Loaders & Linkers: Introduction

The process of transforming a source program into an executable program involves several critical steps, with **loaders** and **linkage editors** playing a pivotal role in the final stages of this transformation. As previously discussed, an **assembler** is responsible for converting a source program, written in a high-level language or assembly language, into an **object program**, which is essentially a machine-code representation of the original program. The **loader** then takes this object program as its input, meticulously prepares it for execution, and subsequently loads the executable code into the main memory of the computer system. This makes the loader a fundamental utility program that is directly responsible for initiating the execution process of a program.

The broader term "loader" is often used to encompass both the functions of a traditional loader and a **linker**, also known as a **linkage editor**. In some system architectures, the linking operation might be handled by a separate program (the linker or linkage editor), while the loader specifically performs the tasks of relocation and the physical loading of the program into memory. Regardless of whether these functions are combined or separated, the overarching goal remains the same: to make the program ready for execution. A significant advantage of this modular approach is that language translators, such as assemblers or compilers, typically produce object programs in a standardized format. This consistency allows for a single system loader/linker to be utilized across various programming languages, simplifying the system software architecture.

Basic Loader Functions

The loader performs a set of fundamental activities to ensure that an object program is properly prepared and placed into memory for execution. These activities are crucial for the seamless operation of any software. The four core functions of a loader are allocation, linking, relocation, and loading.

1. **Allocation**: This initial function involves the loader determining and assigning space in the memory for the program that is to be loaded. The loader achieves this by calculating the total size of the program, including its code and data segments, to ascertain the exact amount of memory required. Once this calculation is complete, the loader reserves a contiguous or segmented block of memory for the program, ensuring that it has dedicated space to reside and execute without conflicts. This process of reserving memory space is specifically referred to as allocation.

2. **Linking**: Linking is the process by which the loader resolves all symbolic references between different object modules. In complex programs, code and data are often organized into multiple, separately compiled or assembled modules. These modules may refer to functions, subroutines, or data located in other modules or in standard library routines. The linking function assigns actual memory addresses to all these user-defined subroutines and library subroutines, effectively connecting all the pieces of the program together. This ensures that when one part of the program calls another, the correct memory location is accessed.

3. **Relocation**: Many programs contain address-dependent locations, such as address constants or relative addresses, that are initially defined relative to a starting point (e.g., zero). However, when a program is loaded into memory, its actual starting address might be different from this assumed base. Relocation is the activity performed by the loader

to adjust these address-dependent locations to correspond to the actual memory space allocated. This dynamic adjustment ensures that all memory references within the program point to the correct physical addresses after the program has been loaded, regardless of where it is placed in memory.

4. **Loading**: The final and most direct function of the loader is loading. This involves the physical placement of all the machine instructions and data of the corresponding programs and subroutines into their designated locations within the main memory. Once the allocation, linking, and relocation steps have been completed, the loader literally transfers the executable code and data from the object file into the reserved memory segments. After this step is completed, the program is fully situated in memory and is ready for the CPU to begin its execution. In essence, while the other functions prepare the program for memory, loading is the act of putting it there.

The most fundamental aspect of a loader, which underpins all other functions, is its ability to take an object program and bring it into memory, making it available for execution by the computer's processor.

Loader Schemes

The methodology by which a loader performs its functions can vary significantly, leading to different categories or "schemes" of loaders, each with its own operational characteristics, advantages, and disadvantages. These schemes represent different trade-offs in terms of memory usage, execution efficiency, and programming flexibility.

1. "Compile and Go" Loader (or "Assemble and Go")

The "Compile and Go" (or "Assemble and Go") loader scheme represents the simplest approach to program loading. In this model, the translation and loading processes are tightly integrated and occur almost simultaneously. Instructions from the source program are read line by line by the translator (assembler or compiler). As each instruction's machine code is generated, it is immediately placed directly into a predefined location in the main memory. This means the assembler itself resides in one part of memory, and as it translates, the resulting machine instructions and data are directly deposited into their assigned memory locations. Upon the completion of the assembly process, the starting address of the program is directly transferred to the program counter, initiating execution. A historical example of a FORTRAN compiler that utilized this "load and go" scheme was WATFOR-77. In this scheme, the "loader" component is often minimal, sometimes consisting of just a single instruction that transfers control to the starting point of the newly assembled program in memory. Figure 6.2 in the provided PDF visually illustrates this direct path from source program through a combined "Compile-and-go translator" (Assembler + Loader) directly to a loaded program in memory.

