

The other system components refer to the components/circuits/IC's which are necessary for proper functioning of the Embedded system.

The following section includes some of the essential circuits.

Reset Circuit:

→ When the system is behaving abnormally then we use RESET circuit.

→ ALSO, RESET circuit is to ensure that the device is not operating at voltage level where the device is not guaranteed to operate during power system on.

→ RESET signal brings internal register and hardware systems to initial stages & starts the firmware execution from reset vector (vector address 0x0000)

Here mainly program counter is set to 0000

→ Since the processor operation is synchronised to a clock signal, the reset pulse should be wide enough to give time for the clock oscillator to stabilise before the internal RESET state starts.

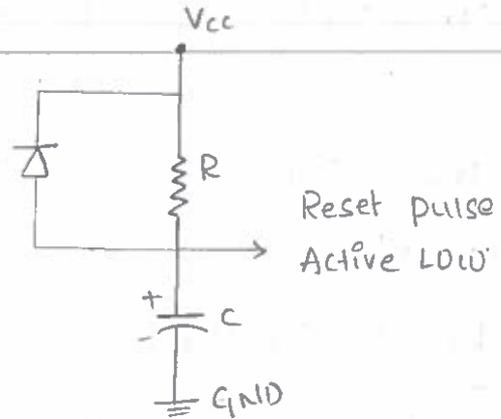
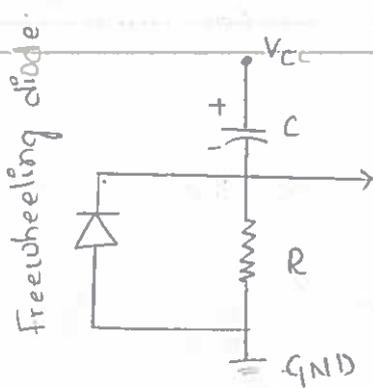
→ RESET signal can be either active high or active low.

• Active high:

The processor undergoes reset when the reset pin of the processor is at logic HIGH (logic 1)

Active low:

The processor undergoes reset when the reset pin of the processor is at logic Low (logic 0).



The RESET circuit consists of capacitor, resistor and a free wheeling diode.

→ The reset signal to the processor can be applied in two ways.

- using external passive RESET circuit or through a standard RESET IC MAX 810 from maxm Dallas.
- using built-in internal circuitry inside the MCMIP.

Brown-out protection circuit:

→ Brown-out protection circuit prevents the processor/controller from unexpected program execution behavior when the supply voltage to the processor/controller falls below a specified voltage.

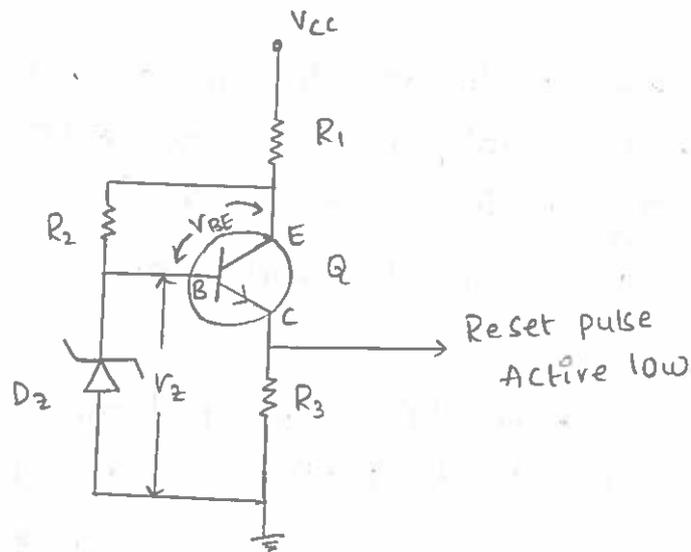
→ It is mainly essential for battery powered circuits since there are greater chances for the battery voltage to drop below the threshold value.

→ The sudden drop in the voltage may lead to situations like data corruption.

→ Brown-out protection circuit holds processor/ controller in RESET state, when the operating voltage falls below the threshold, until it raises above the threshold value. (2)

→ Same like RESET circuits it can be built-in or external passive circuit.

