

Module IV: Embedded Hardware Design & Development

Dr. Rejeesh Rayaroth

Asst. Professor CSE

MITE

SYLLABUS

Module 4: Embedded Hardware Design and Development

No. of Hrs: 9+4

Analog & Digital Electronics Components, Electronic Design Automation tools, Embedded Firmware Design approaches: Super loop based approach & operating system based approach, Firmware development languages: Assembly language & High Level language

Laboratory Components:

1. Using Keil software write a program to count the number of ones and zeros in two consecutive memory locations.
2. Using Keil software display “Hello World” message using Internal UART.

INTRODUCTION

❑ **Embedded System**

❖ Combination of **hardware & software (firmware)** for special-purpose tasks.

❑ **Hardware Components:**

❖ Includes analog & digital circuits, electronic components, and Integrated Circuits (ICs).

❑ **PCB (Printed Circuit Board):** Acts as the platform for hardware components, similar to a chessboard.

❑ **Breadboard vs PCB:**

❖ Breadboard – Used for testing, unstable, and bulky for commercial products.

❑ **PCB** :Backbone of embedded hardware, stable for mass production.

❑ This particular section of the module focusses on

❖ Refreshing analog & digital components knowledge.

❖ Introduces IC design principles.

❖ Covers PCB fundamentals, design & EDA tools.

ANALOG ELECTRONIC COMPONENTS

Common Components

- ❑ **Resistors** – Limit current flow in circuits.
- ❑ **Capacitors** – Used in signal filtering, power supply decoupling, and RF circuit matching.
- ❑ **Diodes** – Control direction of current flow, protect circuits.
- ❑ **Inductors** – Filter power supply, remove noise.
- ❑ **Operational Amplifiers (OpAmps)** – Signal amplification, processing.
- ❑ **Transistors** – Act as switches or amplifiers in electronic circuits.

❑ Resistor Applications

- ❖ Used to interface LEDs, buzzers, etc. with microcontroller pins.
- ❖ Prevent excess current flow in embedded circuits.

❑ Capacitor Applications

- ❖ Used in reset circuits, RF matching circuits, and power decoupling.
- ❖ Types: Electrolytic, ceramic, and tantalum capacitors.

❑ Inductor Applications

- ❖ Used for power supply filtering, ripple removal, and noise reduction.
- ❖ Commonly used in μH range for filters and matching circuits.

DIODES IN EMBEDDED HARDWARE CIRCUITS

□ Commonly Used Diodes

- ❖ **P-N Junction Diode** – Standard diode for general applications.
- ❖ **Schottky Diode** – Lower forward voltage drop (0.15V – 0.45V) than P-N junction diodes (0.7V – 1.7V).
- ❖ **Zener Diode** – Allows reverse current flow beyond breakdown voltage, used in voltage regulation.

□ Schottky Diode Features

- ❖ Low forward voltage drop → Reduced power loss.
- ❖ Fast switching speed → Ideal for high-frequency applications.

❑ Zener Diode Applications

- ❖ **Voltage Clamping** : Maintains desired voltage levels.
- ❖ **Reverse Polarity Protection**: Prevents damage from incorrect connections.
- ❖ **AC-DC Rectification**: Used in power conversion circuits.
- ❖ **Freewheeling Diode**: Manages current in inductive circuits (motors, coils).
- ❖ **Brown-Out Protection**: Stabilizes voltage in fluctuating power conditions

TRANSISTORS IN EMBEDDED APPLICATIONS

❑ Purpose of Transistors

- ❖ **Switching** – Operates in ON or OFF state.
- ❖ **Amplification** – Always partially ON, allowing variable current flow.

❑ Switching Applications

- ❖ Used in *digital circuits* where transistors act as *electronic switches*.
- ❖ Helps control devices like *relays*, *buzzers*, and *motors*.

❑ Amplification Applications

- ❖ Maintains *variable current* below *saturation levels*.
- ❖ Used for *signal processing* and *boosting*.

❑ Common Emitter Configuration

- ❖ Widely used in *embedded driving circuits*.
- ❖ Ideal for *relay*, *buzzer*, and *stepper motor control*.

DIGITAL ELECTRONIC COMPONENTS

❑ Digital Electronics components of embedded system

❖ Works with *digital/discrete signals* instead of analog signals.

❑ Key Components

❖ Microprocessors, Microcontrollers, and SoCs *process digital data*.

❑ **Digital I/O Interfaces:** Facilitate communication between digital systems and external devices.

❑ **Glue Logic:** Custom digital circuitry ensuring compatibility between different IC chips.

❖ Examples: Address decoders, latches, encoders/decoders.

❑ **Logic Standards** – Define electrical characteristics of digital signals.

❖ **TTL** (Transistor-Transistor Logic) – Used in fast digital circuits.

❖ **CMOS** (Complementary Metal Oxide Semiconductor) : Low-power consumption logic standard.

OPEN COLLECTOR AND TRI-STATE OUTPUT

- ❑ Open collector is an I/O interface standard in digital system design.
- ❑ The term ‘open collector’ is commonly used in conjunction with the output of an Integrated Circuit (IC) chip.
- ❑ It facilitates the interfacing of IC output to other systems which operate at different voltage levels.
- ❑ In the open collector configuration, the output line from an IC circuit is connected to the base of an NPN transistor.
- ❑ The collector of the transistor is left unconnected (floating) and the emitter is internally connected to the ground signal of IC.
- ❑ Figure illustrates an open collector output configuration

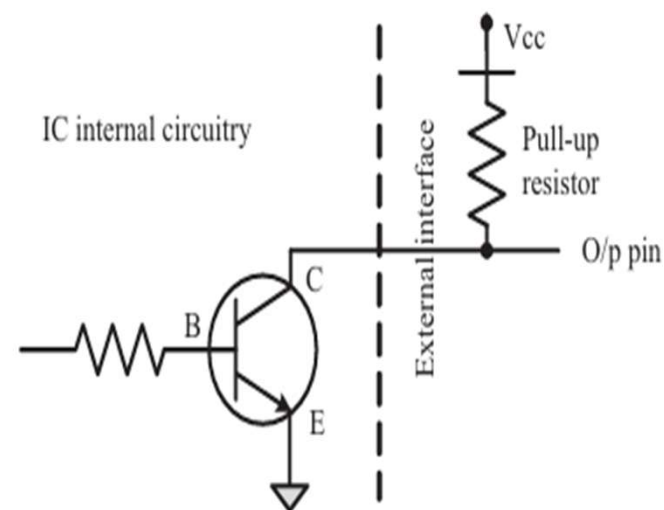


Fig. 8.1 Open collector output configuration

Open Collector Output Basics

- ❑ **Pull-up resistor** ensures the output pin is at the correct voltage.
- ❑ **High Impedance State** – When the transistor is ON, the output floats (not driven to HIGH or LOW).
- ❑ **Logic Levels:**
 - ❑ Base drive ON → Output = Logic 0 (0V).
 - ❑ Base drive OFF → Output = Logic HIGH (V_{cc}).

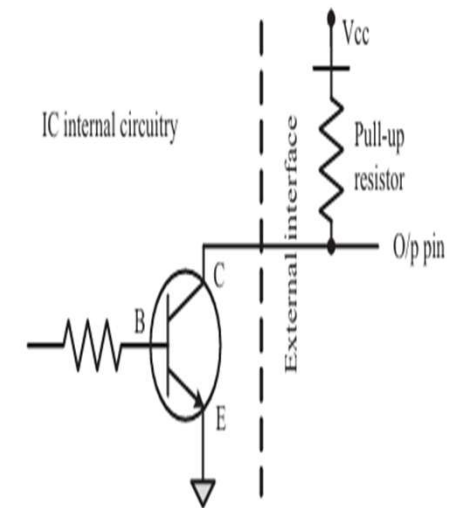


Fig. 8.1 Open collector output configuration

ADVANTAGES OF OPEN COLLECTOR CONFIGURATION

Advantages of Open Collector Configuration

- ❑ Interfacing devices with different voltage levels without extra interface circuits.
- ❑ Multi-drop connections for communication interfaces like I2C, 1-Wire.
- ❑ Easier implementation of Wired AND / Wired OR logic in circuits.

TRI-STATE LOGIC DEVICES

❑ Three states:

- ❖ Logic 0 (LOW)
- ❖ Logic 1 (HIGH)
- ❖ High Impedance (FLOAT)

❑ Device Enable Line controls output state:

- ❑ Enabled → Device functions normally (LOW or HIGH).
- ❑ Disabled → Device enters FLOAT state (acts like it's disconnected).

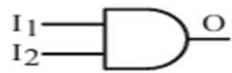
❑ Allows multiple devices to share a common bus, ensuring only one drives it at a time.

LOGIC GATES

- ❑ Logic gates are the building blocks of digital circuits.
- ❑ Logic gates control the flow of digital information by performing a logical operation of the input signals.
- ❑ Depending on the logical operation, the logic gates used in digital design are classified into AND, OR, XOR, NOT, NAND, NOR, and XNOR.
- ❑ The logical relationship between the output signal and the input signals for a logic gate is represented using a truth table.
- ❑ Figure in the next slide illustrates the truth table and symbolic representation of each logic gate.

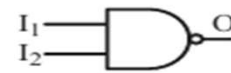
AND Gate –Truth Table

I ₁	I ₂	O
0	0	0
0	1	0
1	0	0
1	1	1



NAND Gate –Truth Table

I ₁	I ₂	O
0	0	1
0	1	1
1	0	1
1	1	0



OR Gate –Truth Table

I ₁	I ₂	O
0	0	0
0	1	1
1	0	1
1	1	1



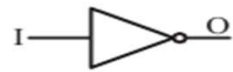
NOR Gate –Truth Table

I ₁	I ₂	O
0	0	1
0	1	0
1	0	0
1	1	0



NOT Gate –Truth Table

I	O
0	1
1	0



XOR Gate –Truth Table

I ₁	I ₂	O
0	0	0
0	1	1
1	0	1
1	1	0



XNOR Gate –Truth Table

I ₁	I ₂	O
0	0	1
0	1	0
1	0	0
1	1	1



Fig. 8.2 Logic Gates Truth Table and Symbolic representation

BUFFER CIRCUITS & TRI-STATE BUFFERS IN EMBEDDED SYSTEMS

□ Buffer Circuit Function

- ❖ Amplifies current or power.
- ❖ Enhances the driving capability of logic circuits.

□ Tri-State Buffer

- ❖ Includes an Output Enable control.
- ❖ When Enable is active → Functions as a buffer.
- ❖ When Enable is inactive → Output remains in high impedance state (floating).

Applications of Tri-State Buffers

- ❖ Used as drivers for address bus.
- ❖ Device selection among multiple devices in a shared data bus.

Types of Tri-State Buffers

- ❖ Unidirectional Buffers – Signal flows in one direction.
- ❖ Bi-Directional Buffers – Allows two-way communication.

