIC	TEST CONDITIONS		LIMITS		UNITS
		V_{DD} VOLTS	TYP.	MAX.	
Propagation Delay Time, t_{PHL}, t_{PLH}		5	150	300	ns
		10	75	150	
		15	55	110	
Transition Time, t_{THL}, t_{TLH}		5	100	200	ns
		10	50	100	
		15	40	80	
Input Capacitance, C_{IN}	Any Input		5	7.5	pF

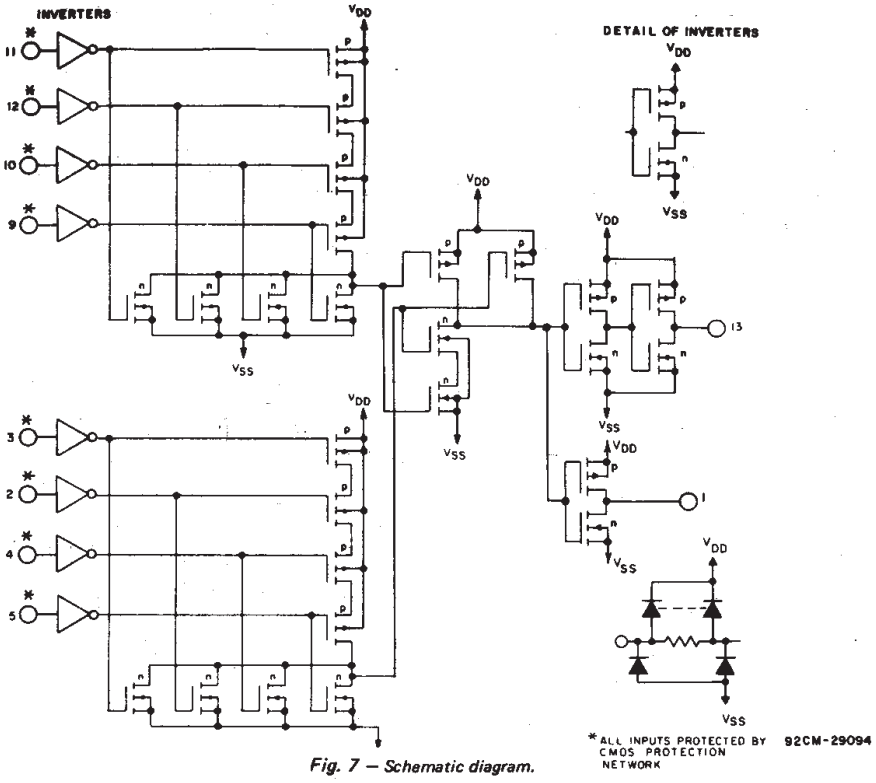


Fig. 7 — Schematic diagram.

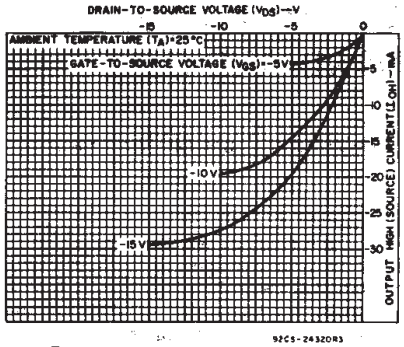


Fig. 4 — Typical output high (source) current characteristics.

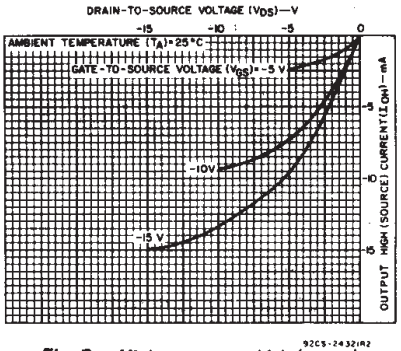


Fig. 5 — Minimum output high (source) current characteristics.

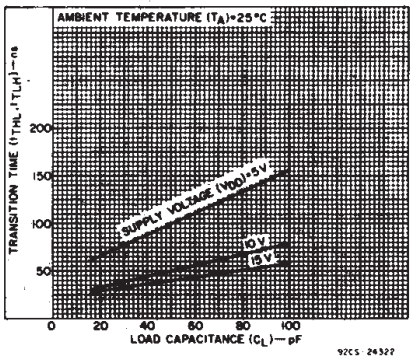


Fig. 6 — Typical transition time as a function of load capacitance.

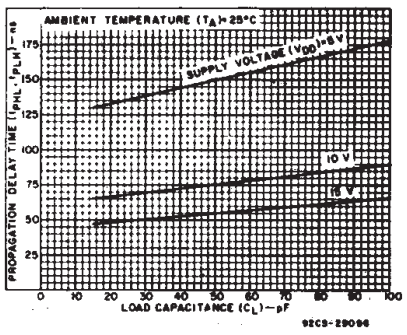


Fig. 8 — Typical propagation delay time as a function of load capacitance.

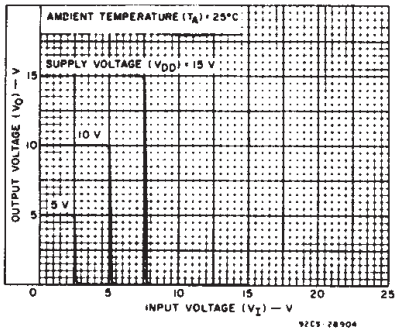


Fig. 9 – Typical voltage transfer characteristics (NAND output).

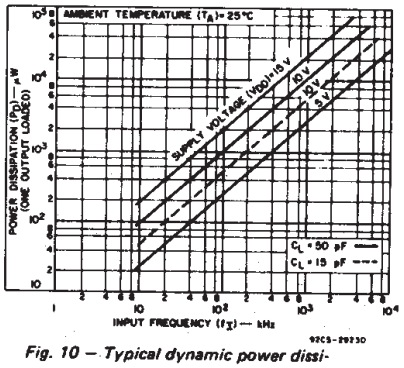


Fig. 10 – Typical dynamic power dissipation as a function of frequency.

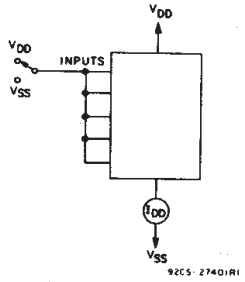


Fig. 11 – Quiescent-device-current test circuit.

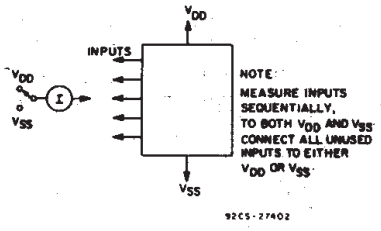


Fig. 12 – Input current test circuit.