→ The circuit below shows implementation of Brown-out circuit protection using Zener diode.



→ Zener diode and transistor Q forms heart of the circuit.

→ Transistor conducts when $V_{CC} > V_{BE} + V_z$.

→ Transistor stops conducting when $V_{CC} < V_{BE} + V_z$.

→ Zener diode is set in such a way that there is less threshold value for V_{CC} .

→ The values of R_1 , R_2 and R_3 can be selected based on the electrical characteristics of the transistor in use.

→ The IC like DS1232 from Maxim Dallas provides Brown-out protection.

Oscillator circuit.

→ A microprocessor/micro controller is a digital device made up of digital combinational and sequential circuits.

The instruction execution of a microprocessor/micro-

-controller occurs in sync with a clock signal.

(Analogous to the heartbeat of living beings).

→ oscillator unit is responsible for generating the precise clock for the processor.

→ certain processors/controllers integrate a built in oscillator unit and simply require an external ceramic resonator/quartz crystal for clock generation.

- Quartz crystal & ceramic resonator are equivalent in operation.

- A quartz crystal normally mounted hermetically sealed metal case with two leads protruding out of the case.

→ certain devices may not contain a built in oscillator unit and require the clock pulses to be generated and supplied externally.

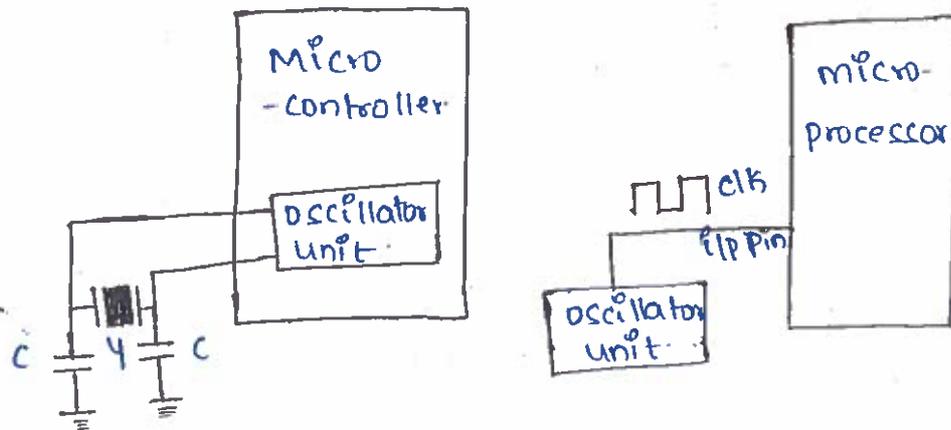
→ The logical circuits lying inside the processor always have an upper threshold value for the maximum clock at which the system can run, beyond which system becomes unstable & non-functional.

→ The power consumption increases with increase in clock frequency.

→ The accuracy of program execution depends on the accuracy of the clock signal. (also represented in ± 1 ppm).

→ The speed of operation depends on the frequency.

fig show how to illustrate the usage of quartz crystal / ceramic resonator and external oscillator chip for circuit generation. (3)



C: Capacitor.
Y: Resonator.

Real Time clock.

- Real time clock holds information like current time (in hours, minutes & seconds) in 12 hours/24 hour format, date, month, years, day of the week etc. and supplies timing reference to the system.
- RTC is intended to function even when power is absent.
- The RTC chip contains a microchip for holding the time and data related information and backup battery cell for functioning in absence of power, in a single IC package.
- For operating system based embedded devices, a timing reference is essential for synchronizing the operations of OS kernel.

→ The RTC can interrupt the OS kernel by asserting the interrupt line of the processor/controller to which RTC interrupt line is connected.

→ The OS kernel identifies the interrupt terms of the Interrupt Request (IRQ) no. generated by an interrupt controller.

→ One IRQ can be assigned to the RTC interrupt and kernel can perform necessary operations when an RTC timer tick interrupt occurs.

Watchdog Timer.

→ We have watchdog to monitor firmware execution and reset the system processor/microcontroller when the program hangs up.

