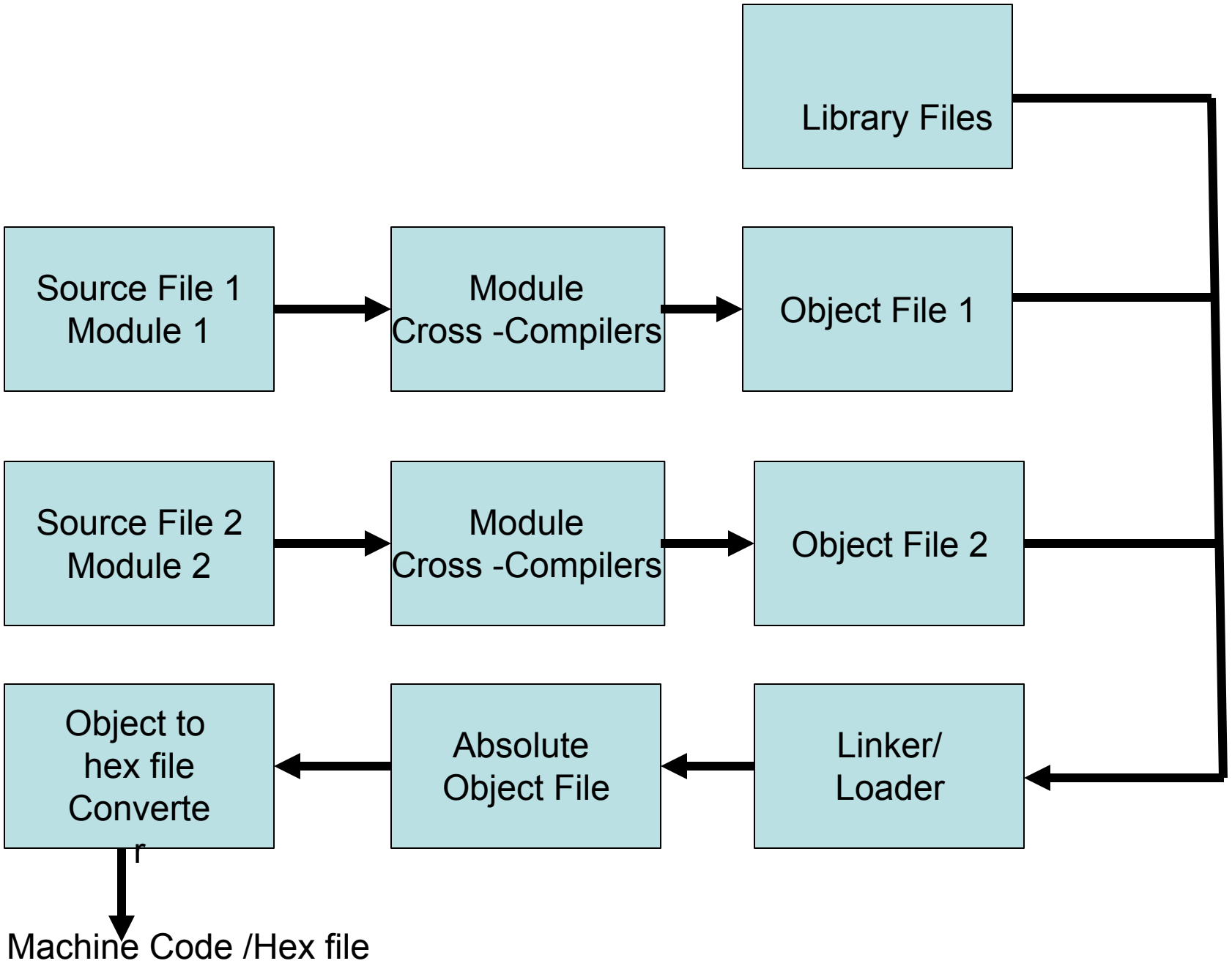
High Level Language Based Development

- Any High level language with a supported cross compilers for the target processor can be used for embedded firmware development
  - Cross Compilers are used for converting the application development in high level language into target processor specific assembly code
- Most commonly used language is C
  - C is well defined easy to use high level language with extensive cross platform development tool support


Library Files
Source File 1
Module 1
Module
Cross -Compilers
Object File 1
Source File 2
Module 2
Module
Cross -Compilers
Object File 2
Object to
hex file
Converte
r
Absolute
Object File
Linker/
Loader
Machine Code /Hex file

- The program written in any of the high level language is saved with the corresponding language extension
- Any text editor provided by IDE tool supporting the high level language in use can be used for writing the program
- Most of the high level language support modular programming approach and hence you can have multiple source files called modules written in corresponding high level language
- The source file corresponding to each module is represented by a file with corresponding language extension
- Translation of high level source code to executable object code is done by a cross compiler
- The cross compiler for different high level language for same target processor are different
- Without cross-compiler support a high level language cannot be used for embedded firmware develoment
  - Example C51 Compiler from Keil software

Advantages of High Level Language based Development

- **Reduced Development Time**
  - Developers requires less or little knowledge on the internal hardware details and architecture of the target processor
  - Syntax of high level language and bare minimal knowledge of memory organization and register details of target processor are the only pre-requisites for high level language based firmware development
  - With High level language, each task can be accomplished by lesser number of lines of code compared to the target processor specific assembly language based development