Advantages:

This scheme is notably simple to implement. Because the assembler is placed in one part of the memory, the loader's function is straightforward: it simply loads the assembled machine instructions into the memory. This direct approach means there are no extra procedures or intermediate steps involved, making it a very uncomplicated solution for basic program execution.

Disadvantages:

While simple, this scheme suffers from several significant drawbacks. A considerable portion of memory is occupied by the assembler, rendering that space unavailable for the object program, which constitutes a wastage of valuable memory resources. Given that this scheme combines both assembler and loader activities into a single program, this combined entity occupies a large block of memory. A critical disadvantage is the absence of an .obj (object) file; the source code is directly converted to its executable form. Consequently, even minor modifications to the source program necessitate re-assembly and re-execution every single time the program needs to be run, making it a highly time-consuming activity. Furthermore, this scheme is incapable of handling multiple source programs or programs written in different programming languages, as a typical assembler is designed to translate only one source language into its corresponding target language. For programmers, creating and maintaining orderly, modular programs becomes extremely difficult, as the "compile and go" loader is not equipped to handle such structured programs. Ultimately, the continuous re-assembly and re-execution contribute to significantly longer execution times.

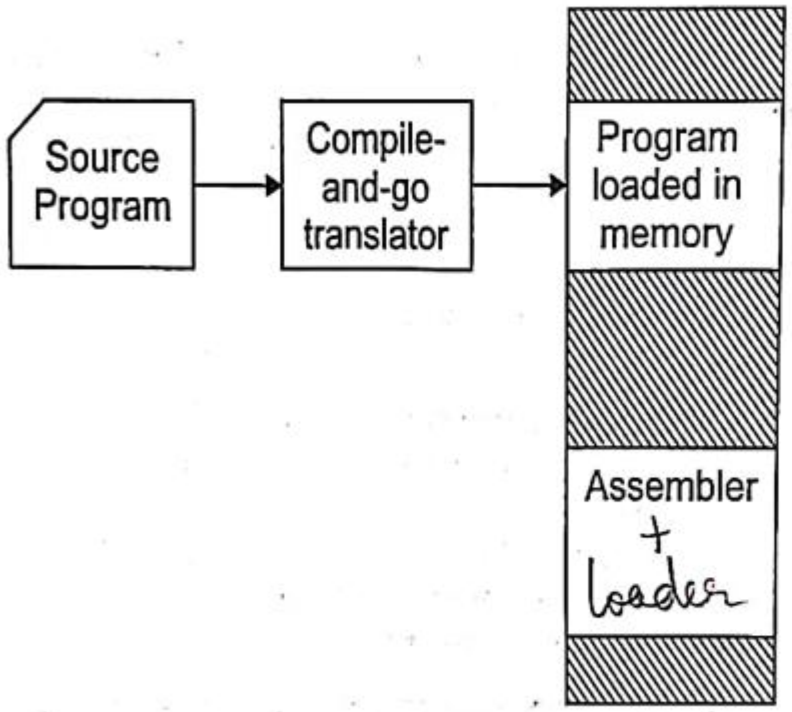


Figure 6.2: Compile-and-go loader scheme

2. General Loader Scheme

To overcome the limitations of the "Compile and Go" scheme, the **General Loader Scheme** introduces an intermediate step and a more modular design. In this approach, the source program is first explicitly converted into an **object program** by a separate translator, such as an assembler or compiler. This object program, which contains the machine instructions and data in a coded form, can then be saved on a secondary storage device (like a disk). The **loader** then takes these object modules as input, accepts them from the secondary storage, and meticulously places the machine instructions and data into an executable form at their assigned memory locations. The loader itself occupies a relatively smaller portion of the main memory compared to the assembler. This separation of translation and loading processes, facilitated by the intermediate object program, requires the addition of a dedicated loader program to the system. Figures 6.1 and 6.3 in the PDF effectively illustrate this scheme, showing source programs being independently translated into object programs, which are then processed by a dedicated loader to be loaded into memory.

Advantages:

This modular scheme offers several key advantages. A primary benefit is that the program does not need to be retranslated (re-assembled or re-compiled) each time it is run. Once the source program is initially executed and an object program is generated, the loader can efficiently utilize this existing object program to convert it into an executable form, provided the program has not been modified. This eliminates the necessity of reassembly for subsequent runs. Memory utilization is also significantly improved compared to the "compile and go" scheme because the assembler is not required to reside in memory during the loading and execution phases. The loader, being typically smaller in size than the assembler, occupies less memory, thereby making a greater portion of the main memory available for the user's program. Additionally, this scheme provides enhanced flexibility, making it possible to write source programs with multiple modules and even in multiple programming languages, as the source programs are first consistently converted into object programs, which the loader can then accept and integrate into an executable form.