Example – 74LS244/74HC244

- ❖ Unidirectional octal buffer with 8 individual buffers.
- ❖ Buffers are grouped into two, each with its own output enable line

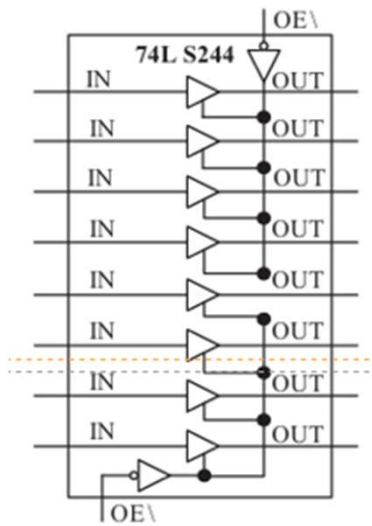


Fig. 8.3 74LS244 Octal Buffer IC

- ❑ IC 74LS245 is an example of bi-directional tri-state buffer.
- ❑ It allows data flow in both directions, one at a time.
- ❑ The data flow direction can be set by the direction control line.
- ❑ One buffer is allocated for the data line associated with each direction.
- ❑ Figure illustrates the 74LS245 octal bi-directional buffer

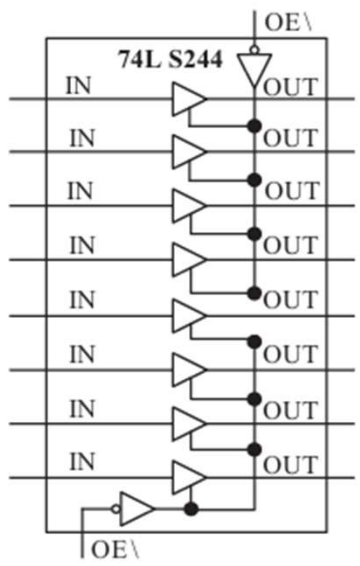


Fig. 8.3 74LS244 Octal Buffer IC

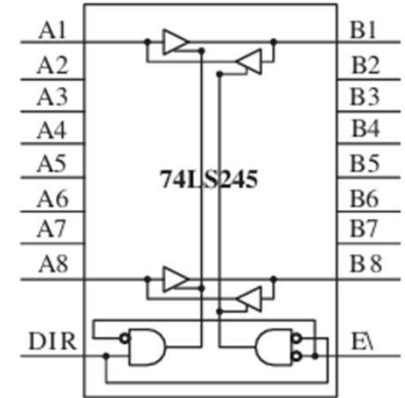


Fig. 8.4 74LS245 Octal bidirectional Buffer IC

LATCH

❑ What is a Latch?

- ❖ Stores binary data temporarily.
- ❖ Contains input data line, clock/gating control, and output line.
- ❖ Triggered by a positive edge (rising) or negative edge (falling) signal.

❑ Functionality of a Latch

- ❖ On trigger, it captures input data.
- ❖ Stored data remains on output until the next trigger occurs.

❑ Example:

- ❖ D Flip-Flop, commonly used in electronic circuits.

□ Applications of Latches

- ❖ Short-duration data storage in embedded systems.
- ❖ Used in multiplexed address-data bus systems to store lower-order address info.

□ Types of Latches

- ❖ S-R, J-K, D, T Latches – Different latch designs for varied applications.

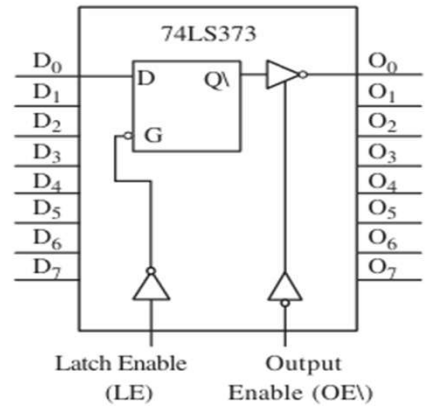
Example – IC 74LS373

Contains 8 individual D latches.

Commonly used in multiplexed bus systems for address latching.

Address Latch Enable (ALE) pulse triggers data storage when the address bits appear on the multiplexed bus.

Figure illustrates the usage of latches in address latching.



IC Fig. 8.5 74LS373 Octal Latch IC

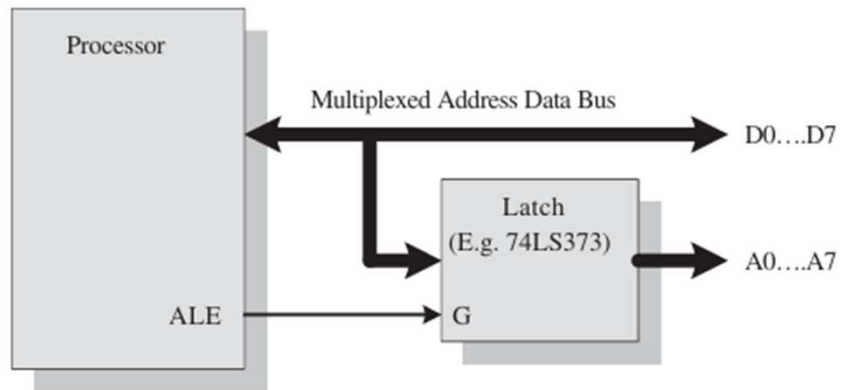


Fig. 8.6 Latch IC for address latching in multiplexed address data-bus

DECODER

- ❑ **Definition:** A logic circuit that generates all possible combinations of input signals.
- ❑ **Naming Convention:** Based on input line numbers and possible output combinations (e.g., 2-to-4, 3-to-8, 4-to-16 decoders).
- ❑ **3-to-8 Decoder:**
 - ❑ Contains 3 input signal lines.
 - ❑ Can generate 8 different configurations (000 to 111 in binary corresponds to 0 to 7 in output).
 - ❑ **Output assertion:** Corresponding output line activates based on input signals (e.g., input 001 → output line 2 asserted).

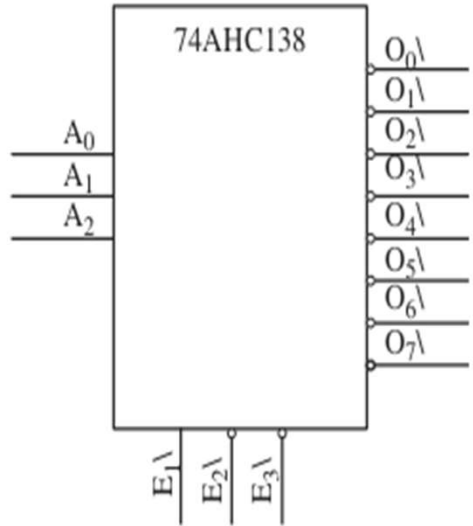
□ Applications:

- ❖ Address decoding in electronic circuits.
- ❖ Chip select signal generation.
- ❖ Used in integrated circuits (ICs).

□ Example ICs:

- ❖ 74LS138 / 74AHC138 – Commonly used 3-to-8 decoder IC.

Illustration: Functional diagram of 74AHC138 decoder and function table.



Input						Output							
A ₂	A ₁	A ₀	E ₁	E ₂	E ₃	O ₀	O ₁	O ₂	O ₃	O ₄	O ₅	O ₆	O ₇
0	0	0	0	0	1	0	1	1	1	1	1	1	1
0	0	1	0	0	1	1	0	1	1	1	1	1	1
0	1	0	0	0	1	1	1	0	1	1	1	1	1
0	1	1	0	0	1	1	1	1	0	1	1	1	1
1	0	0	0	0	1	1	1	1	1	0	1	1	1
1	0	1	0	0	1	1	1	1	1	1	0	1	1
1	1	0	0	0	1	1	1	1	1	1	1	0	1
1	1	1	0	0	1	1	1	1	1	1	1	1	0

Fig. 8.7 3 to 8 Decoder IC and I/O signal states

ENCODER

- ❑ **Definition:** Performs the reverse operation of a decoder by encoding the input state into a specific output format.
- ❑ **Binary Encoder:**
 - ❖ Converts input signals into their binary equivalent.
 - ❖ Naming convention based on input line numbers and output format (e.g., 4-to-2, 8-to-3, 16-to-4 encoders).
- ❑ **8-to-3 Encoder:**
 - ❖ Has 8 input signal lines.
 - ❖ Generates a 3-bit binary output corresponding to the active input (e.g., input 0 to 7 → binary 111 to 000).
 - ❖ Output assertion: Input line 1 asserted → outputs $A_0 = 0$, $A_1 = 1$, $A_2 = 1$.

□ Applications:

- ❖ Address decoding in electronic circuits.
- ❖ Chip select signal generation.
- ❖ Available as integrated circuits (ICs).

□- Example ICs:

- ❖ 74F148 / 74LS148 – Commonly used 8-to-3 encoder IC.

❑ Illustration: Functional diagram of 74F148/74LS148 encoder and function table

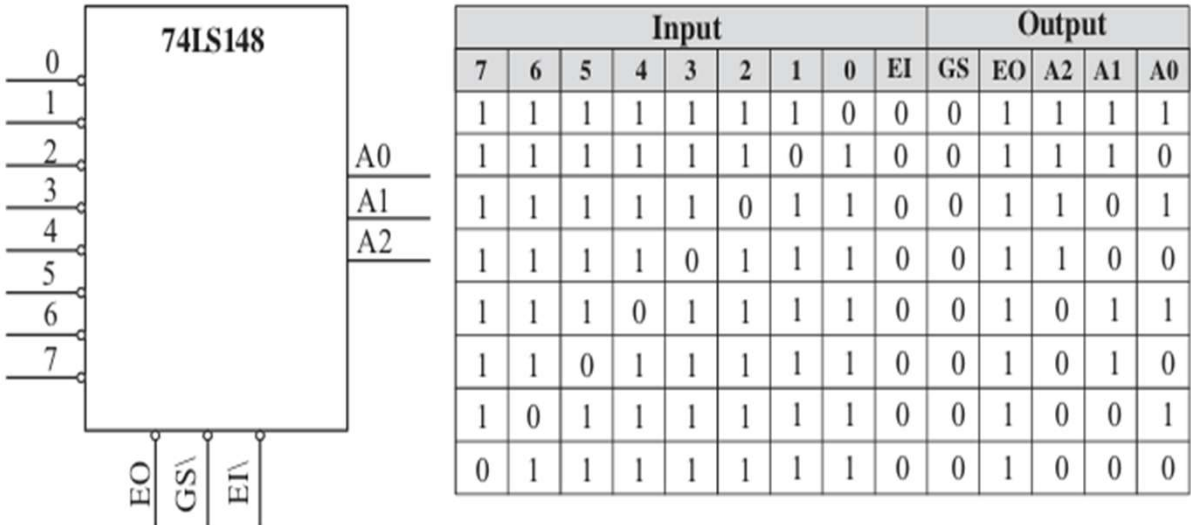


Fig. 8.8 8 to 3 Encoder IC and I/O signal states

ENCODER OUTPUT CONTROL

❑ Enable Input (EI) Signal:

- ❖ Encoder output is enabled when $EI = 0$.
- ❖ $EI = \text{High}$ forces all outputs inactive (High state) to prevent erroneous data.

❑ Group Signal (GS):

- ❖ Active-Low when any input is Low.
- ❖ Indicates when an input is active.

❑ Enable Output (EO):

- ❖ Active-Low when all inputs are High.

Input									Output				
7	6	5	4	3	2	1	0	EI	GS	EO	A2	A1	A0
1	1	1	1	1	1	1	0	0	0	1	1	1	1
1	1	1	1	1	1	0	1	0	0	1	1	1	0
1	1	1	1	1	0	1	1	0	0	1	1	0	1
1	1	1	1	0	1	1	1	0	0	1	1	0	0
1	1	1	0	1	1	1	1	0	0	1	0	1	1
1	1	0	1	1	1	1	1	0	0	1	0	1	0
1	0	1	1	1	1	1	1	0	0	1	0	0	1
0	1	1	1	1	1	1	1	0	0	1	0	0	0

❑ Priority Encoding:

- ❖ 74LS148 / 74F148 are priority encoders.
- ❖ Ensures only the highest order data line is encoded (e.g., input 1 & 6 asserted → only 6 is encoded → output = 001).

❑ Inverted Output Encoding:

- ❖ Encoded output is the inverted value of the corresponding binary data.
- ❖ Example: Input 7 asserted → Output lines A2, A1, A0 = 000.

❑ Real-World Application:

- ❖ Keyboard encoding: Converts each keypress into binary code.