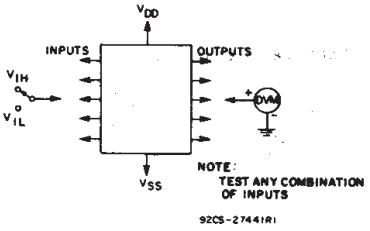


Fig. 13 – Input-voltage test circuit.

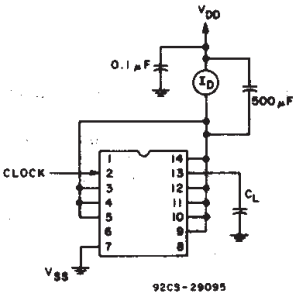
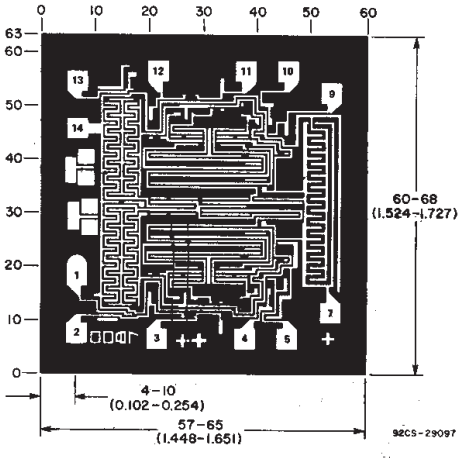


Fig. 14 – Dynamic power dissipation test circuit.



Dimensions and pad layout for CD4068BH.

Dimensions in parentheses are in millimeters and are derived from the basic inch dimensions as indicated. Grid graduations are in mils (10^{-3} inch).

PACKAGING INFORMATION

Orderable Device	Status (1)	Package Type	Package Drawing	Pins	Package Qty	Eco Plan (2)	Lead/Ball Finish	MSL Peak Temp (3)
CD4068BE	ACTIVE	PDIP	N	14	25	Pb-Free (RoHS)	CU NIPDAU	Level-NC-NC-NC
CD4068BF	ACTIVE	CDIP	J	14	1	None	Call TI	Level-NC-NC-NC
CD4068BF3A	ACTIVE	CDIP	J	14	1	None	Call TI	Level-NC-NC-NC
CD4068BM	ACTIVE	SOIC	D	14	50	Pb-Free (RoHS)	CU NIPDAU	Level-2-260C-1 YEAR/ Level-1-235C-UNLIM
CD4068BM96	ACTIVE	SOIC	D	14	2500	Pb-Free (RoHS)	CU NIPDAU	Level-2-260C-1 YEAR/ Level-1-235C-UNLIM
CD4068BMT	ACTIVE	SOIC	D	14	250	Pb-Free (RoHS)	CU NIPDAU	Level-2-260C-1 YEAR/ Level-1-235C-UNLIM
CD4068BNSR	ACTIVE	SO	NS	14	2000	Pb-Free (RoHS)	CU NIPDAU	Level-2-260C-1 YEAR/ Level-1-235C-UNLIM
CD4068BPW	ACTIVE	TSSOP	PW	14	90	Pb-Free (RoHS)	CU NIPDAU	Level-1-250C-UNLIM
CD4068BPWR	ACTIVE	TSSOP	PW	14	2000	Pb-Free (RoHS)	CU NIPDAU	Level-1-250C-UNLIM

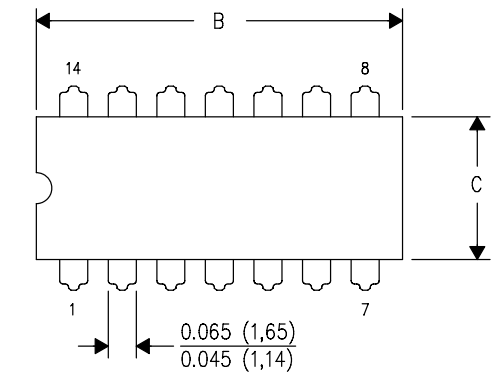
(1) The marketing status values are defined as follows:
ACTIVE: Product device recommended for new designs.
LIFEBUY: TI has announced that the device will be discontinued, and a lifetime-buy period is in effect.
NRND: Not recommended for new designs. Device is in production to support existing customers, but TI does not recommend using this part in a new design.
PREVIEW: Device has been announced but is not in production. Samples may or may not be available.
OBSOLETE: TI has discontinued the production of the device.

(2) Eco Plan - May not be currently available - please check http://www.ti.com/productcontent for the latest availability information and additional product content details.
None: Not yet available Lead (Pb-Free).
Pb-Free (RoHS): TI's terms "Lead-Free" or "Pb-Free" mean semiconductor products that are compatible with the current RoHS requirements for all 6 substances, including the requirement that lead not exceed 0.1% by weight in homogeneous materials. Where designed to be soldered at high temperatures, TI Pb-Free products are suitable for use in specified lead-free processes.
Green (RoHS & no Sb/Br): TI defines "Green" to mean "Pb-Free" and in addition, uses package materials that do not contain halogens, including bromine (Br) or antimony (Sb) above 0.1% of total product weight.

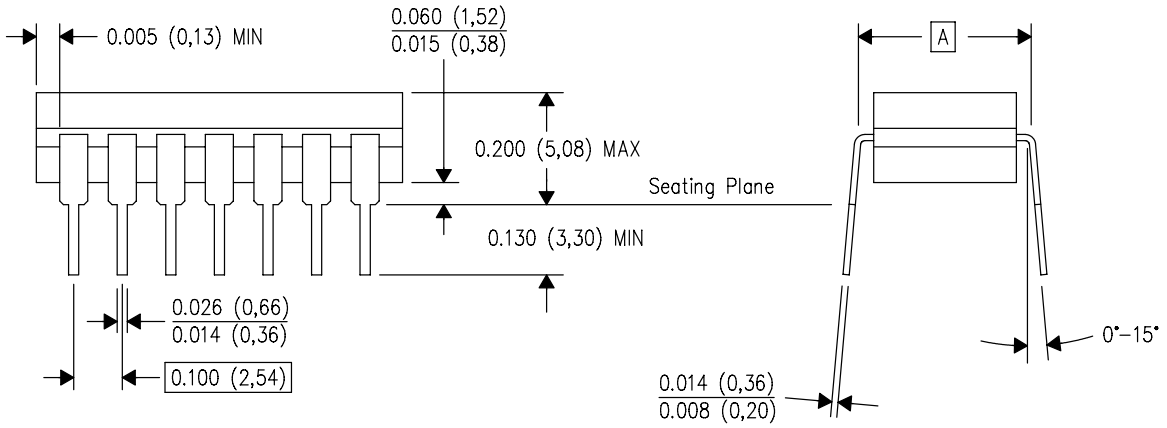
(3) MSL, Peak Temp. -- The Moisture Sensitivity Level rating according to the JEDECindustry standard classifications, and peak solder temperature.