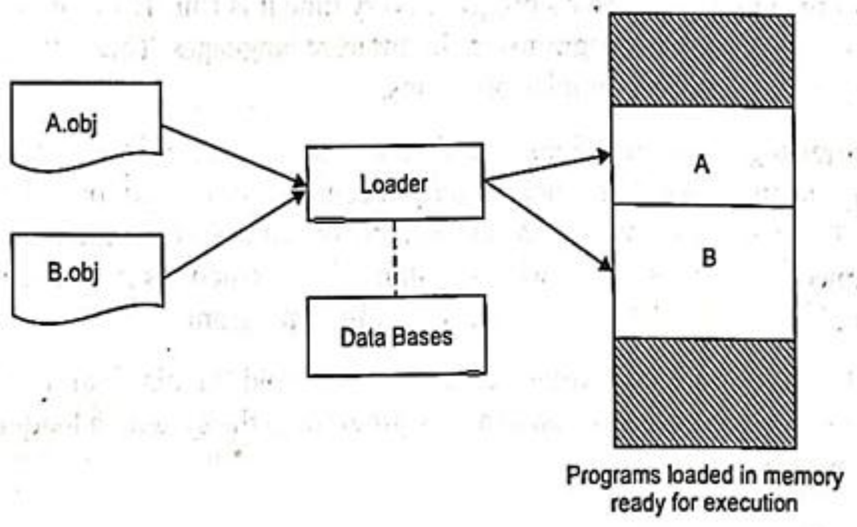


Figure 6.1: General Loading Scheme

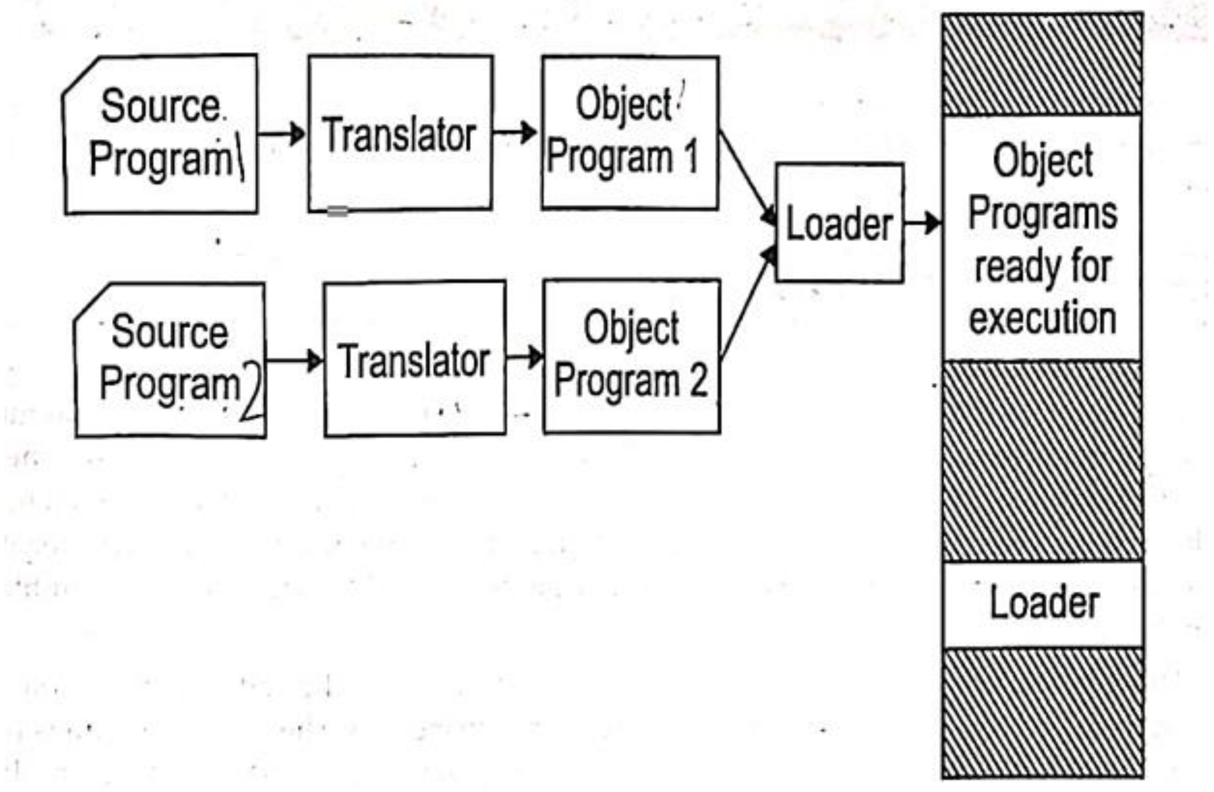


Figure 6.3: General Loader Scheme

3. Absolute Loader

The **Absolute Loader** is a specific type of loader within the general loader scheme, characterized by its reliance on pre-determined memory addresses. In this scheme, the assembler (or programmer) generates **relocated object files**, meaning that the object code within these files is already fixed to specific, absolute memory addresses. The loader's primary function is then to accept these object files and simply place their contents directly into the specified memory locations. It is termed "absolute" precisely because it does not require any dynamic relocation information; this information is either provided by the programmer or inherently handled by the assembler during object code generation, where the absolute starting address of every module is presumed to be known beforehand and stored within the object file itself. Consequently, the loader's task becomes very straightforward: it merely reads the object code and places it at the exact memory addresses indicated in the object file. The assembler outputs the machine language translation in a format similar to the 'assemble-and-go' scheme, but with the key difference that data can be stored on a secondary device rather than directly in memory at assembly time. This scheme also allows more memory to be available to the user, as the assembler is not present in memory during the loading phase. In an absolute loading scheme, the four core loader functions are handled as follows: **Allocation** is managed by the programmer, **Linking** is handled by the programmer, **Relocation** is implicitly performed by the assembler (as it generates code for fixed locations or the code is already adjusted), and **Loading** is performed by the loader.

Advantages:

The absolute loader scheme offers several distinct advantages. It is remarkably simple to implement, as the loader's primary task is merely to place code at predefined addresses. This simplicity contributes to an efficient execution process once the program is loaded. Furthermore, this scheme accommodates multiple programs or source programs written in different languages. If there are multiple programs composed in varying languages, their respective language assemblers will convert them to a common object file format, with all address resolution handled during this conversion. This simplifies the loader's subsequent task, as it merely obeys the instructions regarding where to place the object code in main memory, without needing to perform complex address calculations or linking itself.

Disadvantages:

Despite its simplicity, the absolute loader scheme comes with significant disadvantages, largely stemming from the burden it places on the programmer. It becomes the programmer's explicit duty to adjust all inter-segment addresses and manually perform the linking activity. This necessitates that the programmer possess a deep understanding of memory management and the precise memory layout. A critical issue arises if any modification is made to a program segment: the length of that module may change, which in turn alters the starting addresses of all immediate subsequent segments. The programmer is then solely responsible for meticulously updating the corresponding starting addresses for all affected modules. Additionally, when a program needs to branch from one segment to another, the programmer must explicitly know the absolute starting address of the target module to correctly specify it in the JMP instruction. This constant need for manual address management and linking makes the scheme inflexible and prone to errors, especially for large and complex programs with multiple subroutines, as the programmer must remember and explicitly use the absolute address of each subroutine for linkage.

Design of an Absolute Loader

The **Absolute Loader** is the simplest type of loader scheme that fits the general model of program loading. In this scenario, the assembler generates the machine language translation of the source program in a format very similar to that used in the 'assemble-and-go' scheme. The key difference, however, is that the translated machine code and data are stored on a secondary device rather than being directly placed into memory during the assembly process. The absolute loader then takes this machine language program from the secondary device and precisely places it into memory at the exact locations specified by the assembler. This approach has the significant advantage of making more memory available to the user, as the assembler itself is not required to be present in memory during the program loading time.

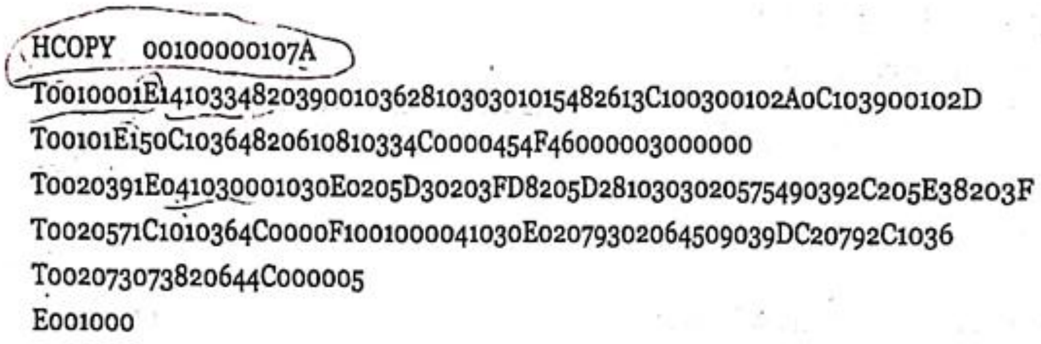
An example of an object program, which would be the output of an assembler or compiler for use with an absolute loader, is provided in the PDF (Figure 6.4: Example Object Program). This example demonstrates the structured format of the object file, typically including Header, Text, and End records. The Header Record usually contains verification information, such as the program name and its length. The Text Records contain the actual machine code instructions and data, along with the memory addresses where they are to be loaded. The End Record specifies the starting address for program execution after loading is complete.