MULTIPLEXER (MUX)

❑ Definition:

- ❖ Acts as a digital switch, connecting one input from a set of many input to a single output.
- ❖ Allows one input to be connected to the output at a time.

❑ Structure:

- ❖ Multiple input lines, single output line.
- ❖ The inputs are multiplexed.

❑- Selection Mechanism:

- ❖ The MUX control lines determine which input is routed to the output.

❑ Example IC:

- ❖ 74S151 – 8-to-1 multiplexer IC.

Illustration:

❖ Functional diagram and function table of 74S151 multiplexer.

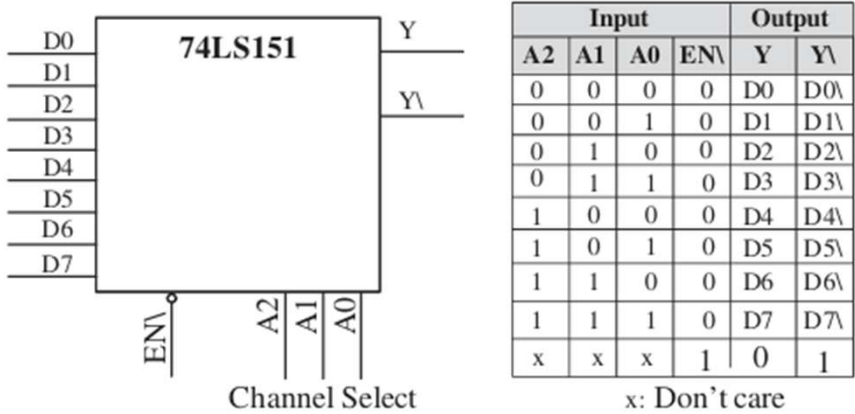


Fig. 8.9 8 to 1 multiplexer IC and I/O signal states

MULTIPLEXER CONTROL & SELECTION

❑ Enable Signal (EN) Function:

- ❖ MUX is enabled when $EN = 0$.
- ❖ $EN = \text{High}$ forces the output inactive (Low state).

❑ Channel Selection Mechanism:

- ❑ Input signal is routed to the output through channel select lines: A_2, A_1, A_0 .

❑ Binary Selection of Input Lines:

- ❖ Apply the binary equivalent of the desired input to A_0, A_1 , and A_2 .

❑ Example selections:

- ❖ $A_2A_1A_0 = 000 \rightarrow$ Selects Input D_0 .
- ❖ $A_2A_1A_0 = 001 \rightarrow$ Selects Input D_1 .

Input				Output	
A_2	A_1	A_0	EN	Y	$Y\backslash$
0	0	0	0	D_0	$D_0\backslash$
0	0	1	0	D_1	$D_1\backslash$
0	1	0	0	D_2	$D_2\backslash$
0	1	1	0	D_3	$D_3\backslash$
1	0	0	0	D_4	$D_4\backslash$
1	0	1	0	D_5	$D_5\backslash$
1	1	0	0	D_6	$D_6\backslash$
1	1	1	0	D_7	$D_7\backslash$
x	x	x	1	0	1

x: Don't care

DEMULTIPLEXER

❑ Definition:

- ❖ Reverse operation of a multiplexer.
- ❖ Routes a single input signal to selected output lines.

❑ Selection Mechanism:

- ❖ The output selector control lines determine which output line receives the input signal.

❑ Example IC:

- ❖ NL7SZ18 – A typical 1-to-2 de-multiplexer IC.

❑ Structure:

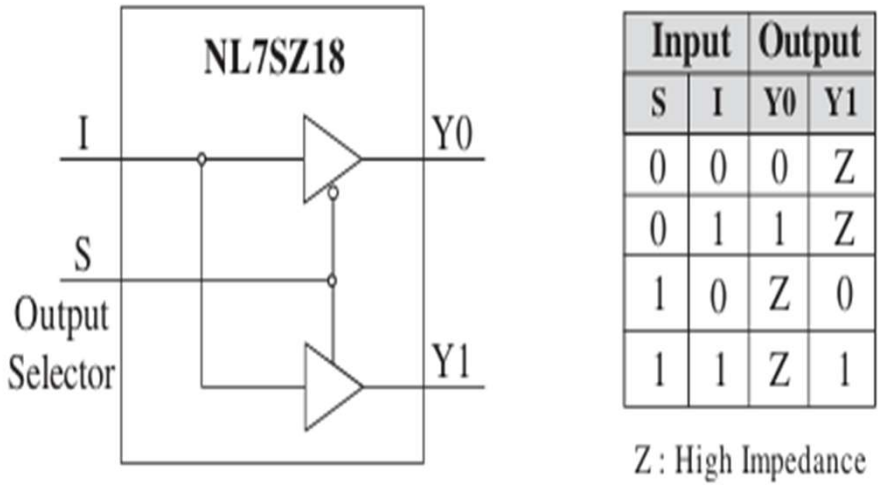
- ❖ Single input line.
- ❖ Two output lines for switching the input signal.

❑ Output Switching:

- ❖ Controlled by the output selector control lines.

❑ Illustration:

❖ Functional diagram and function table of NL7SZ18 de-multiplexer.



Input		Output	
S	I	Y0	Y1
0	0	0	Z
0	1	1	Z
1	0	Z	0
1	1	Z	1

Z : High Impedance

Fig. 8.10 1 to 2 De-multiplexer IC and I/O signal states

❑ De-Multiplexer Output Control

- ❖ When one output line is selected by the output selector control (S),
- ❖ → The other output line remains in the High Impedance state.

Input		Output	
S	I	Y0	Y1
0	0	0	Z
0	1	1	Z
1	0	Z	0
1	1	Z	1

Z: High Impedance

COMBINATIONAL CIRCUIT

Take this as a self study / assignment Topic

SEQUENTIAL CIRCUIT

Take this also as assignment/ self study topic

ELECTRONIC DESIGN AUTOMATION (EDA) TOOLS

❑ Early PCB Design Process:

- ❖ Built using transistors and vacuum tube technologies.
- ❖ Designers manually built the PCB with their hands, oil paper, pencil, pen, ruler and copper plates.
- ❖ The process of building a PCB was a tedious and time consuming process in ancient times where the designers sketch the required connections using pen, pencil and ruler on papers and the finished sketch was used for etching the connections on a copper plate and it took weeks and months to finish a PCB.
- ❖ Tedious and time-consuming process (weeks to months).
- ❖ Higher complexity in interconnections made the process more difficult.
- ❖ Accuracy depended on artistic skills of the designer.

❑ **Advancements in PCB Design:**

- ❖ Shift from manual design to automation with technological progress.
- ❖ Introduction of Electronic Design Automation (EDA) tools.
- ❖ Automated routing and layout within seconds.

❑ **Electronic Design Automation (EDA) Tools:**

- ❖ A set of Computer-Aided Design/Manufacturing (CAD/CAM) software.
- ❖ Used for designing and manufacturing electronic hardware like integrated circuits and printed circuit boards.
- ❖ Supports different operating systems (Windows, UNIX, Linux).

❑ **Key Players in the EDA Tool Industry:**

- ❖ Cadence, Protel, Altium, Cadsoft, Zuken, Mentor, etc.
- ❖ Popular PCB design software: OrCAD, Cadstar, Protel, Eagle, etc.

Cadence OrCAD – A Flexible & User-Friendly Tool

□ OrCAD as a Reference Tool:

- ❖ Considered flexible and user-friendly for hardware design.
- ❖ Used as the reference tool throughout this book.

□ Evaluation Copy Availability:

- ❖ Free demo version available for Windows OS.
- ❖ Downloadable from the Cadence website: <http://www.orcad.com>.

□ Usage Limitations:

- ❖ Demo version with limited features.
- ❖ Not suitable for commercial design.

EMBEDDED FIRMWARE DESIGN

□ Introduction to Embedded Firmware

- ❖ Responsible for controlling embedded hardware peripherals
- ❖ Generates responses based on functional requirements
- ❖ Acts as the master brain of the embedded system

□ Intelligence in Embedded Systems

- ❖ Intelligence is imparted to the embedded product, by embedding the firmware in the hardware.
- ❖ Intelligence is imparted once and can happen at any stage
- ❖ Embedded after fabrication or at a later stage
- ❖ Once embedded, the product functions continuously unless hardware failure or firmware corruption occurs

❑ **Handling Hardware & Firmware Failures**

- ❖ Hardware breakdown → Requires component replacement
- ❖ Firmware corruption → Needs firmware reloading
- ❖ Ensures continued proper functioning

❑ **Permanent vs. Adaptive Firmware Storage**

- ❖ Most firmware is stored in ROM → Non-alterable by end users
- ❖ Some adaptive products use configurable parameters
- ❖ Configurable parameters stored in NVRAM/FLASH

❑ **Adaptability in Control & Instrumentation**

- ❖ Parameters update based on deviation from expected behavior
- ❖ Firmware uses updated parameters for improved responses
- ❖ Enables dynamic adaptability in embedded products

❑ **Designing embedded firmware** requires an *understanding* of

- ❖ the particular embedded product hardware, like various component interfacing, memory map details, I/O port details, configuration and register details of various hardware chips used
- ❖ and some programming language (either target processor/controller specific low level assembly language or a high level language like C/C++/JAVA).

❑ **Embedded firmware development process** starts with

- ❖ the conversion of the **firmware requirements** into a **program model** using modelling tools like Unified Modelling Language (UML) diagrams or flow chart based representation.
- ❖ The UML diagrams or flow chart gives a diagrammatic representation of the decision items to be taken and the tasks to be performed.
- ❖ Once the program model is created, the next step is the implementation of the tasks and actions by capturing the model using a language which is understandable by the target processor/controller.

.

EMBEDDED FIRMWARE DESIGN APPROACHES

- ❑ The **firmware design approaches** for embedded product is purely dependent on the *complexity of the functions to be performed, the speed of operation required, etc.*
- ❑ Two basic approaches are used for Embedded firmware design.
- ❑ They are
 1. **‘Conventional Procedural Based Firmware Design’ (super loop approach)** and
 2. **‘Embedded Operating System (OS) Based Design’.**
 - ❖ The conventional procedural based design is also known as **‘Super Loop Model’**

1. THE SUPER LOOP BASED APPROACH

- ❑ The Super Loop based firmware development approach is adopted for applications
 - ❖ that are *not time critical* and
 - ❖ where the *response time* is *not* so *important* (embedded systems where missing deadlines are acceptable).
- ❑ It is very similar to a conventional procedural programming
 - ❖ where the code is executed task by task.
 - ❖ The task listed at the top of the program code is executed first and the tasks just below the top are executed after completing the first task.
 - ❖ This is a true procedural one.
- ❑ In a multiple task based system, each task is executed in serial in this approach

THE FIRMWARE EXECUTION FLOW FOR SUPERLOOP BASED APPROACH

- 1.** *Configure the common parameters and perform initialisation* for various hardware components memory, registers, etc
- 2.** *Start the first task and execute it*
- 3.** *Execute the second task*
- 4.** *Execute the next task*
- 5.** *:*
- 6.** *:*
- 7.** *Execute the last defined task*
- 8.** *Jump back to the first task and follow the same flow*

-
- ❑ From the firmware execution sequence, it is obvious that the *order in which the tasks to be executed are fixed* and they are *hard coded* in the code itself.
 - ❑ Also the operation is an *infinite loop* based approach.
 - ❑ We can *visualise* the operational sequence listed above in terms of a 'C' program code as

```
void main ()
{
  Configurations ();
  Initialisations ();
  while (1)
  {
    Task 1 ();
    Task 2 ();
    :
    :
    Task n ();
  }
}
```

-
- ❑ Almost all tasks in embedded applications are non-ending and are repeated infinitely throughout the operation.
 - ❑ From the above 'C' code you can see that the tasks 1 to n are performed one after another and when the last task (nth task) is executed, the firmware execution is again re-directed to Task 1 and it is repeated forever in the loop.
 - ❑ This repetition is achieved by using an *infinite loop*.
 - ❑ Here the **while (1) { } loop**.
 - ❑ This approach is also referred as '**Super loop based Approach**'.

```
void main ()
{
  Configurations ();
  Initialisations ();
  while (1)
  {
    Task 1 ();
    Task 2 ();
    :
    :
    Task n ();
  }
}
```

-
- Since the tasks are running inside an infinite loop, the only way to come out of the loop is
- ❖ either a **hardware reset** or an **interrupt assertion**.
 - ❖ A **hardware reset** brings the program execution back to the main loop.
 - ❖ Whereas an **interrupt request** suspends the task execution temporarily and performs the corresponding **interrupt routine**
 - ❖ and on completion of the interrupt routine, *it restarts the task execution* from the point where *it got interrupted*.

```
void main ()
{
  Configurations ();
  Initialisations ();
  while (1)
  {
    Task 1 ();
    Task 2 ();
    :
    :
    Task n ();
  }
}
```

-
- ❑ The '*Super loop based design*' doesn't require an **operating system**,
 - ❖ since there is no need for scheduling which task is to be executed and assigning priority to each task.
 - ❑ In a super loop based design,
 - ❖ the priorities are fixed and
 - ❖ the order in which the tasks to be executed are also fixed.
 - ❖ Hence the code for performing these tasks will be residing in the *code memory* without an operating system image.