J (R–GDIP–T**) CERAMIC DUAL IN–LINE PACKAGE

14 LEADS SHOWN



DIM \ PINS **	14	16	18	20
A	0.300 (7,62) BSC	0.300 (7,62) BSC	0.300 (7,62) BSC	0.300 (7,62) BSC
B MAX	0.785 (19,94)	.840 (21,34)	0.960 (24,38)	1.060 (26,92)
B MIN	—	—	—	—
C MAX	0.300 (7,62)	0.300 (7,62)	0.310 (7,87)	0.300 (7,62)
C MIN	0.245 (6,22)	0.245 (6,22)	0.220 (5,59)	0.245 (6,22)



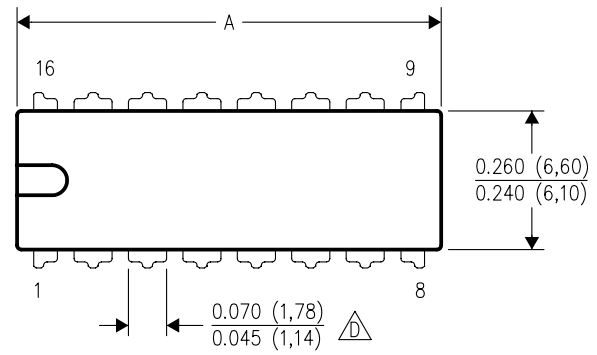
4040083/F 03/03

NOTES:
A. All linear dimensions are in inches (millimeters).
B. This drawing is subject to change without notice.
C. This package is hermetically sealed with a ceramic lid using glass frit.
D. Index point is provided on cap for terminal identification only on press ceramic glass frit seal only.
E. Falls within MIL STD 1835 GDIP1–T14, GDIP1–T16, GDIP1–T18 and GDIP1–T20.

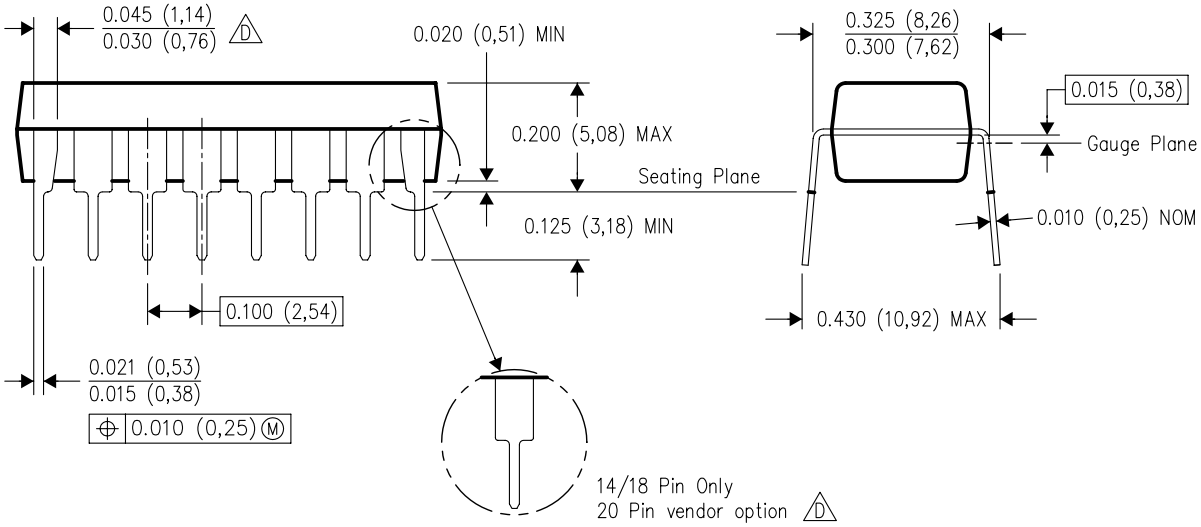
N (R–PDIP–T**)

PLASTIC DUAL–IN–LINE PACKAGE

16 PINS SHOWN



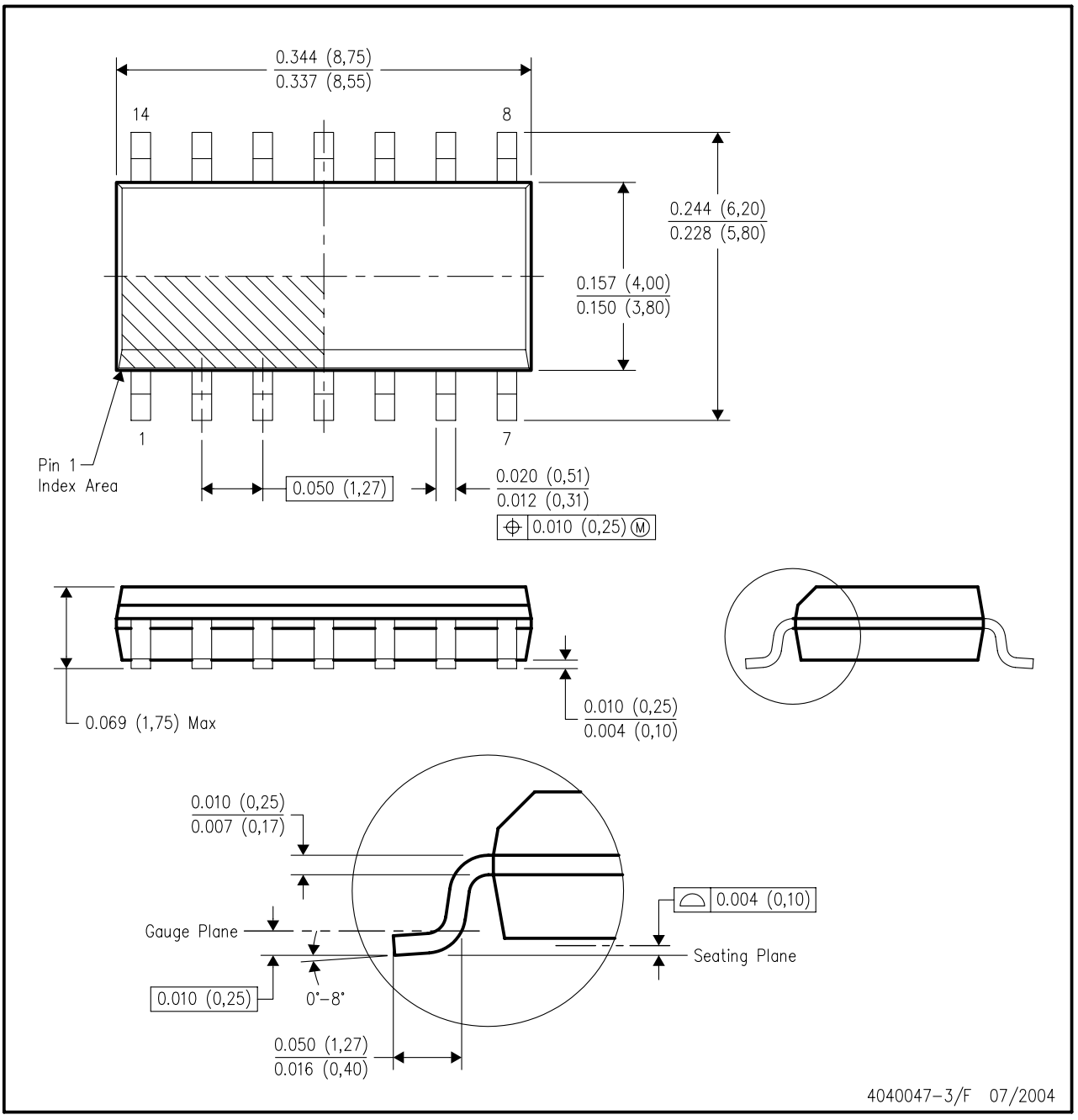
DIM \ PINS **	14	16	18	20
A MAX	0.775 (19,69)	0.775 (19,69)	0.920 (23,37)	1.060 (26,92)
A MIN	0.745 (18,92)	0.745 (18,92)	0.850 (21,59)	0.940 (23,88)
C MS–001 VARIATION	AA	BB	AC	AD