The design of an absolute loader is inherently simple because it does not need to perform complex functions like dynamic linking or program relocation. Its operations are straightforward, and the loading process can often be accomplished in a single pass. The basic algorithm for an absolute loader involves:

1. **Reading the Header Record**: This step is crucial for verifying the program name and its expected length to ensure the correct program is being loaded.
2. **Processing Text Records**: The loader reads each Text Record. If the object code is represented in character form (e.g., hexadecimal characters), it must first convert these characters into their internal binary representation. Then, the loader moves this object code to the exact memory locations specified within that Text Record. This process continues iteratively for all Text Records.
3. **Handling the End Record**: Once the End Record is encountered, the loading process is complete. The loader then jumps to the address specified in this record, which is the designated starting point for the execution of the newly loaded program. This simple, direct approach makes absolute loaders efficient in terms of execution time.

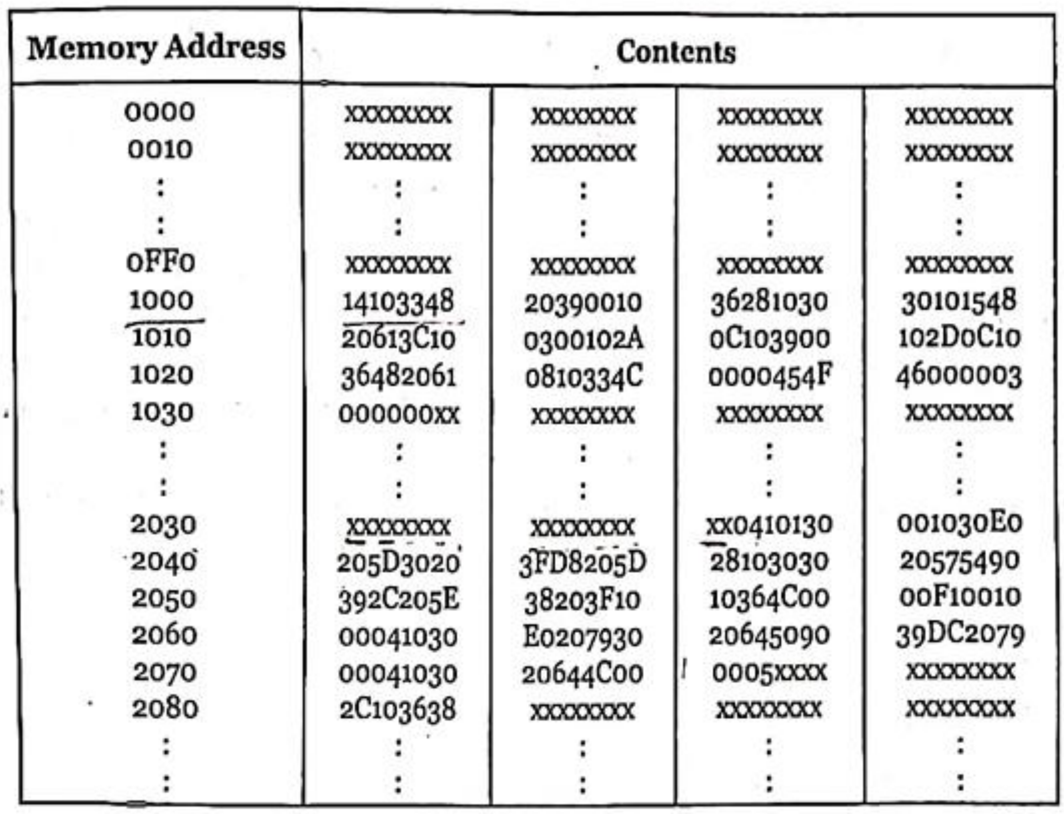
It is important to note how object programs are represented for absolute loaders. Often, they use hexadecimal representation in character form. For instance, a machine operation code like '14' (hexadecimal) for an STL instruction might be represented by the characters '1' and '4' in the object file. When the loader reads these, they initially occupy two bytes of memory (one for '1', one for '4'). However, for execution, this operation code must be packed into a single byte with the hexadecimal value 14. Thus, each pair of bytes from the object program record must be combined into one byte during the loading process. While convenient for human readability and printing, this character-based representation is less efficient in terms of both storage space and execution time compared to binary formats, where each byte of object program is stored as a single byte. However, for simplicity and to avoid issues with control characters in binary files, hexadecimal conventions are often used in discussions.