- Developer Independency
  - The syntax used by most of the high level languages are universal and a program written in high level language can be easily be understood by a second person knowing the syntax of the language
  - High level language based firmware development makes the firmware, developer independent
  - High level language always instruct certain set of rules for writing code and commenting the piece of code
  - If the developer strictly adheres to the rules, the firmware will be 100% developer independent

- Portability
  - Target applications written in high level languages are converted to target processor understandable format by a cross compiler
  - An application written in high level language for a particular target processor can be easily converted to another target processor with little effort by simply recompiling/ little code modification followed by recompiling the application for the required processor
  - This makes the high level language applications are 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

- The merits offered by high level language based design take advantage over its limitations
- Some cross compilers avail for the high level languages may not be so efficient in generating optimized target processor specific instructions
- Target images created by such compilers may be messy and no optimized in terms of performance as well as code size
- 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 access timing is critical(of the order of nano or micro seconds)
- The investment required for high level language based development

Mixing Assembly And High Level Language

- High level language and assembly languages are usually mixed in three ways
  - Mixing assembly language with high level language
  - Mixing high level language with Assembly
  - In line assembly programming

Mixing Assembly language with high level language (Assembly language with c)

- Assembly routines are mixed with C in situations where entire program is written in C and the cross compiler in use do not have built in support for implementing certain features like Interrupt Service Routine or if the programmer want to take the advantage of speed and optimized code offered by machine code generated by hand written assembly rather than cross compiler generated machine code
- When accessing certain low level hardware, the timing specification may be very critical and a cross compiler generated binary may not be able to offer the required time specifications accurately
- Writing the hardware access routine in processor specific assembly language and invoking it from C is the most advised method to handle such situations

- Mixing C and Assembly is little complicated in the sense-
  - the programmer must be aware of how parameters are passed from the C routine to Assembly and
  - values are returned from assembly routine to C and
  - how the assembly routine is invoked from the C code
- These are cross compiler dependent
- There is no universal rule for it
- You must get these information from the documentation of cross compiler you are using
- Different cross compilers implement these features in different ways depending upon the general purpose registers and the memory supported by the target processor

- Example (Keil C51 cross compiler for 8051 controller)

1. Write a simple function in C that passes parameters and return values the way you want your assembly routine to.
2. Use the SRC directive (#PRAGMA SRC) so that C compiler generate an .SRC file instead of .OBJ file
3. Compile the C code. Since the SRC directive is specified the .SRC file is generated. The .SRC file contain the assembly code generated for the C code you wrote
4. Rename .SRC to .A51 file
5. Edit .A51 file and insert the assembly code you want to execute in the body of the assembly function shell included in the .A51 file

Mixing High Level Language with Assembly (e.g. ‘C’ with Assembly Language)

- Mixing the code written in a high level language like ‘C’ and Assembly language is useful in the following scenarios:

1. The source code is already available in Assembly language and a routine written in a high level language like ‘C’ needs to be included to the existing code.

2. The entire source code is planned in Assembly code for various reasons like optimised code, optimal performance, efficient code memory utilisation and proven expertise in handling the Assembly, etc. But some portions of the code may be very difficult and tedious to code in Assembly. For example 16bit multiplication and division in 8051 Assembly Language.

3. To include built in library functions written in ‘C’ language provided by the cross compiler. For example Built in Graphics library functions and String operations supported by ‘C’.

- Most often the functions written in ‘C’ use parameter passing to the function and returns value/s to the calling functions.
- The major question that needs to be addressed in mixing a ‘C’ function with Assembly is that how the parameters are passed to the function and how values are returned from the function and how the function is invoked from the assembly language environment.
- Parameters are passed to the function and values are returned from the function using CPU registers, stack memory and fixed memory.
- Its implementation is cross compiler dependent and it varies across cross compilers. A typical example is given below for the Keil C51 cross compiler

- C51 allows passing of a maximum of three arguments through general purpose registers R2 to R7.
- If the three arguments are char variables, they are passed to the function using registers R7, R6 and R5 respectively.
- If the parameters are int values, they are passed using register pairs (R7, R6), (R5, R4) and (R3, R2).
- If the number of arguments is greater than three, the first three arguments are passed through registers and rest is passed through fixed memory locations.
- Refer to C51 documentation for more details.
- Return values are usually passed through general purpose registers.

- R7 is used for returning char value and register pair (R7, R6) is used for returning int value.
- The ‘C’ subroutine can be invoked from the assembly program using the subroutine call Assembly instruction (Again cross compiler dependent).
- E.g. LCALL _Cfunction

  Where Cfunction is a function written in ‘C’.
- The prefix _ informs the cross compiler that the parameters to the function are passed through registers.
- If the function is invoked without the _ prefix, it is understood that the parameters are passed through fixed memory locations.

Inline Assembly

- Inline assembly is another technique for inserting target processor/controller specific Assembly instructions at any location of a source code written in high level language ‘C’.
- This avoids the delay in calling an assembly routine from a ‘C’ code
- Special keywords are used to indicate that the start and end of Assembly instructions.
- The keywords are cross-compiler specific.
- C51 uses the keywords #pragma asm and #pragma endasm to indicate a block of code written in assembly.
- E.g. #pragma asm

  MOV A, #13H

  #pragma endasm