4040049/E 12/2002

NOTES:
A. All linear dimensions are in inches (millimeters).
B. This drawing is subject to change without notice.
C. Falls within JEDEC MS–001, except 18 and 20 pin minimum body length (Dim A).
D. The 20 pin end lead shoulder width is a vendor option, either half or full width.

D (R–PDSO–G14) PLASTIC SMALL–OUTLINE PACKAGE

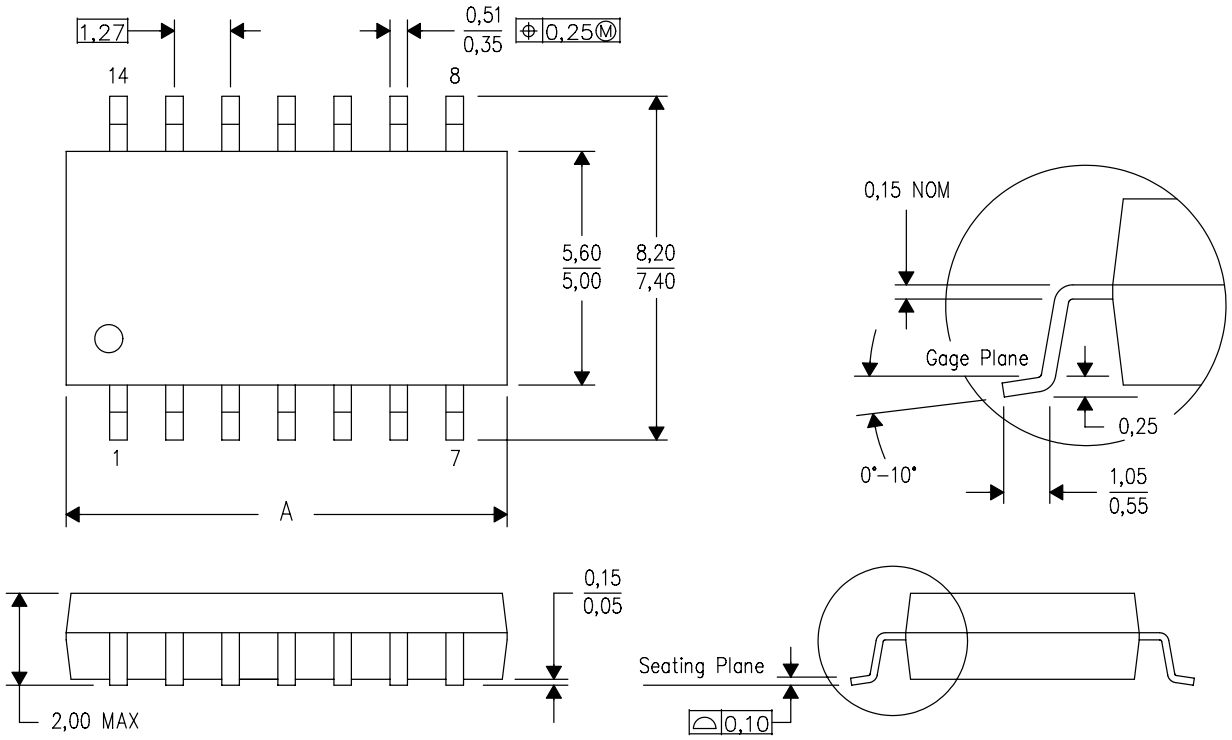


NOTES:
A. All linear dimensions are in inches (millimeters).
B. This drawing is subject to change without notice.
C. Body dimensions do not include mold flash or protrusion not to exceed 0.006 (0,15).
D. Falls within JEDEC MS–012 variation AB.