In an absolute loading scheme, the four primary loader functions are handled as follows: **Allocation** is determined by the programmer who specifies the load address. **Linking** is also managed by the programmer, who must resolve all inter-segment references manually. **Relocation** is performed by the assembler during the assembly process, as it generates the object code with fixed absolute addresses, meaning no further relocation is needed by the loader at load time. Finally, the **Loading** itself is the direct responsibility of the absolute loader, which places the code into memory.



```
HCOPY  00100000107A
T0010001E141033482039001036281030301015482613C100300102A0C103900102D
T00101E150C1036482061081033 4C0000454F46000003000000
T0020391E041030001030E0205D30203FD8205D2810303020575490392C205E38203F
T0020571C1010364C0000F1001000041030E02079302064509039DC20792C1036
T002073073820644C000005
E001000
```

Figure 6.4: Example Object Program



Memory Address	Contents			
0000	xxxxxxxx	xxxxxxxx	xxxxxxxx	xxxxxxxx
0010	xxxxxxxx	xxxxxxxx	xxxxxxxx	xxxxxxxx
:	:	:	:	:
:	:	:	:	:
0FF0	xxxxxxxx	xxxxxxxx	xxxxxxxx	xxxxxxxx
1000	14103348	20390010	36281030	30101548
1010	20613C10	0300102A	0C103900	102D0C10
1020	36482061	0810334C	0000454F	46000003
1030	000000xx	xxxxxxxx	xxxxxxxx	xxxxxxxx
:	:	:	:	:
:	:	:	:	:
2030	xxxxxxxx	xxxxxxxx	xx0410130	001030E0
2040	205D3020	3FD8205D	28103030	20575490
2050	392C205E	38203F10	10364C00	00F10010
2060	00041030	E0207930	20645090	39DC2079
2070	00041030	20644C00	0005xxxx	xxxxxxxx
2080	2C103638	xxxxxxxx	xxxxxxxx	xxxxxxxx
:	:	:	:	:
:	:	:	:	:

Figure 6.5: Loading of an object program in memory

Relocating Loaders

The absolute loader, while simple and efficient for basic scenarios, presents several potential disadvantages, particularly concerning memory management in more complex systems. One major limitation is that the programmer must explicitly specify the exact memory location where the program will be loaded. This is manageable for very simple machines with limited memory running a single program. However, in advanced machines with larger memories where users might want to run multiple independent programs concurrently, sharing memory between them, this becomes impractical. In such multi-programming environments, the user (or the operating system) cannot know in advance where each program will be loaded. Efficient memory sharing, therefore, necessitates the use of **relocatable programs** rather than absolute ones.

A **relocating loader** is designed to address this limitation by allowing programs to be loaded at any arbitrary location in memory. Unlike absolute loaders, which expect fixed addresses, relocating loaders dynamically adjust the addresses within the program at load time based on the actual starting address assigned by the operating system. This dynamic adjustment is the core function of **relocation**. Relocation is defined as the activity where address-dependent locations in a program, such as address constants, are adjusted according to the allocated space in memory. While the absolute loader relies on the assembler to effectively "pre-relocate" the code to a fixed address, a relocating loader performs this adjustment dynamically, meaning the object code contains information that allows the loader to modify addresses during the loading process.

The need for relocating loaders also arises from the difficulty in using subroutine libraries with absolute programs. Most subroutine libraries contain a vast number of routines, far more than any single program would typically use. To make efficient use of memory and avoid loading unnecessary code, it is desirable to load only those routines that are actually needed by a program. This selective loading and linking can only be achieved effectively if the subroutines themselves are relocatable and can be loaded into any available memory space, and then their addresses adjusted by the loader. The ability to create relocatable programs allows for greater flexibility in memory management, supports multi-programming environments, and facilitates the efficient use of shared libraries by allowing programs to be loaded and run irrespective of their eventual physical memory address. The exact mechanism for relocation typically involves **relocation bits** or **relocation dictionaries** within the object file, which guide the loader on which parts of the code need address modification.