-
- ❑ ***Superloop-based design*** is deployed
 - ❖ in low-cost embedded products and
 - ❖ products where response time is not time critical.
 - ❖ Some embedded products demands this type of approach if some tasks itself are sequential.

 - ❑ **For example**, reading/writing data to and from a card using a card reader requires a sequence of operations like
 - ❖ checking the presence of card, authenticating the operation, reading/writing, etc.
 - ❖ it should strictly follow a specified sequence and the combination of these series of tasks constitutes a single task-namely data read/ write.
 - ❖ There is no use in putting the sub-tasks into independent tasks and running them parallel.
 - ❖ It won't work at all

❑ **Example of a Super Loop-Based Product**

- ❖ Electronic video game toy with a keypad and display unit
- ❖ Program reads user inputs and updates the graphic display accordingly

❑ **Execution of the Super Loop Model**

- ❖ Keyboard scanning and display updating occur at a high rate
- ❖ If a key press is missed, it's not a critical issue, only a firmware bug

❑ **Why Avoid an OS in Low-Cost Products?**

- ❖ Embedding an OS is uneconomical for simple, low-cost devices
- ❖ Wasteful if real-time response requirements are not crucial

❑ **Super Loop-Based Design - Simplicity & Efficiency**

- ❖ Simple and straightforward firmware design approach
- ❖ No OS-related overheads, making it ideal for basic embedded systems

❑ **Drawbacks of Super Loop-Based Design**

- ❖ Single task failure can affect the entire system
- ❖ If a task hangs, the system may remain stuck indefinitely
- ❖ Leads to product malfunction and halted operation

❑ **Remedial Measures - Watchdog Timers (WDTs)**

- ❖ Hardware & software WDTs prevent indefinite hangs
- ❖ Helps exit the loop in case of unexpected failures
- ❖ Allows system recovery when the processor stops responding

❑ **Impact of Using Watchdog Timers**

- ❖ Improves reliability of Super Loop-based firmware
- ❖ Adds additional hardware cost
- ❖ Introduces firmware overheads for handling failures

❑ **Another important Drawback of Super Loop Design**

❑ **Lack of Real Timeliness**

- ❖ As the number of tasks increases, task repetition time also increases
- ❖ Leads to a higher probability of missing important events
- ❖ Example: *Keypad-based system may not detect key press immediately*

❑ **Keypad Monitoring Issue in Super Loop Design**

- ❖ Key monitoring task runs at fixed intervals
- ❖ User may need to press keys for a longer time to ensure detection
- ❖ Keypress event may not be in sync with the keypad status monitoring task

❑ **Corrective Measures for Real-Time Response**

- ❖ Use of interrupts for external events requiring real-time attention
- ❖ Interrupts allow immediate response to crucial tasks
- ❖ Reduces missed events and enhances system performance

❑ **Advances in Processor Technology**

- ❖ Low-cost, high-speed processors/controllers improve response time
- ❖ Helps to service multiple tasks faster
- ❖ Provides near real-time attention in Super Loop-based systems

THE EMBEDDED OPERATING SYSTEM (OS) BASED APPROACH

- ❑ The *Operating System (OS) based approach* contains operating systems, which can be either a *General Purpose Operating System (GPOS)* or a *Real Time Operating System (RTOS)* to host the user written application firmware.

GPOS based Design

- ❑ The General Purpose OS (GPOS) based design is very similar to a conventional PC based application development where the device contains an operating system (Windows/Unix/ Linux, etc. for Desktop PCs) and you will be creating and running *user applications* on top of it.
- ❑ Example of a GPOS used in embedded product development is Microsoft® Windows Embedded 8.1 which offers customisation to use with a range of industry devices like *Handhelds, Point of Sale Terminals, Patient Monitoring Systems*, etc

-
- ❑ Use of GPOS in embedded products merges the demarcation of Embedded Systems and general computing systems in terms of OS.
 - ❑ For Developing applications on top of the OS, the OS supported APIs are used.
 - ❑ Similar to the different hardware specific drivers, OS based applications also require '**Driver software**' for different hardware present on the board to communicate with them.

RTOS BASED DESIGN

- ❑ Real Time Operating System (RTOS) based design approach is employed in embedded products demanding **Real-time response**.
- ❑ RTOS respond in a timely and predictable manner to events
- ❑ **Core Components of RTOS**
 - ❖ *Real-Time Kernel* – Manages multitasking and task execution
 - ❖ *Scheduler* – Determines task priorities and execution order
 - ❖ **Threads** – Allows multiple tasks to run independently
- ❑ **Benefits of RTOS-Based Design**
 - ❖ Enables flexible scheduling of system resources (CPU, memory)
 - ❖ Provides task communication mechanisms for efficient operation
 - ❖ Suitable for critical applications with strict timing requirements

❑ **Examples of Popular RTOS for Embedded Product Development**

- ❖ Windows Embedded Compact
- ❖ pSOS, VxWorks, ThreadX
- ❖ MicroC/OS-III, Embedded Linux, Symbian

EMBEDDED FIRMWARE DEVELOPMENT LANGUAGES

- ❑ For embedded firmware development you can use either a target processor/controller specific language (Generally known as **Assembly language** or **low level language**) or a target processor/controller independent language (Like C, C++, JAVA, etc. commonly known as **High Level Language**)
- ❑ or we can use a or a combination of Assembly and High level Language.

ASSEMBLY LANGUAGE BASED DEVELOPMENT

□ Introduction to Assembly & Machine Language

- ❖ *Assembly language* is a *human-readable* notation of *machine language*
- ❖ Machine language is directly understood by the processor (*processor understandable language*)
- ❖ Processors deal only with binary code (1s and 0s)

□ Machine Language & Mnemonics

- ❖ Machine language consists of binary instructions
- ❖ Machine language is made readable by using specific symbols called ‘**mnemonics**’
- ❖ Machine language acts as an **interface between** the processor and programmer

❑ Processor Dependence in Assembly & Machine Language

- ❖ Assembly language and machine language are processor/controller dependent
- ❖ An assembly program written for one processor family will not work with others
- ❖ Requires adaptation for different processor architectures

❑ *Assembly language programming is the task of writing processor-specific machine code in mnemonic form, converting the mnemonics into actual processor instructions (machine language) and associated data using an assembler.*

❑ **Assembly Language Instruction Format**

- ❖ Instruction format consists of Opcode followed by Operands
- ❖ Opcode specifies the action for the processor/controller
- ❖ Operands provide necessary data for execution

❑ **Understanding Opcode & Operands**

- ❖ Not all Opcodes require Operands
- ❖ Some Opcodes implicitly contain operands, making them self-sufficient
- ❖ Operands can be single, dual, or multiple, depending on the instruction

❑ We will analyse each of them with the **8051 ASM** instructions as an example

□ Example 1

□ MOV A, #30

❖ This instruction mnemonic moves decimal value 30 to the 8051 Accumulator register.

❖ Here MOV A is the Opcode and 30 is the operand (single operand)

□ The same instruction when written in machine language will look like

❖ 01110100 00011110

❖ where the first 8 bit binary value 01110100 represents the opcode MOV A

❖ and the second 8 bit binary value 00011110 represents the operand 30

❑ Example 2

- ❖ The mnemonic **INC A** is an example for instruction holding operand implicitly in the Opcode
- ❖ The machine language representation for this is 00000100.
- ❖ This instruction **increments** the 8051 Accumulator register content by **1**

❑ The mnemonic *MOV A, #30* is an example for single operand instruction

❑ *LJMP 16bit address* is an example for dual operand instruction.

❑ The machine language for *LJMP 16bit address* is

❖ 00000010 addr_bit15 to addr_bit 8 addr_bit7 to addr_bit 0

❖ The first binary data is the representation of the LJMP machine code.

❖ The first operand that immediately follows the opcode represents the bits 8 to 15 of the 16bit address to which the jump is required and the second operand represents the bits 0 to 7 of the address to which the jump is targeted.

-
- ❑ Assembly language instructions are written one per line.
 - ❑ A machine code program thus consists of a sequence of assembly language instructions, where each statement contains a mnemonic (Opcode + Operand).
 - ❑ Each line of an assembly language program is split into four fields as given below

❖ **LABEL** **OPCODE** **OPERAND** **COMMENTS**

❑ LABEL is an optional field.

❑ A 'LABEL' is an identifier used extensively in programs to reduce the reliance on programmers for remembering where data or code is located.

- ❖ LABEL is commonly used for representing A memory location, address of a program, sub-routine, code portion, etc.

- ❖ The maximum length of a label differs between assemblers.

- ❖ Assemblers insist strict formats for labelling.

- ❖ Labels are always suffixed by a colon and begin with a valid character.

- ❖ Labels can contain number from 0 to 9 and special character _ (underscore).

- ❖ Labels are used for representing subroutine names and jump locations in Assembly language programming.

- ❖ It is to be noted that 'LABEL' is not a mandatory field; it is optional only.

SAMPLE 8051 CODE

❑ Structure of an Assembly Program

- ❖ The main routine starts at address 0000H
- ❖ May contain subroutines to modularize tasks
- ❖ Subroutines are invoked from the main routine using assembly instructions

❑ Calling a Subroutine - Example

- ❖ LCALL DELAY → Calls the subroutine labeled DELAY
- ❖ Transfers program execution to the memory address referenced by LABEL DELAY
- ❖ Helps organize repeatable tasks efficiently

```
#####  
; SUBROUTINE FOR GENERATING DELAY  
; DELAY PARAMETR PASSED THROUGH REGISTER R1  
; RETURN VALUE NONE  
; REGISTERS USED: R0, R1  
#####  
DELAY: MOV     R0, #255    ; Load Register R0 with 255  
       DJNZ   R1, DELAY   ; Decrement R1 and loop till  
                               ; R1= 0  
       RET                ; Return to calling program
```

❑ Best Practices for Subroutine Documentation

- ❖ Always include comments before defining a subroutine

❑ Comments should specify

- ❖ the Purpose of the subroutine
- ❖ Input parameters and how they are passed
- ❖ Return values and how they are sent back to the calling function

❑ Assembly Language Coding Rules

- ❖ Use ‘;’ for comments → Helps in code readability
- ❖ Each instruction should be on a separate line
- ❖ Unlike high-level languages (C, Java, etc.), multiple assembly instructions cannot be written on the same line

❑ In the above example the LABEL DELAY represents the reference to the start of the subroutine DELAY.