MECHANICAL DATA

NS (R-PDSO-G)** **PLASTIC SMALL-OUTLINE PACKAGE**

14-PINS SHOWN



DIM \ PINS **	14	16	20	24
A MAX	10,50	10,50	12,90	15,30
A MIN	9,90	9,90	12,30	14,70

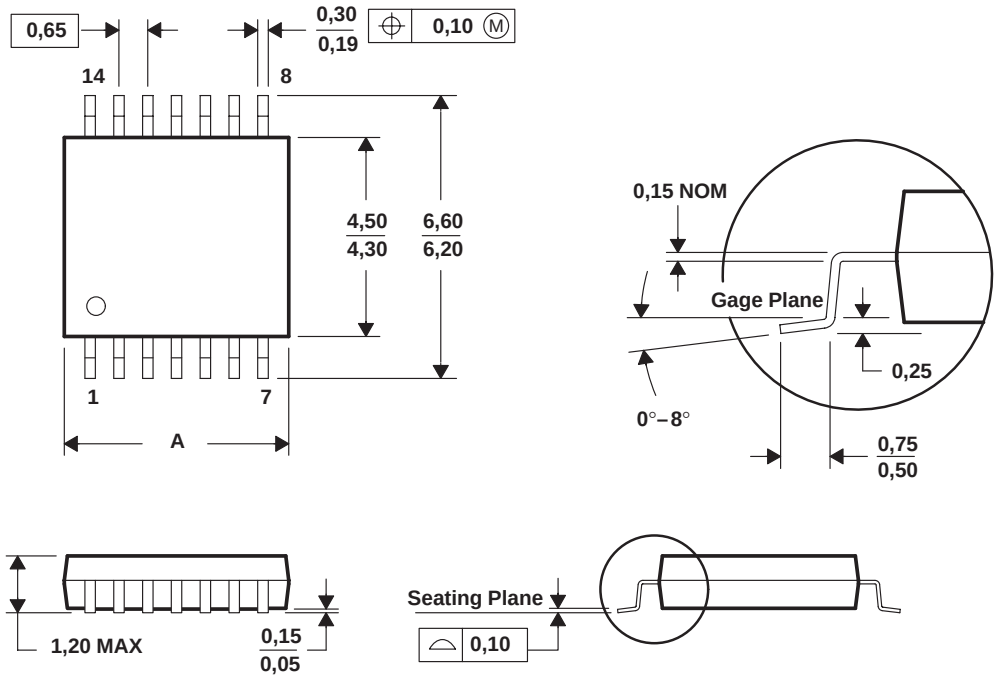
4040062/C 03/03

NOTES:
A. All linear dimensions are in millimeters.
B. This drawing is subject to change without notice.
C. Body dimensions do not include mold flash or protrusion, not to exceed 0,15.

MTSS001C – JANUARY 1995 – REVISED FEBRUARY 1999

PW (R-PDSO-G)** **PLASTIC SMALL-OUTLINE PACKAGE**

14 PINS SHOWN



DIM \ PINS **	8	14	16	20	24	28
A MAX	3,10	5,10	5,10	6,60	7,90	9,80
A MIN	2,90	4,90	4,90	6,40	7,70	9,60

4040064/F 01/97

NOTES: A. All linear dimensions are in millimeters.
B. This drawing is subject to change without notice.
C. Body dimensions do not include mold flash or protrusion not to exceed 0,15.
D. Falls within JEDEC MO-153

IMPORTANT NOTICE

Texas Instruments Incorporated and its subsidiaries (TI) reserve the right to make corrections, modifications, enhancements, improvements, and other changes to its products and services at any time and to discontinue any product or service without notice. Customers should obtain the latest relevant information before placing orders and should verify that such information is current and complete. All products are sold subject to TI's terms and conditions of sale supplied at the time of order acknowledgment.

TI warrants performance of its hardware products to the specifications applicable at the time of sale in accordance with TI's standard warranty. Testing and other quality control techniques are used to the extent TI deems necessary to support this warranty. Except where mandated by government requirements, testing of all parameters of each product is not necessarily performed.

TI assumes no liability for applications assistance or customer product design. Customers are responsible for their products and applications using TI components. To minimize the risks associated with customer products and applications, customers should provide adequate design and operating safeguards.

TI does not warrant or represent that any license, either express or implied, is granted under any TI patent right, copyright, mask work right, or other TI intellectual property right relating to any combination, machine, or process in which TI products or services are used. Information published by TI regarding third-party products or services does not constitute a license from TI to use such products or services or a warranty or endorsement thereof. Use of such information may require a license from a third party under the patents or other intellectual property of the third party, or a license from TI under the patents or other intellectual property of TI.

Reproduction of information in TI data books or data sheets is permissible only if reproduction is without alteration and is accompanied by all associated warranties, conditions, limitations, and notices. Reproduction of this information with alteration is an unfair and deceptive business practice. TI is not responsible or liable for such altered documentation.

Resale of TI products or services with statements different from or beyond the parameters stated by TI for that product or service voids all express and any implied warranties for the associated TI product or service and is an unfair and deceptive business practice. TI is not responsible or liable for any such statements.

Following are URLs where you can obtain information on other Texas Instruments products and application solutions:

Products		**Applications**	
Amplifiers	amplifier.ti.com	Audio	www.ti.com/audio
Data Converters	dataconverter.ti.com	Automotive	www.ti.com/automotive
DSP	dsp.ti.com	Broadband	www.ti.com/broadband
Interface	interface.ti.com	Digital Control	www.ti.com/digitalcontrol
Logic	logic.ti.com	Military	www.ti.com/military
Power Mgmt	power.ti.com	Optical Networking	www.ti.com/opticalnetwork
Microcontrollers	microcontroller.ti.com	Security	www.ti.com/security
		Telephony	www.ti.com/telephony
		Video & Imaging	www.ti.com/video
		Wireless	www.ti.com/wireless

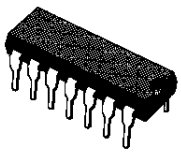
Mailing Address: Texas Instruments
Post Office Box 655303 Dallas, Texas 75265

SGS-THOMSON MICROELECTRONICS

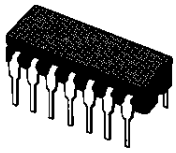
M54HC04
M74HC04

HEX INVERTER

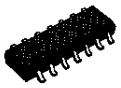
- HIGH SPEED
  t_{PD} = 6 ns (TYP.) AT V_{CC} = 5 V
- LOW POWER DISSIPATION
  I_{CC} = 1 µA (MAX.) AT T_A = 25 °C
- HIGH NOISE IMMUNITY
  $V_{NIH} = V_{NIL}$ = 28 % V_{CC} (MIN.)
- OUTPUT DRIVE CAPABILITY
  10 LSTTL LOADS
- SYMMETRICAL OUTPUT IMPEDANCE
  $|I_{OH}| = I_{OL}$ = 4 mA (MIN.)
- BALANCED PROPAGATION DELAYS
  $t_{PLH} = t_{PHL}$
- WIDE OPERATING VOLTAGE RANGE
  V_{CC} (OPR) = 2 V TO 6 V
- PIN AND FUNCTION COMPATIBLE WITH 54/74LS04



B1R
(Plastic Package)



F1R
(Ceramic Package)



M1R
(Micro Package)

C1R
(Chip Carrier)

ORDER CODES :

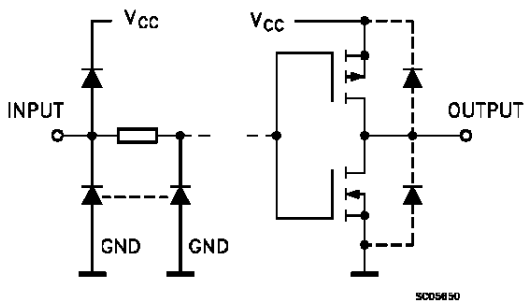
M54HC04F1R	M74HC04M1R
M74HC04B1R	M74HC04C1R

DESCRIPTION

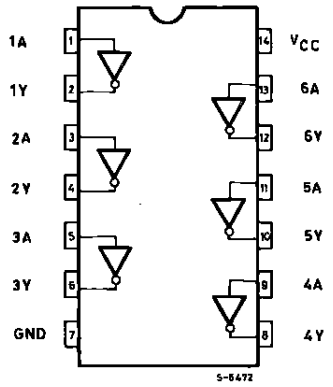
The M54/74HC04 is a high speed CMOS HEX INVERTER fabricated in silicon gate C^2MOS technology. It has the same high speed performance of LSTTL combined with true CMOS low power consumption.