Design of a Linking Loader

A **linking loader** is a type of loader that combines the functions of both linking and loading. While the absolute loader relies on the programmer or assembler to handle all linking and relocation, a linking loader takes on the crucial responsibility of resolving symbolic references between different program modules and library routines, as well as adjusting addresses (relocation) before loading the executable code into memory. This makes programs more modular and flexible, enabling the use of subroutines written independently and facilitating the creation of complex software systems.

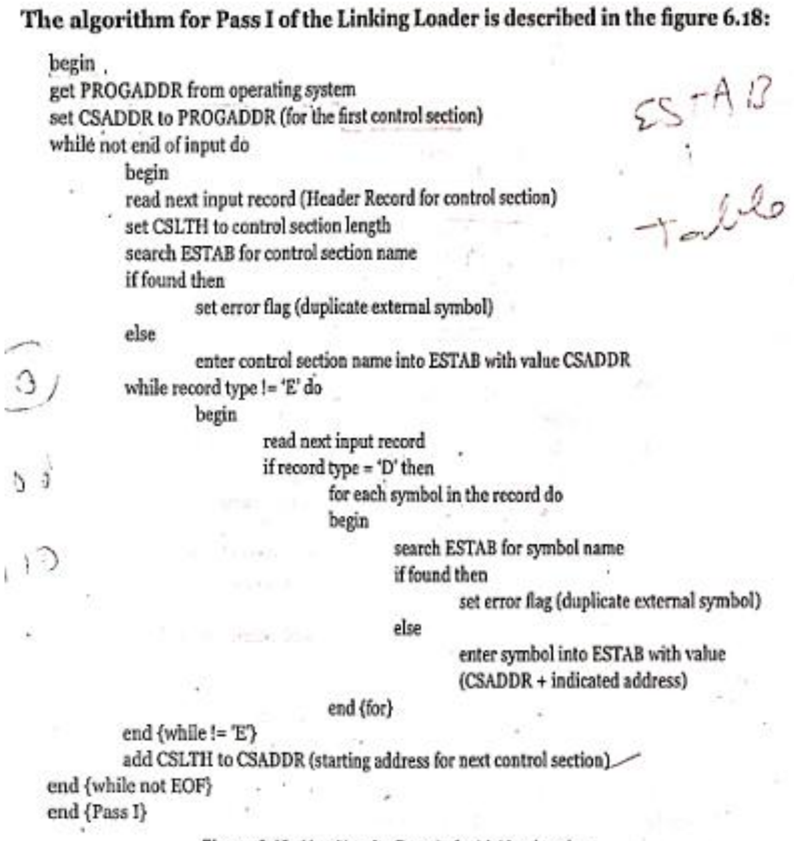
The core design of a linking loader revolves around its ability to perform the **linking** and **relocation** functions dynamically at load time. As discussed previously, **linking** involves resolving symbolic references (for both code and data) between various object modules by assigning concrete memory addresses to all user-defined subroutines and external library subroutines. This means that a linking loader must be able to read and interpret the object files produced by assemblers or compilers, which, unlike the simple object files for absolute loaders, must contain additional information. This vital information typically includes:

- **External references**: Symbols (labels) that are defined in other modules but referred to in the current module.
- **Entry points**: Symbols that are defined in the current module and can be referred to by other modules.
- **Relocation information**: Details about which parts of the code and data are address-dependent and need adjustment.

The process of a linking loader would conceptually involve multiple passes or internal data structures to manage these complexities. In a typical approach, the loader might first perform a pass to determine the total memory requirements and establish the loading addresses for all interconnected program segments and subroutines. During this pass, it would build a symbol table (sometimes called a global symbol table or external symbol table) that maps all external symbols and entry points to their eventual memory addresses. This process is crucial for resolving the cross-module references.

Following the establishment of addresses, the linking loader proceeds to the actual loading and relocation phase. It retrieves the machine instructions and data from the object modules. For each instruction or data item that contains an address-dependent reference (indicated by relocation information within the object file), the loader adjusts these addresses based on the final assigned load address for that module and the resolved addresses of any external symbols. This is where the **relocation** function comes into play, ensuring that all address constants and jumps within the program correctly point to their targets in the loaded memory space. Finally, the adjusted machine instructions and data are physically placed into the designated memory locations, making the complete, linked program ready for execution.