❑ You can directly replace this LABEL by putting the desired address first and then writing the Assembly code for the routine as given below.

```
ORG 0100H
    MOV R0, #255    ; Load Register R0 with 50H
    DJNZ R1, 0100H ; Decrement R1 and loop till R1= 0
    RET            ; Return to calling program
```

- ❑ **ORG** directive in 8051 Assembly Language Programming (ALP) is used to set the starting address for the program or data in memory.
- ❑ It defines the origin of the code or data segment.

-
- ❑ The advantage of using a label is that the required address is calculated by the assembler at the time of assembling the program and it replaces the Label.
 - ❑ Hence even if you add some code above the LABEL 'DELAY' at a later stage, it won't create any issues like code overlapping,
 - ❑ whereas in the second method where you are implicitly telling the assembler that this subroutine should start at the specified address (in the above example 0100H).
 - ❑ If the code written above this subroutine itself is crossing the 0100H mark of the program memory, it will be over written by the subroutine code and it will generate unexpected results

-
- ❑ Hence for safety don't assign any address by yourself, let us refer the required address by using labels and let the assembler handle the responsibility for finding out the address where the code can be placed.
 - ❑ In the above example you can find out that the label DELAY is used for calling the subroutine as well as looping (using jumping instruction based on decision-DJNZ).
 - ❑ You can also use the normal jump instruction to jump to the label by calling LJMP DELAY.

-
- ❑ The statement `ORG 0100H` in the above example is not an assembly language instruction; it is an *assembler directive instruction*.
 - ❑ It tells the assembler that the Instructions from here onward should be placed at location starting from `0100H`.
 - ❑ The Assembler directive instructions are known as ‘pseudo-ops’.
 - ❑ They are used for
 1. Determining the start address of the program (e.g. `ORG 0000H`)
 2. Determining the entry address of the program (e.g. `ORG 0100H`)
 3. Reserving memory for data variables, arrays and structures (e.g. `var EQU 70H`)
 4. Initialising variable values (e.g. `val DATA 12H`)
 - ❑ The **EQU** directive is used for allocating memory to a variable and **DATA** directive is used for initialising a variable with data.
 - ❑ No machine codes are generated for the ‘pseudo-ops’.

❑ **Writing & Saving Assembly Code**

- ❖ Assembly programs are saved as .asm, .src, or tool-specific formats.
- ❖ Can be written using Notepad, WordPad, or IDE-provided text editors.

❑ **Modular Programming in Assembly**

- ❖ Similar to C programming, multiple source files (modules) can be used.
- ❖ Each module is saved as .asm, .src, or assembler-specific formats.
- ❖ Used for large or complex programs to improve reusability and debugging.

❑ **Benefits of Modular Programming**

- ❖ Divides complex code into manageable submodules.
- ❖ Enhances reusability, debugging, and modification.
- ❖ Simplifies coding and maintenance.

❑ **Assembly to Machine Code Conversion**

- ❖ Assembly language is converted into machine code through a sequence of operations.
- ❖ That will be explained in the next slide

- ❑ Assembly language to Machine language Conversion is done through the following sequence of operation
- ❑ Source File to Object File Translation
- ❑ Library File Creation and Usage
- ❑ Linker and Locator
- ❑ Object to Hex File Converter

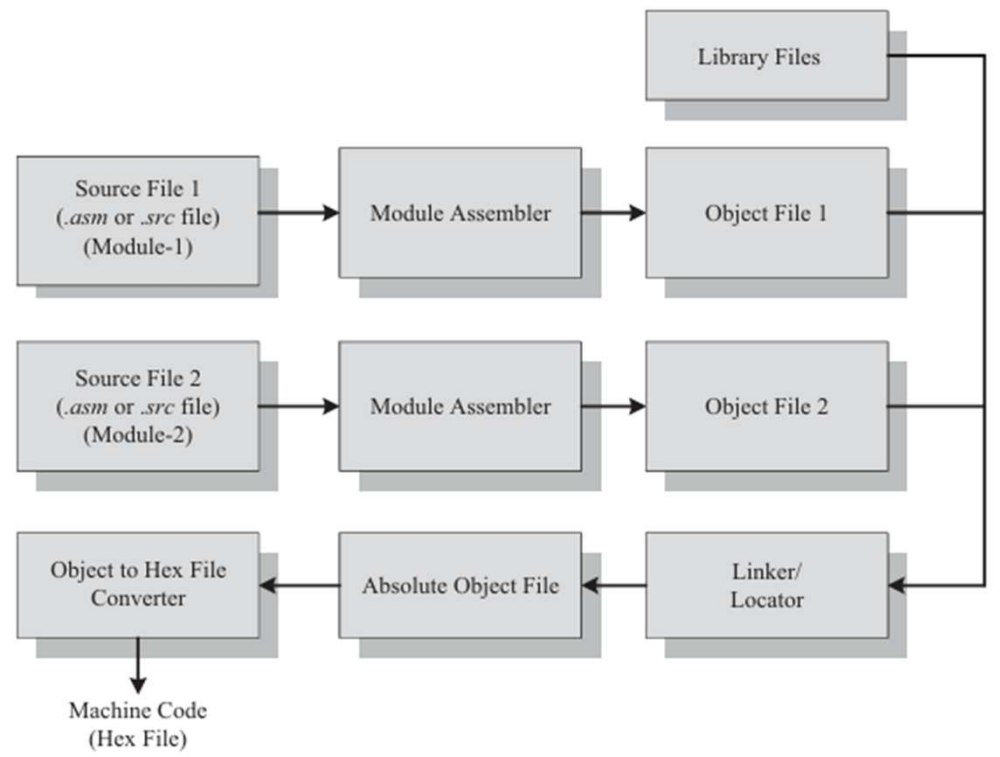


Fig. 9.1 Assembly language to machine language conversion process

SOURCE FILE TO OBJECT FILE TRANSLATION

- ❑ Translation of assembly code to machine code is performed by assembler.
- ❑ The assemblers for different target machines are different and it is common that assemblers from multiple vendors are available in the market for the same target machines.
- ❑ Some target processor's/controller's assembler may be proprietary and is supplied by a single vendor only.
- ❑ Some assemblers are freely available in the internet for downloading.
- ❑ Some assemblers are commercial and requires licence from the vendor.
- ❑ A51 Macro Assembler from Keil software is a popular assembler for the 8051 family microcontroller.

❑ Assembly Source Modules

- ❖ Each source module is written in Assembly language.
- ❖ Stored as .src or .asm files.

❑ Assembly Process

- ❖ Each file is assembled separately.
- ❖ Helps detect syntax errors and incorrect instructions.
- ❖ Successful assembly creates a corresponding object file (.obj extension).

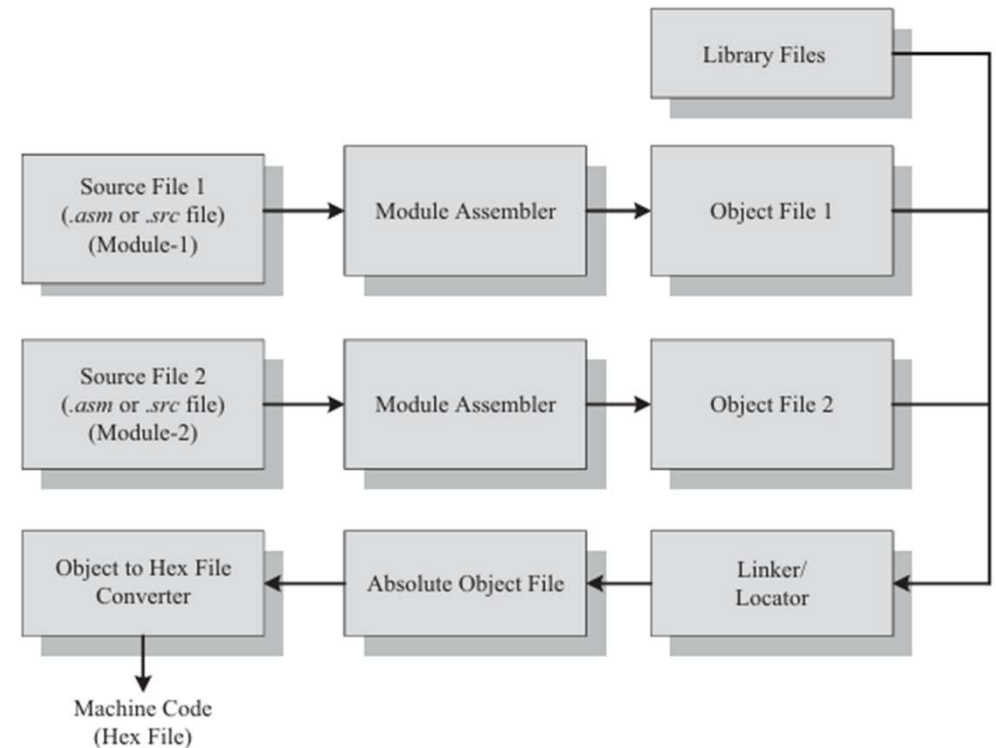


Fig. 9.1 Assembly language to machine language conversion process

❑ Object Files

- ❖ .obj file is a re-locatable segment (does not contain absolute memory address).
- ❖ Can be placed at any code memory location.
- ❖ Linker/locator assigns the absolute address during linking.

❑ Module Interaction

- ❖ Modules can share variables and subroutines (functions).
- ❖ To make a variable/function accessible to other modules, it is declared as **PUBLIC** in the source module.

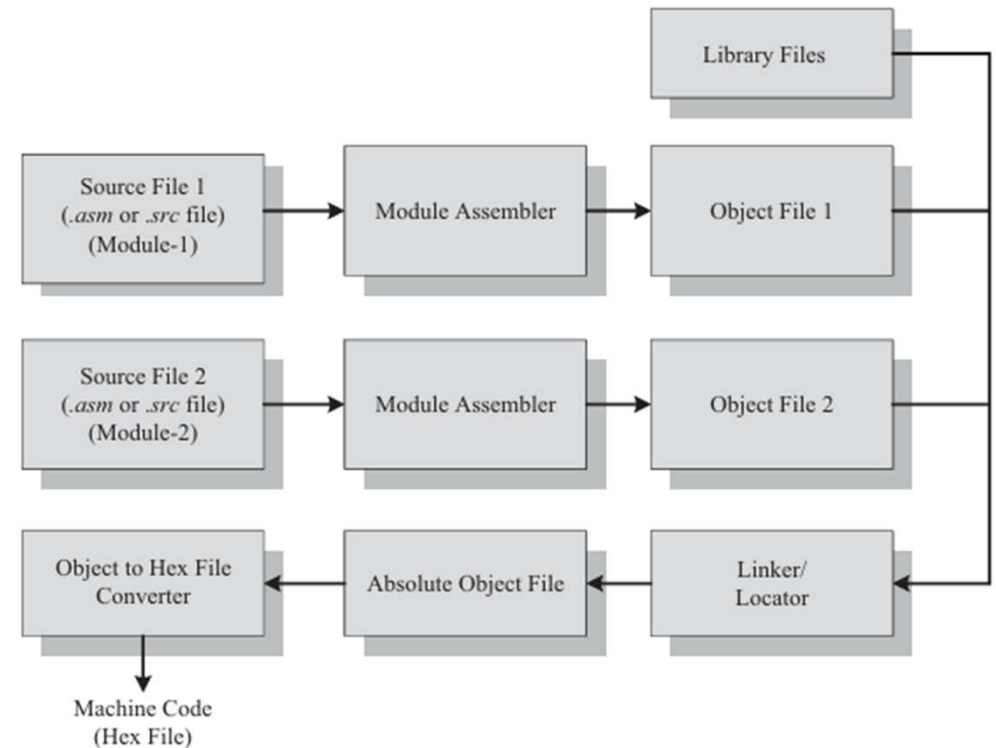


Fig. 9.1 Assembly language to machine language conversion process

❑ Importing Variables & Functions in Assembly

- ❖ **EXTRN (EXTERN)** declaration is used to import variables or functions **from** other modules.
- ❖ The **PUBLIC** keyword is used to export variables or functions **to** other modules.