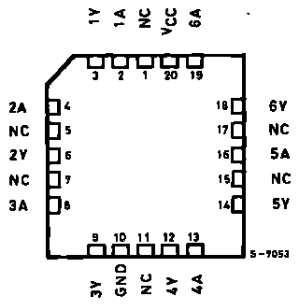
The internal circuit is composed of 3 stages including buffer output, which enables high noise immunity and stable output. All inputs are equipped with circuits against static discharge and transient excess voltage.

INPUT AND OUTPUT EQUIVALENT CIRCUIT



PIN CONNECTIONS (top view)



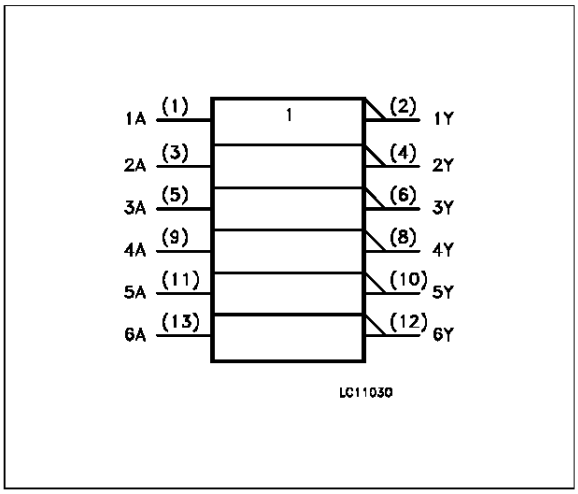


NC =
No Internal
Connection

TRUTH TABLE

A	Y
L	H
H	L

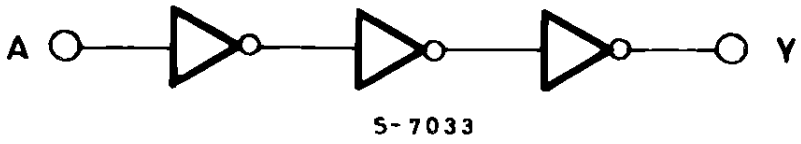
IEC LOGIC SYMBOL



PIN DESCRIPTION

PIN No	SYMBOL	NAME AND FUNCTION
1, 3, 5, 9, 11, 13	1A to 6A	Data Inputs
2, 4, 6, 8, 10, 12	1Y to 6Y	Data Outputs
7	GND	Ground (0V)
14	V_{CC}	Positive Supply Voltage

LOGIC DIAGRAM (Per Gate)



ABSOLUTE MAXIMUM RATINGS

Symbol	Parameter	Value	Unit
V_{CC}	Supply Voltage	-0.5 to +7	V
V_I	DC Input Voltage	-0.5 to V_{CC} + 0.5	V
V_O	DC Output Voltage	-0.5 to V_{CC} + 0.5	V
I_{IK}	DC Input Diode Current	± 20	mA
I_{OK}	DC Output Diode Current	± 20	mA
I_O	DC Output Source Sink Current Per Output Pin	± 25	mA
I_{CC} or I_{GND}	DC V_{CC} or Ground Current	± 50	mA
P_D	Power Dissipation	500 (*)	mW
T_{stg}	Storage Temperature	-65 to +150	°C
T_L	Lead Temperature (10 sec)	300	°C

Absolute Maximum Ratings are those values beyond which damage to the device may occur. Functional operation under these condition is not implied.
(*) 500 mW: ≅ 65 °C derate to 300 mW by 10mW/°C: 65 °C to 85 °C

RECOMMENDED OPERATING CONDITIONS

Symbol	Parameter		Value	Unit
V_{CC}	Supply Voltage		2 to 6	V
V_I	Input Voltage		0 to V_{CC}	V
V_O	Output Voltage		0 to V_{CC}	V
T_{op}	Operating Temperature: **M54HC** Series **M74HC** Series		-55 to +125 -40 to +85	^oC ^oC
t_r, t_f	Input Rise and Fall Time	V_{CC} = 2 V	0 to 1000	ns
		V_{CC} = 4.5 V	0 to 500	
		V_{CC} = 6 V	0 to 400	

DC SPECIFICATIONS

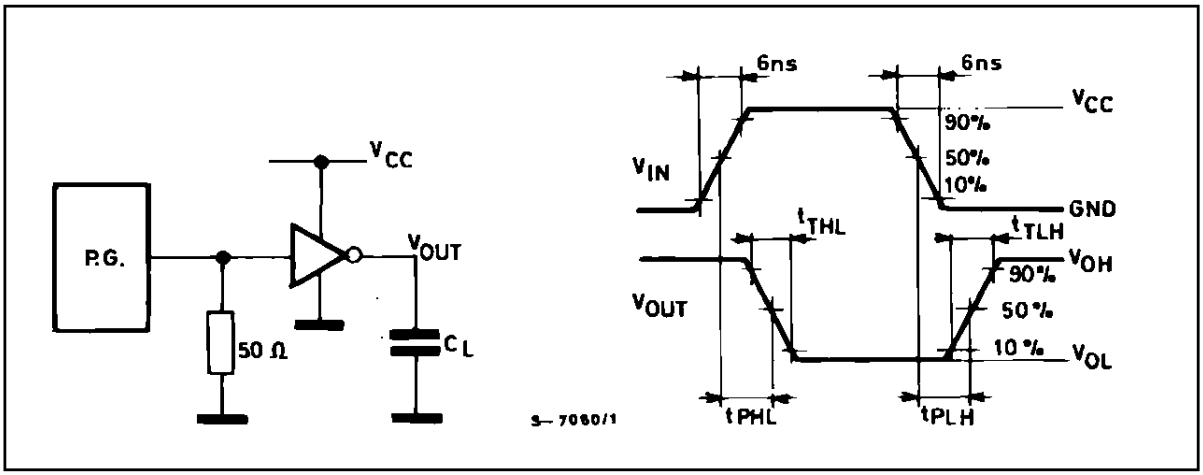
Symbol	Parameter	Test Conditions			Value							Unit
		V_{CC} (V)			T_A = 25 ^oC 54HC and 74HC			-40 to 85 ^oC 74HC		-55 to 125 ^oC 54HC		
					Min.	Typ.	Max.	Min.	Max.	Min.	Max.	
V_{IH}	High Level Input Voltage	2.0			1.5			1.5		1.5		V
		4.5			3.15			3.15		3.15		
		6.0			4.2			4.2		4.2		
V_{IL}	Low Level Input Voltage	2.0					0.5		0.5		0.5	V
		4.5					1.35		1.35		1.35	
		6.0					1.8		1.8		1.8	
V_{OH}	High Level Output Voltage	2.0	V_I = V_{IH} or V_{IL}	I_O=-20 μA	1.9	2.0		1.9		1.9		V
		4.5			4.4	4.5		4.4		4.4		
		6.0			5.9	6.0		5.9		5.9		
		4.5		I_O=-4.0 mA	4.18	4.31		4.13		4.10		
		6.0		I_O=-5.2 mA	5.68	5.8		5.63		5.60		
V_{OL}	Low Level Output Voltage	2.0	V_I = V_{IH} or V_{IL}	I_O= 20 μA		0.0	0.1		0.1		0.1	V
		4.5				0.0	0.1		0.1		0.1	
		6.0				0.0	0.1		0.1		0.1	
		4.5		I_O= 4.0 mA		0.17	0.26		0.33		0.40	
		6.0		I_O= 5.2 mA		0.18	0.26		0.33		0.40	
I_I	Input Leakage Current	6.0	V_I = V_{CC} or GND				±0.1		±1		±1	μA
I_{CC}	Quiescent Supply Current	6.0	V_I = V_{CC} or GND				1		10		20	μA