→ A watchdog timer, or simply a watchdog, is a hardware timer for monitoring firmware execution.

→ Depending on the internal implementation, the watchdog timer increments or decrements a free running counter with each clock pulse and generates reset signal to reset processor.

If the count reaches zero for a down counting watchdog (or) if the count reaches the highest count for upcounting watchdog.

→ If the watchdog counter is in the enabled state, the firmware can write a zero to it before starting the execution of a piece of code and watchdog will start counting.

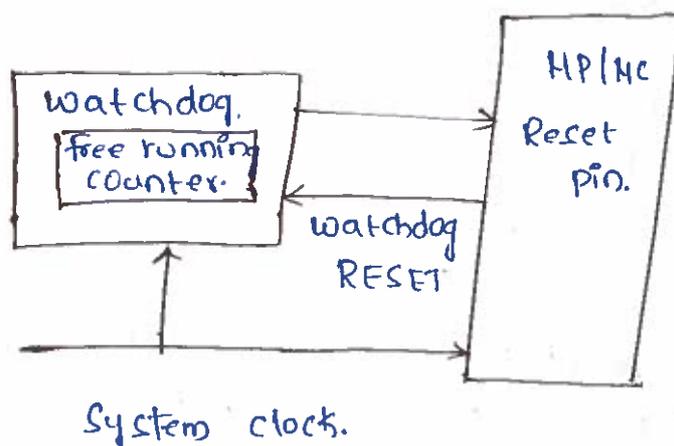
If the firmware execution doesn't complete due to malfunctioning, within the time required by the watchdog to reach the maximum count, the counter will generate reset pulse and this will reset the processor. (4)

→ If the firmware execution completes before the expiration of the watchdog timer you can reset the count by writing zero to the watchdog timer register.

→ If processor/controller doesn't have in-built watchdog timer, then same can be implemented using external watchdog timer.

It uses a hardware logic for enabling/disabling, resetting the watchdog count etc. Instead of the firmware based 'writing' to the status and watchdog timer register.

→ The implementation of external watchdog timer based MP supervisor circuit for a small scale embedded system.



Embedded Firmware

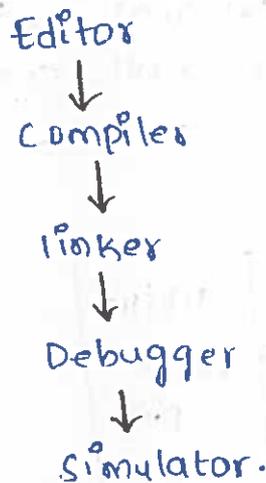
→ Firmware is a type of software which has been pre-installed from the factory itself into the non-volatile memory of the device in order to make it work properly.

→ Also, Embedded firmware refers to the control algorithm and or the configuration settings that an Embedded system developer dumps into the code.

→ There are various methods of developing the embedded firmware. They are listed below.

(i) write the program in high level languages like embedded C/C++ using an Integrated development Environment (IDE).

IDE will contain



(ii) write the source code in assembly language using the instructions supported by your applications target processor/controller.

The process of converting the source code written in either a high level language or processor/controller specific assembly code to machine readable binary code is called 'HEX file creation'.

→ Two types of control algorithm design exist in embedded firmware development. (5)

(i) The first type of control algorithm development is known as the infinite loop or 'super loop', where the control runs from top to bottom and then again jumps back to top of the program.

It is similar to the while (1) based technique in c

```
{  
}
```

(ii) Second method deals with splitting the functions to be executed into tasks & running these tasks using a scheduler which is a part of a general purpose or real-time embedded operating system.

Embedded Firmware Design Approaches.

The firmware design approaches for embedded products are 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 'conventional procedural based firmware design' and 'embedded operating system (OS) based design'. The conventional procedural based design is also known as 'super loop model'. The two embedded firmware design approaches are discussed below.

The Superloop 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 task below the top are executed after completing the first task. This is a true procedural one. It is a multiple task based system, each task is executed in serial in this approach. The firmware execution flow for this will be

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 ();
    initializations ();
    while (1)
    {
      Task1 ();
      Task2 ();
      :
      :
      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 the tasks (1) are performed one after another and when the last task (nth task) is executed, the firmware execution is again redirected to Task1 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".