❑ Assembler Behavior

- ❖ The **EXTRN** keyword informs the assembler that a variable or function comes from an **external module**.
- ❖ The assembler proceeds with assembly *without errors*, even if it does not find the definition of the variable or function.

❑ Module Interactions

- ❖ **PUBLIC** variables/functions can be accessed by one or more modules using **EXTRN**.
- ❖ There must be only one module that exports a given variable/function using **PUBLIC**.
- ❖ If multiple modules export the same variable/function using **PUBLIC**, it will lead to linker errors.

ILLUSTRATIVE EXAMPLE – A51 ASSEMBLER

❑ Target application (Simulator) contains three modules:

- ❖ ASAMPLE1.A51
- ❖ ASAMPLE2.A51
- ❖ ASAMPLE3.A51

❑- The .A51 file extension is specific to the A51 assembler.

❑ Issue with **PUBLIC** Declaration

- ❖ ASAMPLE2.A51 and ASAMPLE3.A51 both define a function named **PUTCHAR**.
- ❖ Both modules export **PUTCHAR** by declaring it as **PUBLIC**.
- ❖ The linker detects multiple **PUBLIC** definitions for the same function.

```
Build target 'Simulator'  
assembling ASAMPLE1.A51...  
assembling ASAMPLE2.A51...  
assembling ASAMPLE3.A51...  
linking...  
*** ERROR L104: MULTIPLE PUBLIC DEFINITIONS  
SYMBOL: PUTCHAR  
MODULE: ASAMPLE3.obj (CHAR_IO)
```

❑ Linker Error

- ❖ The linker encounters conflicting **PUBLIC** declarations of **PUTCHAR**.
- ❖ This causes the error: **‘MULTIPLE PUBLIC DEFINITIONS’**.
- ❖ Only one module should export a function using **PUBLIC** to avoid this error.

```
Build target 'Simulator'  
assembling ASAMPLE1.A51...  
assembling ASAMPLE2.A51...  
assembling ASAMPLE3.A51...  
linking...  
*** ERROR L104: MULTIPLE PUBLIC DEFINITIONS  
SYMBOL:  PUTCHAR  
MODULE:  ASAMPLE3.obj (CHAR_IO)
```

-
- ❑ If a variable or function declared as 'EXTRN'
 - ❖ in one or two modules,
 - ❖ there should be one module defining these variables or functions and exporting them using 'PUBLIC' keyword.
 - ❑ If no modules in a project export the variables or functions which are declared as 'EXTRN' in other modules,
 - ❖ it will generate 'linker' warnings or errors depending on the error level/warning level settings of the linker.

ILLUSTRATIVE EXAMPLE: A51 ASSEMBLER (EXTRN WITHOUT EXPORTED VARIABLES)

❑ The target application (Simulator) consists of three modules:

- ❖ ASAMPLE1.A51
- ❖ ASAMPLE2.A51
- ❖ ASAMPLE3.A51
- ❖ A51 file extension is specific to the A51 assembler.

❑ **EXTRN Usage & Missing PUBLIC Declaration**

- ❖ ASAMPLE1.A51 imports **PUT_CRLF** using the **EXTRN** keyword.
- ❖ It expects **PUT_CRLF** to be exported in **ASAMPLE2.A51** or **ASAMPLE3.A51** using **PUBLIC**.
- ❖ However, none of the modules export **PUT_CRLF**.

```
*** WARNING L1: UNRESOLVED EXTERNAL SYMBOL
SYMBOL: PUT_CRLF
MODULE: ASAMPLE1.obj (SAMPLE)
```

❑ Linker Behavior

- ❖ The linker searches for the exported function but finds none.
- ❖ This results in a warning or error message: ‘UNRESOLVED EXTERNAL SYMBOL’.
- ❖ The severity of the error depends on the linker’s level settings.

```
*** WARNING L1: UNRESOLVED EXTERNAL SYMBOL  
SYMBOL: PUT CRLF  
      _  
MODULE: ASAMPLE1.obj (SAMPLE)
```

❑ Best Practices

- ❖ Ensure that every EXTRN variable or function is properly exported by one module using PUBLIC.
- ❖ Check linker settings to manage warnings/errors effectively.

LIBRARY FILE CREATION AND USAGE

❑ Libraries in Assembly Programming

- ❖ Libraries are specially formatted collections of object modules.
- ❖ Used by the linker to include necessary object modules in a program.

❑ Library File Characteristics

- ❖ Library files have the extension “.lib”.
- ❖ Only necessary modules from the library are used during linking.
- ❖ Libraries provide a form of source code hiding.

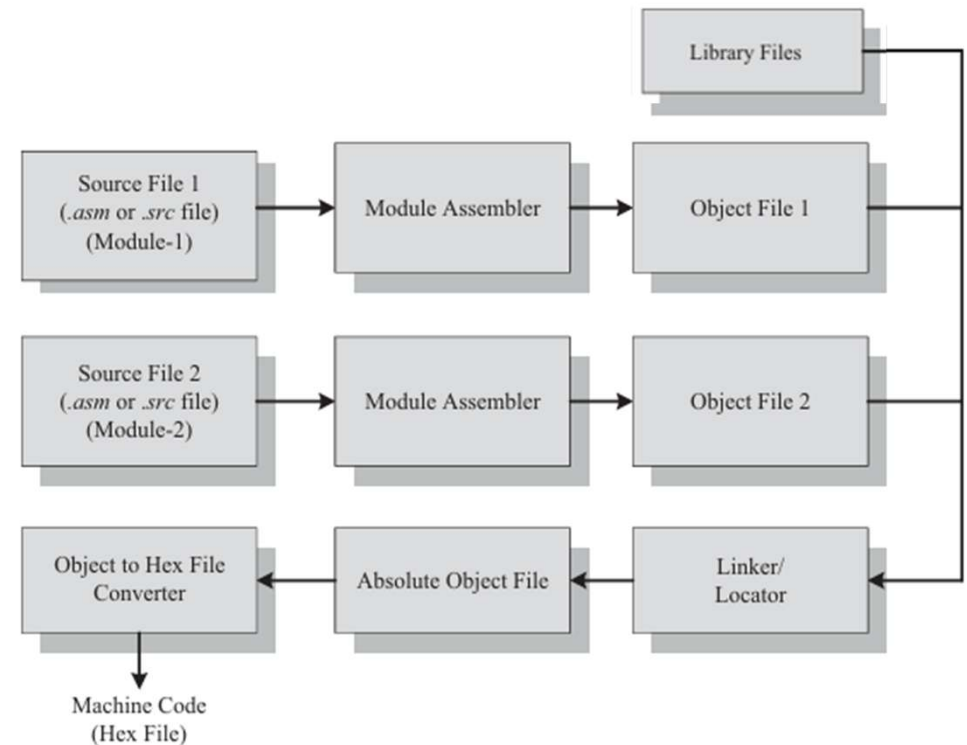


Fig. 9.1 Assembly language to machine language conversion process

❑ **Advantages of Libraries**

- ❖ Developers can share functions without revealing their source code.
- ❖ Only the function name, input/output details, and usage guidelines need to be provided.
- ❖ Useful for distributing functions to application developers.

❑ **Using a Library in a Project**

- ❖ To use a library file, add it to the project.
- ❖ The linker automatically includes required object modules from the library.

-
- ❑ **Commercial assembler/compiler suites** may include pre-written library files.
 - ❖ These libraries perform tasks like multiplication, floating point arithmetic, etc.
 - ❖ Provided as an add-on utility or a bonus by the vendor.

 - ❑ **Example – LIB51 from Keil Software**
 - ❖ LIB51 is a library creator tool from Keil Software.
 - ❖ Used for generating library files for A51 Assembler / C51 Compiler.
 - ❖ Specifically designed for 8051 microcontroller development.

 - ❑ **Benefits of Using Pre-Written Libraries**
 - ❖ Saves time and effort by utilizing existing optimized functions.
 - ❖ Encourages code reuse and enhances development efficiency.
 - ❖ Provides reliable and tested implementations for essential functionalities.

LINKER AND LOCATOR

❑ Linker & Locator in Assembly Programming

- ❖ The Linker & Locator links multiple object modules in a multi-module project.
- ❖ Assigns absolute addresses to each module.

❑ Linker Responsibilities

- ❖ Extracts object modules from libraries and assembler-generated obj files.
- ❖ Resolves external dependencies among modules.
- ❖ Links external variables and functions across modules.

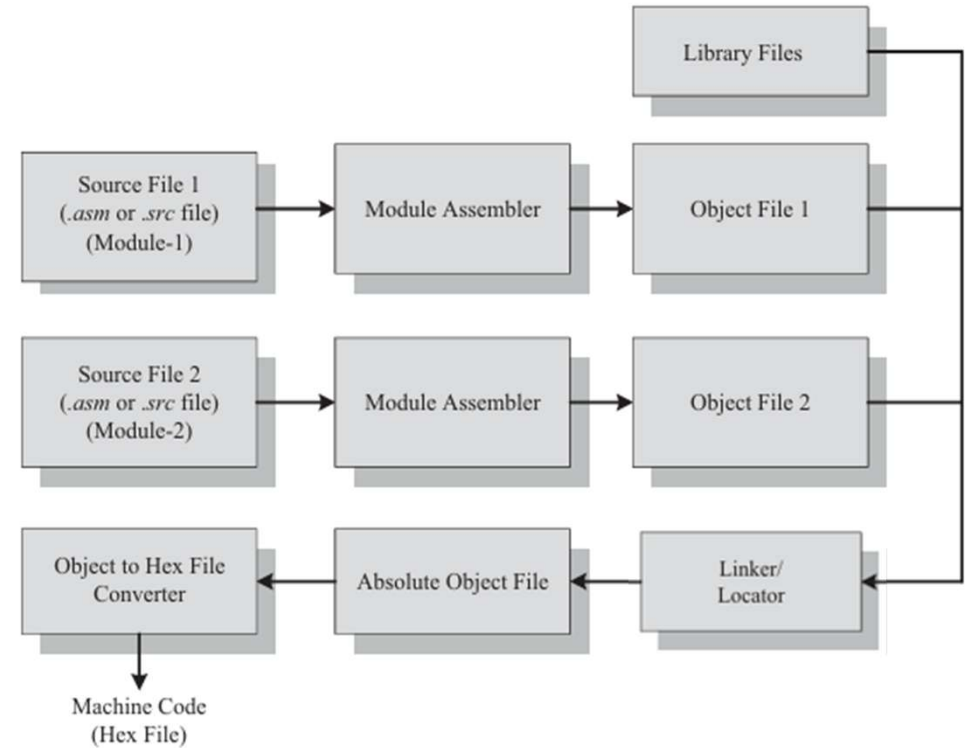


Fig. 9.1 Assembly language to machine language conversion process

❑ Absolute Object File

- ❖ Does not contain re-locatable code or data.
- ❖ All code and data reside at fixed memory locations.
- ❖ Used for creating hex files for code memory dumping in processors/controllers.