AC ELECTRICAL CHARACTERISTICS (C_L = 50 pF, Input $t_r = t_f$ = 6 ns)

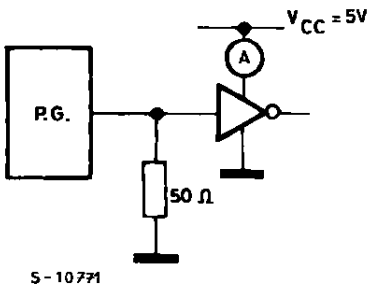
Symbol	Parameter	Test Conditions		Value							Unit
		V_{CC} (V)		T_A = 25 °C 54HC and 74HC			-40 to 85 °C 74HC		-55 to 125 °C 54HC		
				Min.	Typ.	Max.	Min.	Max.	Min.	Max.	
t_{TLH} t_{THL}	Output Transition Time	2.0			30	75		95		110	ns
		4.5			8	15		19		22	
		6.0			7	13		16		19	
t_{PLH} t_{PHL}	Propagation Delay Time	2.0			27	75		95		110	ns
		4.5			9	15		19		22	
		6.0			8	13		16		19	
C_{IN}	Input Capacitance				5	10		10		10	pF
C_{PD} (*)	Power Dissipation Capacitance				22						pF

(*) C_{PD} is defined as the value of the IC's internal equivalent capacitance which is calculated from the operating current consumption without load. (Refer to Test Circuit). Average operting current can be obtained by the following equation. $I_{CC}(opr) = C_{PD} \bullet V_{CC} \bullet f_{IN} + I_{CC}/6$ (per Gate)

SWITCHING CHARACTERISTICS TEST CIRCUIT



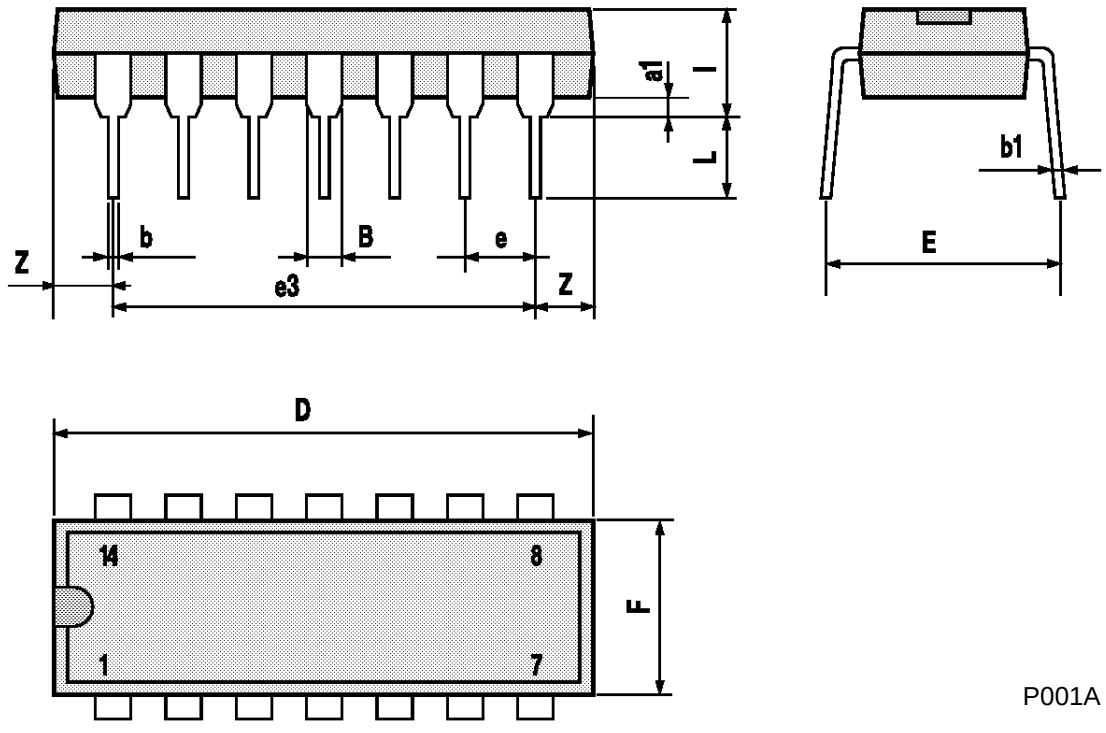
TEST CIRCUIT I_{CC} (Opr.)



INPUT WAVEFORM IS THE SAME AS THAT IN CASE OF SWITCHING CHARACTERISTICS TEST.