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 then performs the corresponding interrupt routine and on completion of the interrupt routine it restarts the task execution from the point where it got interrupted.

The 'Superloop 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 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 the code memory without any operating system image.

This type of design is deployed in low-cost embedded products where response time is not time critical.

Some Embedded products demand this type of approach if some task 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.

The 'Super loop based design' is simple and straight forward without any OS related overheads. The major drawback of this approach is that any failure in any part of a single task will affect the total system. If the program hangs up at some point while executing a task, it will remain there forever and ultimately the product stops functioning. There are remedial measures for overcoming this use of hardware and software watch dog timers (WDTs) helps in coming out from the loop when an unexpected failure occurs or when the processor hangs up. This, in turn, may cause additional hardware cost and firmware overheads.

approach is the lack of real timeliness. If the number of tasks to be executed within an application increases, the time at which each task is repeated also increases. This brings the probability of missing out some events. For example in a system with keypad, according to the 'super loop design', there will be a task for monitoring the keypad connected I/O lines and this need to be the task running while you press the keys (that is key pressing event may not be in sync with the keypad press monitoring task within the firmware). In order to identify the key press, you may have to press the keys for a sufficiently long time till the keypad status monitoring task is executed internally by the firmware. This will really lead to the lack of real timeliness. There are corrective measures for this also. The best advised option in use interrupts for external events requiring real time attention. Advances in processor technology brings out low cost high speed processors/controllers and thereby are capable of providing a nearly real time attention to external events.

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. 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/Linux, etc. for Desktop PCs) and you will be creating and running user applications on top of it. Example of GPOS used in embedded product development is Microsoft® Windows XP Embedded.

Examples of Embedded products using microprocessors
Windows XP OS are Personal Digital Assistants
(PDAs), Hand held devices/ portable devices and point
of sale (Pos) terminals. 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 them.

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. Real Time Operating System
contains a Real time kernel responsible for performing
pre-emptive multi tasking, scheduler for scheduling
tasks, multiple threads, etc. A Real Time Operating
System (RTOS) allows flexible scheduling of system
resources like the CPU and memory and offers
some way to communicate between tasks.

'Windows CE', 'psos', 'vxworks', 'ThreadX', 'MicroClos-II',
'Embedded Linux', 'Symbian' etc are examples of
RTOS employed in embedded product development.
mobile phones, PDAs (Based on Windows CE/ Windows
mobile platforms), handheld devices etc. are examples
of 'Embedded products' based on RTOS. Most of the
mobile phones are built around the popular RTOS
System.

Assembly Language based Development.

'Assembly language' is the human readable notation of 'machine language', where as 'machine language' is a processor understandable language. Processors deal only with binaries (1s and 0s). Machine language is a binary representation and it consists of 1s and 0s. Machine language is made readable by using specific symbols called 'mnemonics'. Hence machine language can be considered as an interface between processor and programmer. Assembly language and machine languages are processor/controller dependent and an assembly program written for one processor/controller family will not work with others.

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 program was the most common type of programming adopted in the beginning of software revolution. If we look back to the history of programming, we can see that a large number of programs were written entirely in assembly language. Even in the 1990s, the majority of the console video games were written in assembly language, including most popular games written for the Sega Genesis and the Super Nintendo Entertainment System. The popular arcade game NBA Jam released in 1993 was also coded entirely using assembly language.

Even today also almost all low level programming is carried out using assembly language. Some operating system dependent tasks require low-level languages. In particular, assembly language is often used in writing the low level interaction between the operating system and the hardware, for instance in device drivers.

The general format of an assembly language instructions is an opcode followed by operands. The opcode tells the processor/controllers what to do and the operands provide the data and information required to perform the action specified by the opcode. It is not necessary that all opcode should have operands following them. Some of the opcode implicitly contains the operand and in such situation ^{no} do ~~it~~ necessary that all opcodes should have operands following them. Some of the opcode implicitly contains the operand) operand is required. The operand may be a single operand, dual operand or more. we will analyse each of them with the 8081 ASM instructions as a example.