❑ Example – BL51 from Keil Software

- ❖ BL51 is a Linker & Locator tool for A51 Assembler/C51 Compiler.
- ❖ Used for 8051 microcontroller development.

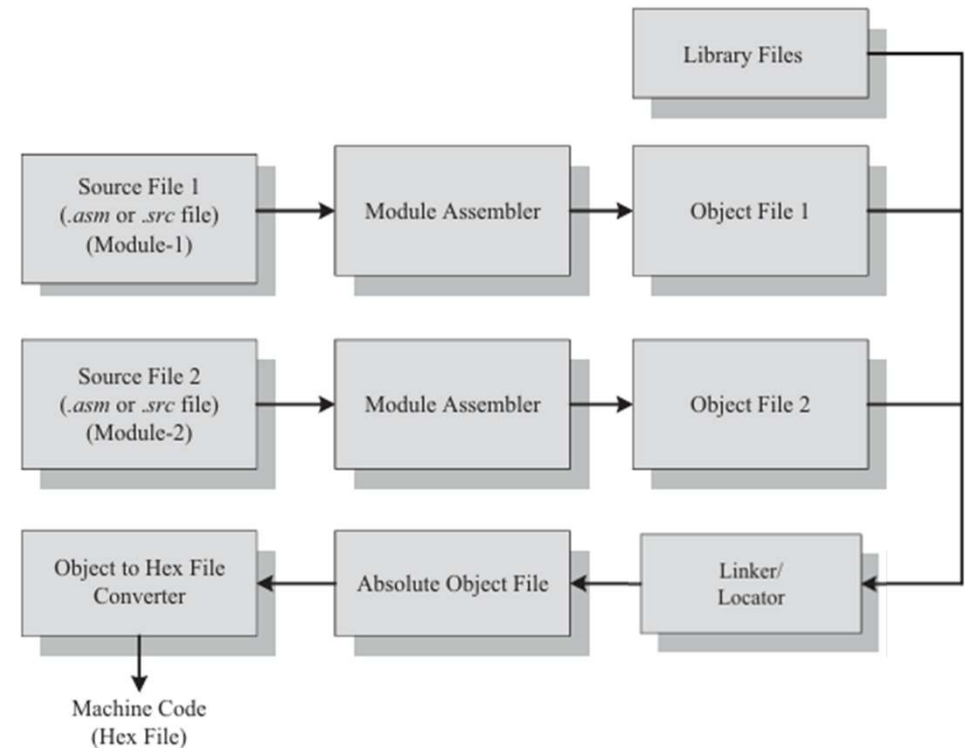


Fig. 9.1 Assembly language to machine language conversion process

OBJECT TO HEX FILE CONVERTER

❑ Hex File – Final Stage of Assembly Conversion

- ❖ Converts Assembly language (mnemonics) into machine code.
- ❖ The Hex File represents the machine code.
- ❖ The Hex File is dumped into the code memory of the processor/controller.

❑ Hex File Formats

- ❖ Hex file format varies based on the target processor/controller.
- ❖ Intel processors/controllers use Intel HEX format.
- ❖ Motorola processors/controllers use Motorola HEX format.

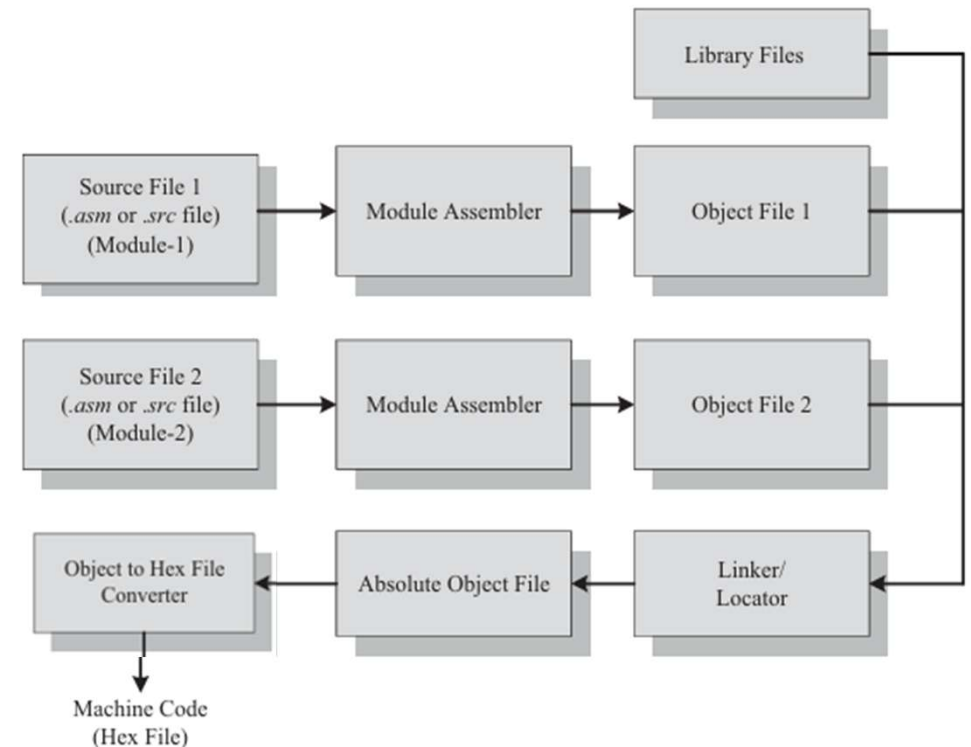


Fig. 9.1 Assembly language to machine language conversion process

Characteristics of HEX Files

- ❖ HEX files are ASCII files containing hexadecimal representation of the target application.
- ❖ Created from the final Absolute Object File.

Hex File Conversion Utility

- ❖ The Object to Hex File Converter is used to generate HEX files.
- ❖ OH51 from Keil Software is an example utility for A51 Assembler / C51 Compiler.
- ❖ Used specifically for 8051 microcontroller development

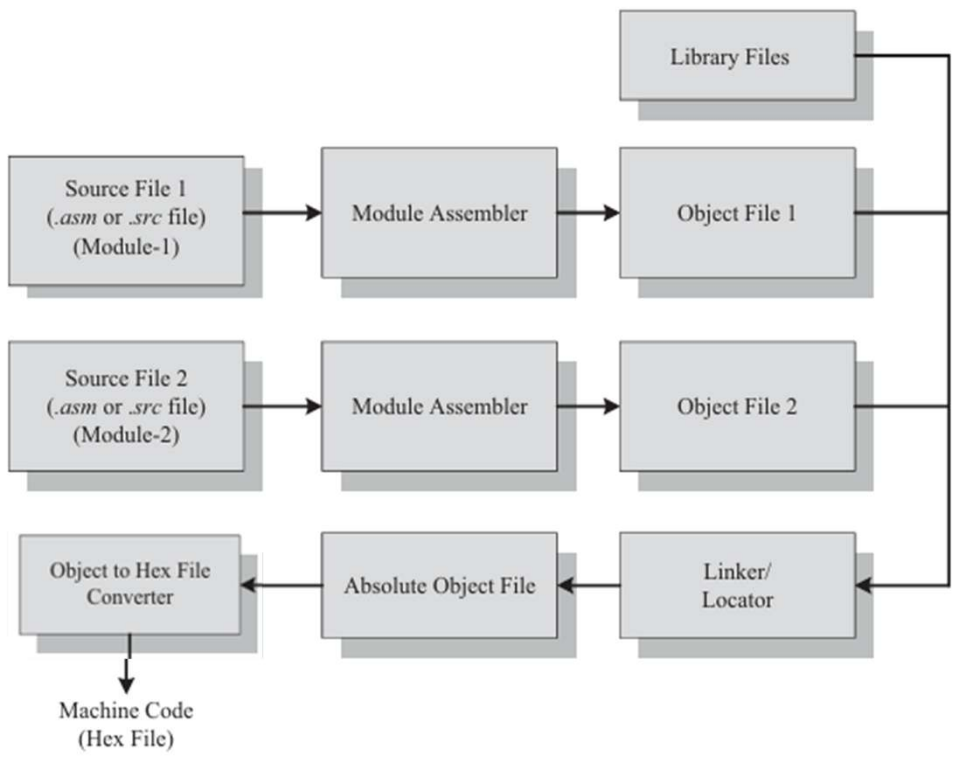


Fig. 9.1 Assembly language to machine language conversion process

❑ **Assembly Language in Embedded Development**

- ❖ Assembly Language based development was (is ☺) the most common technique adopted from the beginning of embedded technology development.
- ❖ Requires thorough understanding of:
 - Processor architecture
 - Memory organization
 - Register sets
 - Mnemonics (Assembly instructions)

❑ **Mastering Assembly Language**

- ❖ If you master one processor architecture and its assembly instructions, you can make the processor flexible

ADVANTAGES OF ASSEMBLY LANGUAGE BASED DEVELOPMENT

❑ Efficient Code Memory and Data Memory Usage (Memory Optimisation)

- ❖ Since the developer is well versed with the *target processor architecture* and *memory organisation*, optimised code can be written for performing operations
- ❖ Efficient coding ensures less utilization of **code memory** and optimized use of **data memory**.
- ❖ Memory management is a primary concern in embedded product development.

❑ High Performance

- ❖ Optimised code not only improves the code memory usage but also improves the total system performance.
- ❖ Through effective assembly coding, optimum performance can be achieved for a target application.

❑ **Low Level Hardware Access**

- ❖ Most of the code for low level programming like accessing external device specific registers from the operating system kernel, device drivers, and low level interrupt routines, etc. are making use of direct assembly coding
- ❖ since low level device-specific operation support is not commonly available with most of the high-level language cross compilers.

❑ **Code Reverse Engineering**

- ❖ Reverse engineering is the process of understanding the technology behind a product by extracting the information from a finished product.
- ❖ Reverse engineering is performed by ‘hackers’ to reveal the technology behind ‘Proprietary Products’.
- ❖ Though most of the products employ code memory protection, if it may be possible to break the memory protection and read the code memory, it can easily be converted into assembly code using a dis-assembler program for the target machine.

DRAWBACKS OF ASSEMBLY LANGUAGE BASED DEVELOPMENT

- ❑ Every technology has its own pros and cons.
- ❑ From certain technology aspects assembly language development is the most efficient technique.
- ❑ But it is having the following technical limitations also.
- ❑ **High Development Time:**
 - ❖ Assembly language is much harder to program than high level languages.
 - ❖ The developer must pay attention to more details and must have thorough knowledge of the architecture, memory organisation and register details of the target processor in use.
 - ❖ Learning the inner details of the processor and its assembly instructions is highly time consuming and it creates a delay impact in product development.
 - ❖ One probable solution for this is use a readily available developer who is well versed in the target processor architecture assembly instructions.
 - ❖ Also more lines of assembly code are required for performing an action which can be done with a single instruction in a high-level language like 'C'.

❑ Developer Dependency

- ❖ There is no standardized rules for assembly language application development, whereas all high level languages instruct certain set of rules for application development
- ❖ Developers have freedom to choose memory locations and registers.
- ❖ Programming approach varies from developer to developer based on individual preference.
- ❖ Different methods exist for performing the same task (e.g., moving data to an accumulator).
- ❖ If the approach done by a developer is not documented properly at the development stage, he/ she may not be able to recollect why this approach is followed at a later stage (Lack of documentation can make it difficult to understand code later.)
- ❖ Developer dependency increases without proper documentation.
- ❖ New developers may struggle to analyze undocumented code.
- ❖ Proper documentation reduces developer dependency.
- ❖ - Helps in code maintenance and future modification

❑ Non-Portable

- ❖ Target applications written in assembly instructions are valid only for that particular family of processors (e.g. Application written for Intel x86 family of processors) and cannot be re-used for another target processors/controllers (Say ARM Cortex M family of processors).
- ❖ If the target processor/controller changes, a complete re-writing of the application using the assembly instructions for the new target processor/controller is required.
- ❖ This is the major drawback of assembly language programming and it makes the assembly language applications non-portable.