Plastic DIP14 MECHANICAL DATA

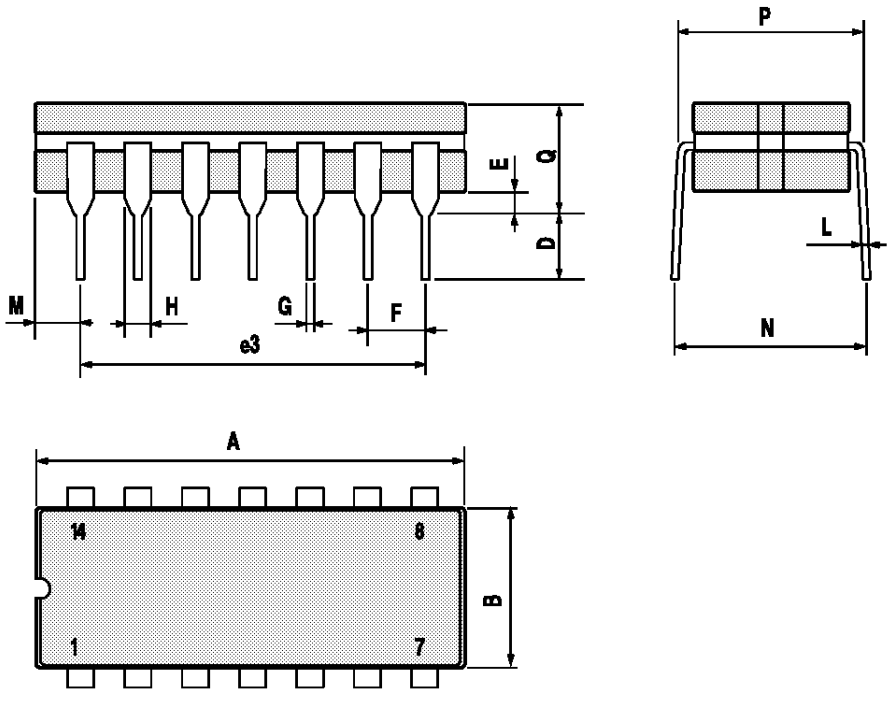
DIM.	mm			inch		
	MIN.	TYP.	MAX.	MIN.	TYP.	MAX.
a1	0.51			0.020		
B	1.39		1.65	0.055		0.065
b		0.5			0.020	
b1		0.25			0.010	
D			20			0.787
E		8.5			0.335	
e		2.54			0.100	
e3		15.24			0.600	
F			7.1			0.280
I			5.1			0.201
L		3.3			0.130	
Z	1.27		2.54	0.050		0.100



P001A

Ceramic DIP14/1 MECHANICAL DATA

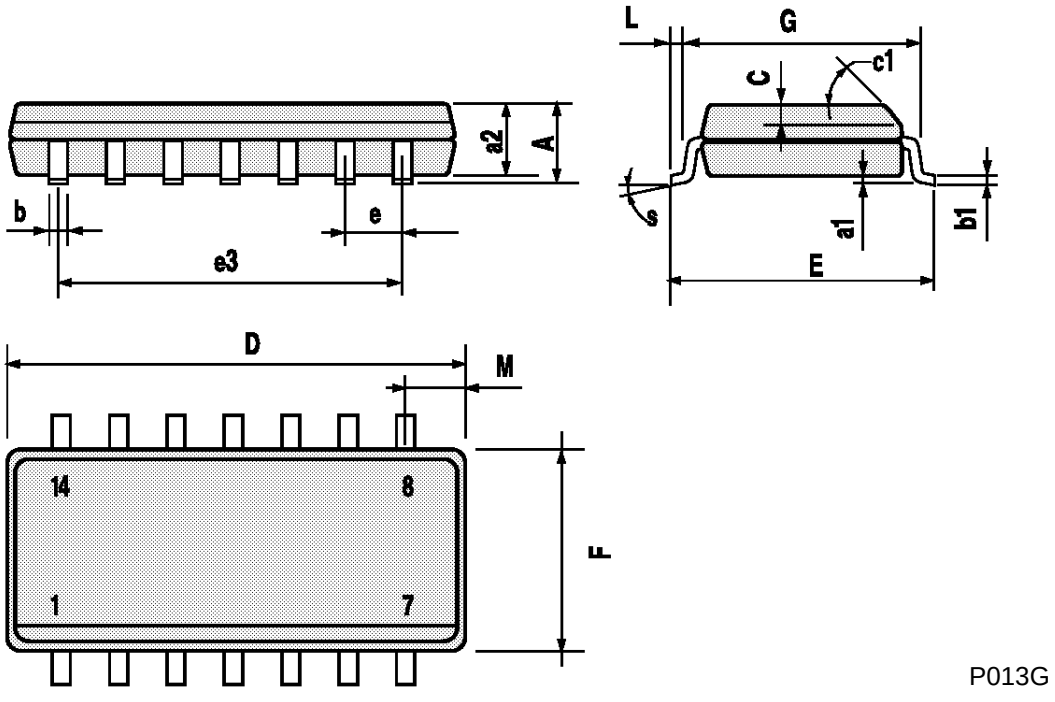
DIM.	mm			inch		
	MIN.	TYP.	MAX.	MIN.	TYP.	MAX.
A			20			0.787
B			7.0			0.276
D		3.3			0.130	
E	0.38			0.015		
e3		15.24			0.600	
F	2.29		2.79	0.090		0.110
G	0.4		0.55	0.016		0.022
H	1.17		1.52	0.046		0.060
L	0.22		0.31	0.009		0.012
M	1.52		2.54	0.060		0.100
N			10.3			0.406
P	7.8		8.05	0.307		0.317
Q			5.08			0.200



P053C

SO14 MECHANICAL DATA

DIM.	mm			inch		
	MIN.	TYP.	MAX.	MIN.	TYP.	MAX.
A			1.75			0.068
a1	0.1		0.2	0.003		0.007
a2			1.65			0.064
b	0.35		0.46	0.013		0.018
b1	0.19		0.25	0.007		0.010
C		0.5			0.019	
c1	45° (typ.)					
D	8.55		8.75	0.336		0.344
E	5.8		6.2	0.228		0.244
e		1.27			0.050	
e3		7.62			0.300	
F	3.8		4.0	0.149		0.157
G	4.6		5.3	0.181		0.208
L	0.5		1.27	0.019		0.050
M			0.68			0.026
S	8° (max.)					



P013G

PLCC20 MECHANICAL DATA

DIM.	mm			inch		
	MIN.	TYP.	MAX.	MIN.	TYP.	MAX.
A	9.78		10.03	0.385		0.395
B	8.89		9.04	0.350		0.356
D	4.2		4.57	0.165		0.180
d1		2.54			0.100	
d2		0.56			0.022	
E	7.37		8.38	0.290		0.330
e		1.27			0.050	
e3		5.08			0.200	
F		0.38			0.015	
G			0.101			0.004
M		1.27			0.050	
M1		1.14			0.045