MOV A, #30.

This instruction mnemonic moves decimal value 30 to the 8081 Accumulator register. Here MOV A is the opcode and 30 is the operand (single operand). The same instruction when written in machine language looks 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.

the instruction is an example for instruction holding operand implicitly in the opcode. The machine language representation of the same is 0000100. This instruction increments the 8051 Accumulator register content by 1. (9)

The Assembly language program written in assembly code is saved as .asm (ASSEMBLY file) file or an .src (SOURCE) file. Any text editor like 'notepad' or 'wordpad' from Microsoft® or the text editor provided by an Integrated Development (IDL) tool can be used for writing the assembly instructions.

Similar to 'C' and other high level language programming you can have multiple source files called modules in assembly language programming. Each module is represented by an '.asm' or '.src' file similar to the '.c' files in C programming. The approach is known as 'modular programming', the entire code is divided into submodules and each module is made re-usable. Modular programs are usually easy to code, debug and alter. Conversion of the assembly language to machine language is carried out by a sequence of operations.

Advantages of Assembly Based Development

Assembly language based development was the most common technique adopted from the beginning of embedded technology development. Through understanding of the processor architecture, memory organization, register sets and mnemonics is very essential for assembly language based development. If

You master one processor architecture and its assembly instructions, you can make the processor as flexible as gymnast. The major advantages of Assembly Language based development is listed below.

1. Efficient code memory and Data memory usage (memory optimisation).
2. High performance.
3. Low level hardware access.
4. Code Reverse Engineering.

Drawbacks of Assembly language Based development.

Every technology has its own pros and cons. From certain technology aspects assembly language development is most efficient technique. But it is having the following technical limitations also.

1. High Development time.
2. Developer Dependency.
3. Non-portable.

High level language Based development.

Assembly language based programming is highly time consuming, tedious and requires skilled programmers with sound knowledge of the target processor architecture. Also applications developed in assembly language are non-portable. Here comes the role of high level languages. Any high level language (like C, C++ or Java) with a supported cross compiler (for converting the applications developed in high level language to target processor specific assembly code we will discuss cross compilers in details). For the target processor can

most commonly used high level language for embedded firmware development. The language is 'C'. You may be thinking why 'C' is used as the popular embedded firmware development language. The answer 'C' is the well defined, easy to use high level language with extensive cross platform development tool support. (10)

The various steps involved in high level language based embedded firmware development is same as that of assembly language based development except that the conversion of source file written in high level language to objective file is done by cross platform compiler, whereas in assembly language based development is carried out by an assembler.

Advantages of High level language Based development.

The advantages are

1. Reduced development time.
2. Developer independency.
3. portability.

Limitations.

The merits offered by high level language based design take advantage over its limitations. Some cross compilers available for high level languages may not be so efficient in generating optimised target processor specific instructions. Target images created by such compilers may be messy and non-optimised in terms of performance as well as code size. The time required to execute a task also increases with the number of instructions. However modern cross-compilers are tending to adopt designs incorporating optimisation techniques for both code size and performance.

High level language based code snippets may not be efficient in accessing low level hardware where hardware accessing timing is critical. (of the order of nano or micro seconds).

Demonstration of High level language

High-level language based Embedded firmware Development.

Embedded firmware can be developed using any high level language which supports cross compiler for the target processor.

For Example, C, C++, Java.

The most commonly used high level language for the development of an embedded firmware is 'C', as it is well defined and easy to use with a vast cross platform development tool support.

The steps involved in development of a embedded firmware using high level language are similar to those involved in assembly language based development. Except cross compiler converts source file (written in high level language) to objectfile. whereas, in assembly language based development, an assembler is used to convert source file to object file.

Figure shows the various steps involved in conversion of a program written in high level language to it's equivalent binary file in machine language.