HIGH LEVEL LANGUAGE BASED DEVELOPMENT

- ❑ Any high level language (like C, C++ or Java) with a supported cross compiler (for converting the application developed in high level language to target processor specific assembly code We will discuss cross-compilers in detail in a later section) for the target processor can be used for embedded firmware development.
- ❑ The most commonly used high level language for embedded firmware application development is 'C'.
- ❑ **why 'C' is used as the popular embedded firmware development language.**
 - ❖ The answer is "C is the well defined, easy to use high level language with extensive cross platform development tool support".
 - ❖ Nowadays Cross-compilers for C++ is also emerging out and embedded developers are making use of C++ for embedded application development.

❑ The development steps for high-level language-based embedded firmware are similar to those for assembly-based development.

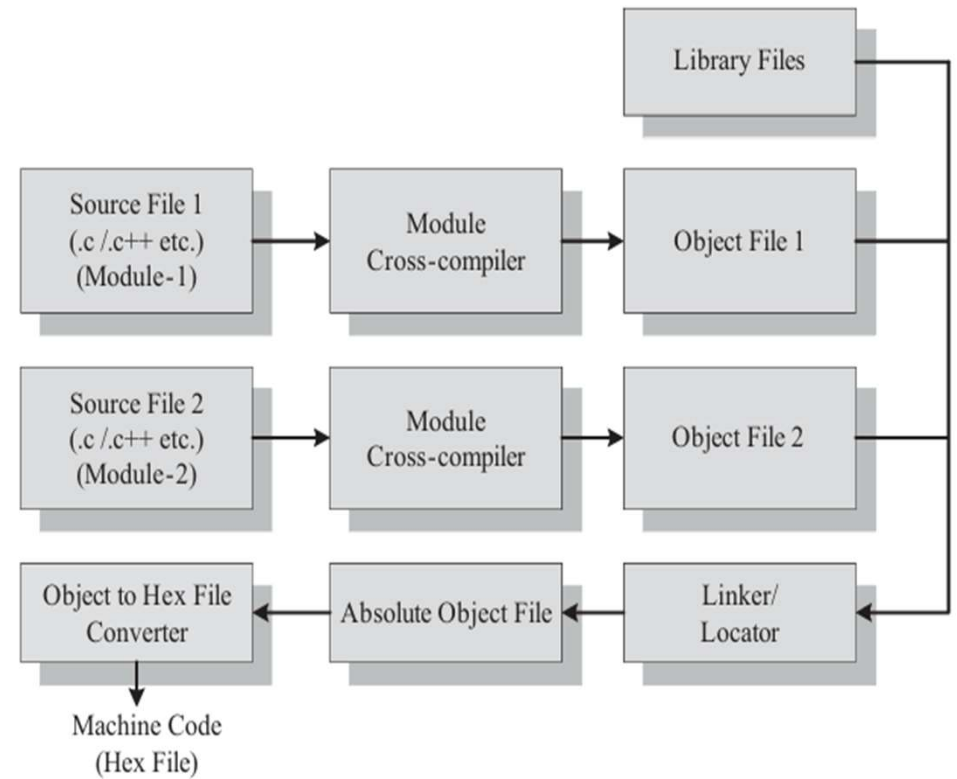
❑ The primary difference lies in the conversion of source files.

❑ Conversion Process

❖ In high-level language development, the conversion from source file to object file is performed by a **cross-compiler**.

❖ In assembly language development, this conversion is carried out by an assembler.

❑ The various steps involved in the conversion of a program written in high level language to corresponding binary file/machine language is illustrated in Fig.



HIGH-LEVEL LANGUAGE PROGRAMMING FOR EMBEDDED SYSTEMS

- ❑ **Programs written in high-level languages are saved with specific extensions:**
 - ❖ .c for C
 - ❖ .cpp for C++

- ❑ **Can be written using:**
 - ❖ Text editors like Notepad or WordPad
 - ❖ IDE-provided text editors supporting the language.

- ❑ **Modular Programming Approach**
 - ❖ Most high-level languages support modular programming.
 - ❖ Allows multiple source files (modules) written in the same language.
 - ❖ Each module's source file carries its corresponding language extension.

❑ **Cross-Compilation Process**

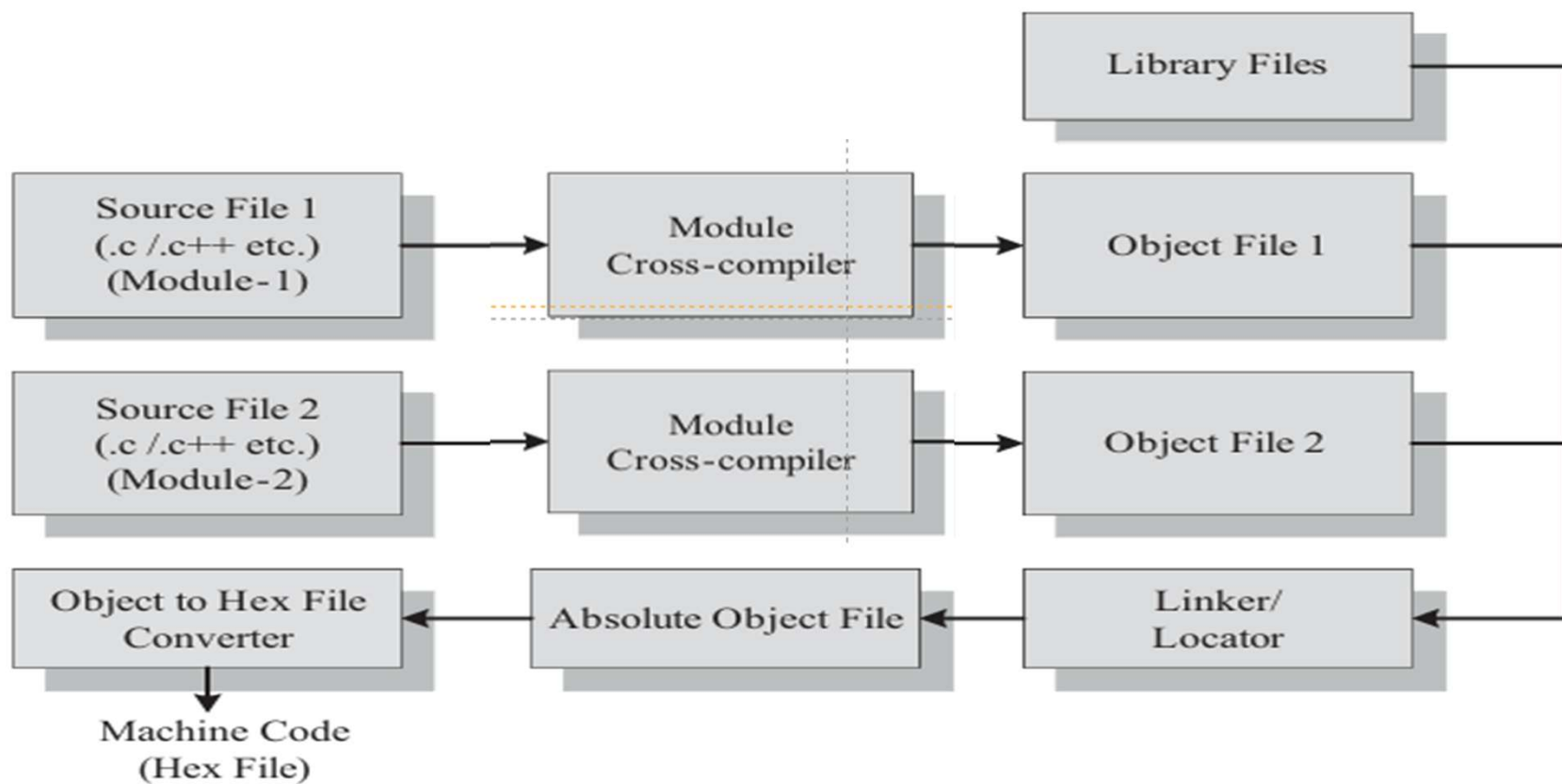
- ❖ High-level language source code is translated into executable object code using a cross-compiler.
- ❖ Different processors require different cross-compilers.
- ❖ Without a cross-compiler, a high-level language cannot be used for embedded firmware development.

❑ **Example of a Cross-Compiler**

- ❖ C51 from Keil Software is a widely-used cross-compiler for the C language.
- ❖ Specifically designed for the 8051 microcontroller family.

❑ **Conversion Process**

- ❖ Each module's source code is converted into an object file by the cross-compiler.
- ❖ The remaining steps are similar to those in assembly language-based development.



ADVANTAGES OF HIGH LEVEL LANGUAGE BASED DEVELOPMENT

❑ Reduced Development Time

- ❖ *Minimal hardware knowledge required:*
- ❖ Developers need only basic understanding of:
 - Memory organization
 - Register details of the target processor
 - Syntax of the high-level language
- ❖ Cross-Compiler Handles hardware-specific conversions, reducing the developer's need to understand assembly instructions.
- ❖ Cross compiler Accelerates development time by reducing effort in learning target machine architecture.
- ❖ High-level language development broadens accessibility beyond specialized architects.
- ❖ Enables any developer familiar with syntax and willing to learn minimal hardware details.
- ❖ High-level languages require fewer lines of code to accomplish tasks.
- ❖ Compared to assembly language, they provide simpler and more manageable development.

❑ **Developer Independency**

- ❖ The syntax used by most of the high level languages are universal and a program written in the high level language can easily be understood by a second person knowing the syntax of the language.
- ❖ Certain instructions may require little knowledge of the target hardware details like register set, memory map etc.
- ❖ Apart from these, the high level language based firmware development makes the firmware developer independent.
- ❖ High level languages always instruct certain set of rules for writing the code and commenting the piece of code.
- ❖ If the developer strictly adheres to the rules, the fi rmware will be 100% developer independent.

❑ Portability

- ❖ Target applications written in high level languages are converted to target processor/controller understandable format (machine codes) by a cross-compiler.
- ❖ An application written in high level language for a particular target processor can easily be converted to another target processor/controller specific application, with little or less effort by simply re-compiling/little code modification followed by re-compiling the application for the required target processor/controller, provided, the cross-compiler has support for the processor/controller selected.
- ❖ This makes applications written in high level language highly portable.
- ❖ Little effort may be required in the existing code to replace the target processor specific header files with new header files, register definitions with new ones, etc.
- ❖ This is the major flexibility offered by high level language based design.

LIMITATIONS OF HIGH-LEVEL LANGUAGE-BASED DEVELOPMENT

- ❑ High-level language design offers significant advantages, often outweighing its limitations.
- ❑ **Challenges with Cross-Compilers**
 - ❖ Some cross-compilers may not generate optimized machine instructions for target processors.
 - ❖ Target images from such compilers may be non-optimized in terms of performance & code size.
 - ❖ A manually hand-coded assembly implementation may achieve the same task more efficiently.
 - ❖ More instructions lead to longer execution time.
- ❑ **Modern Optimization Trends**
 - ❖ Newer cross-compilers incorporate techniques to optimize both code size and performance.

❑ **Limitations in Low-Level Hardware Access**

- ❖ High-level language code may not be efficient for accessing low-level hardware.
- ❖ Critical timing operations (nanoseconds/microseconds) may require assembly coding.

❑ **Investment Considerations**

- ❖ High-level language development tools (IDE & cross-compilers) require higher investment.
- ❖ Assembly language tools are relatively more cost-effective for firmware development.