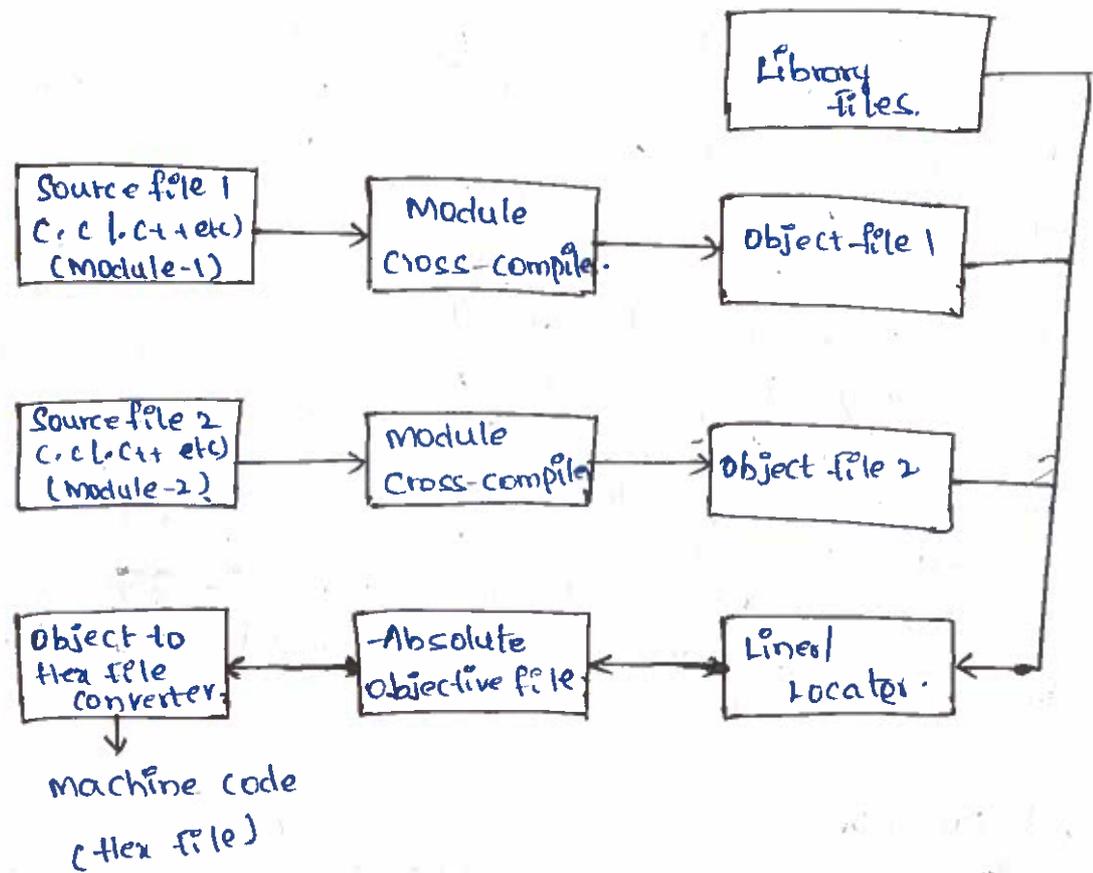


Fig: process -for conversion of High level language to machine language.

Source file to Object file translation.

In high level language, the program written is saved with the associated language extension (i.e., .c, .cpp for C++ etc). Modular programming approach is supported by all the high level languages. Thus, multiple source files i.e., modules can be written in corresponding high level language. A cross compiler is used for translating a high level source code into an executable object code. Different cross compilers are used for translating a high level languages for the same target processors. Therefore, every high level language is required to have its own cross compiler for conversion of high level source code to target processor machine code. This condition is unavoidable for the development of firmware. C51 cross compiler is most popular and is widely used in 'c' language for 8051 family of microcontrollers.

Library file creation and usage

A linker processes a library in which only required object modules are considered for creating a program. These library files are created with an extension '.lib'. Library file is a source code hiding technique, that hides source code of various functions written in a program and simultaneously distributes them to application developers so that they can be used various applications. This is done by supplying source code as library files and providing them with the details of public functions that are available from library. Library files can also be used in a project when added to it. 'LIB 51' from Keil software is an example for a library creator.

Linker and Locator

Linker and Locator is a software service which "perform linking of various object modules in a multi-module project and allocates absolute address to each module". A linker creates an absolute address to each object module by drawing out the object modules from the library and the object files that are created by the assembler. It is used to link any external dependent variables or functions and to overcome the external dependencies of various modules. The locator is used to assign fixed memory locations to all the codes and data provided. The hex files in the code memory of the processor or controller are created by using an absolute object file.

'BL51' from Keil software is an example for linker and locator.

Object to Hex file converter is a software used to create hex file i.e., ASCII file (hexadecimal representation of target application) from the 'Absolute Object file'. They are considered as machine code present in code memory of the processor or controller. Hex file representation vary with respect to different target processors/controllers etc.

Example:

For Intel processors/controllers, the hex file is 'Intel HEX' whereas for Motorola, the hex file is 'Motorola Hex'. 'OH1C1' from Keil software is an example for object to hex file converter utility.

Mixed Assembly and High level language.

Mixing assembly with high level language means combining of assembly routines with some high level language. For example, mixing of assembly language 'c'.

The mixing of languages is carried out in situations where the entire program is written in 'c'. In other hand, the cross-compiler doesnot support certain functions like interrupt service routine (ISR), or the situation where the programmer particular in the speed and optimized code provided by machine code (which is generated by hand written assembly). But, when accessing low level hardware, the timing constraints ^{are} are very crucial which is not compatible for cross compiler. Thus, mixing of assembly language with c language are preferred to handle such situations. For mixing 'c' with assembly the programmer should be aware of the following things

1. passing of parameters from 'c' routine to assembly.
2. Returning of values from assembly routine to 'c'.
3. calling of assembly routines from the 'c' code.

Consider an example of mixing assembly code with 'c' by taking Keil C51 cross compiler for 8051 controllers.

(i) Initially, a simple function is written in 'c' to pass parameters and return values as the programmer wants assembly routines to pass.

Ex:

(ii) A .SRC file is generated by the compiler using .SRC file directive instead of generating an .OBJ file.

Ex:

(iii) C-file should be compiled. The generated .SRC file contain the assembly for 'c' code.

(iv) .SRC file should be renamed as .AS1 file.

(v) The .AS1 file can be edited by inserting the assembly code in the function which has to be executed in the .AS1 file.

Ex:

Consider the following code which is taken from Keil C51 documentation.

```
#pragma SRC
```

```
unsigned char my-assembly-func(unsigned int argument)
```

```
{
```

```
    return (argument + 1); // dummy lines are inserted to access
                           all arguments and return values
```

```
}
```

This C-function during cross compilation generates the following assembly SRC file

```
NAME TESTCODE?PR?_my-assembly-time? TESTCODE SEGMENT
                                CODE
```

```

public - my-assembly-func
; #pragma SRC
; unsigned char my-assembly-func(
RSEG? PR? my-assembly-func? TESTCODE.
USING 0
-my-assembly-func:
; ... variable 'argument?010' assigned to register 'R61R7'
; SOURCE LINE #2
; unsigned in argument)
; {
; SOURCE LINE #4
; return (argument + 1); // dummy lines are inserted to access all
MOV R7, # arguments and returns
; }
; SOURCE LINE #6
; cool;
RET
; END OF -my-assembly-func
END

```

Inline Assembly

The other technique is inline assembly used for inserting specific assembly instructions of processors controller at any location of a source code which is written in a high level language 'C'.

Thus, this eliminates delay in calling assembly routine from a ~~base~~ code. The special keywords are used for pointing the beginning and ending of assembly instructions, any they are specific to the cross-compiler.

Ex: C51 uses the keywords #pragma asm and #pragma endasm for indicating the start and end of assembly code.

Ex: #pragma asm MOV A, #13H.
#pragma endasm.

Handwritten text at the top of the page, possibly a header or title, which is mostly illegible due to blurring.

Main body of handwritten text, consisting of several paragraphs. The text is extremely faint and difficult to read, but appears to be a continuous narrative or report. There are some faint markings and lines that might be part of a diagram or table, but they are not clearly defined.