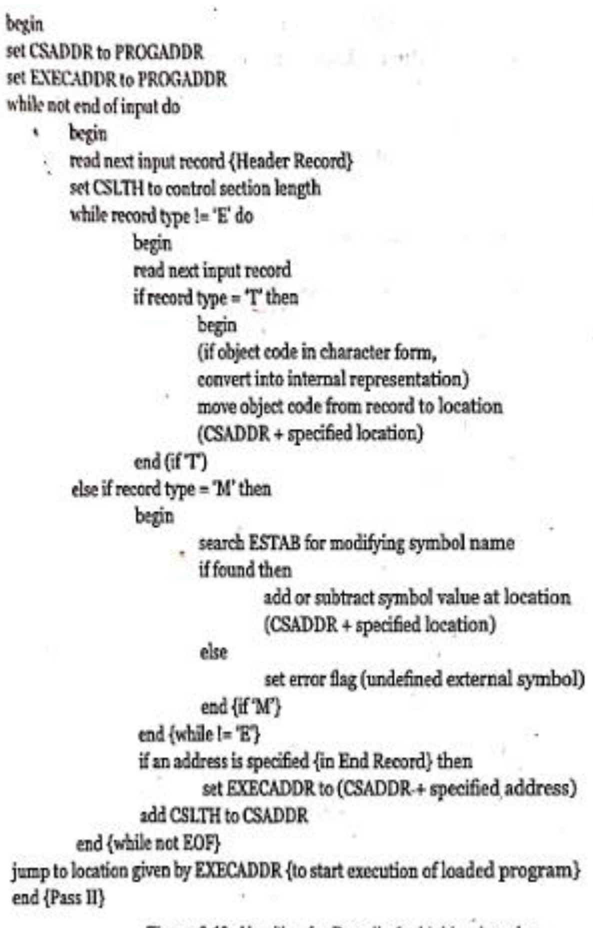
It is clear that a linking loader is essential for modern operating systems and programming practices where modularity, shared libraries, and dynamic memory allocation are key. The complexities of a linking loader often necessitate the use of intermediate data structures to manage symbols, track relocation information, and handle potential conflicts or unresolvable references, ensuring that all parts of a multi-segment program correctly interact once loaded into memory.



The algorithm for Pass I of the Linking Loader is described in the figure 6.18:

```
begin
get PROGADDR from operating system                                   ES-AB
set CSADDR to PROGADDR (for the first control section)
while not end of input do
        begin
        read next input record (Header Record for control section)
        set CSLTH to control section length                          Table
        search ESTAB for control section name
        if found then
                set error flag (duplicate external symbol)
        else
                enter control section name into ESTAB with value CSADDR
3)      while record type != 'E' do
                begin
                        read next input record
                        if record type = 'D' then
                                for each symbol in the record do
                                begin
                                        search ESTAB for symbol name
                                        if found then
                                                set error flag (duplicate external symbol)
                                        else
                                                enter symbol into ESTAB with value
                                                (CSADDR + indicated address)
                                end {for}
        end {while != 'E'}
        add CSLTH to CSADDR (starting address for next control section)
end {while not EOF}
end {Pass I}
```

Figure 6.18: Algorithm for Pass I of a Linking Loader



```
begin
set CSADDR to PROGADDR
set EXECADDR to PROGADDR
while not end of input do
    begin
    read next input record {Header Record}
    set CSLTH to control section length
    while record type != 'E' do
        begin
        read next input record
        if record type = 'T' then
            begin
            (if object code in character form,
            convert into internal representation)
            move object code from record to location
            (CSADDR + specified location)
        end (if 'T')
    else if record type = 'M' then
        begin
            search ESTAB for modifying symbol name
            if found then
                add or subtract symbol value at location
                (CSADDR + specified location)
            else
                set error flag (undefined external symbol)
            end {if 'M'}
        end {while != 'E'}
        if an address is specified {in End Record} then
            set EXECADDR to (CSADDR + specified address)
        add CSLTH to CSADDR
    end {while not EOF}
jump to location given by EXECADDR {to start execution of loaded program}
end {Pass II}
```

Figure 6.19: Algorithm for Pass II of a Linking Loader

Linkage Editors & Their Functions

A **linkage editor** (sometimes used interchangeably with the term "linker," though there can be subtle distinctions) is a crucial utility program in the software development process. It operates on one or more object files (the output of assemblers or compilers) to produce a single executable file or another object file. While we use the term "loader" to sometimes encompass both loading and linking functions, a linkage editor specifically focuses on the linking aspect.

The primary function of a linkage editor is to **combine and resolve external references** between separate object modules. This process involves:

1. **Collecting Object Modules**: The linkage editor takes as input a set of object files, which may include the main program module, subroutines, and library routines.

2. **Symbol Resolution**: It examines the symbol tables of each object module to resolve references to external symbols (functions, variables, etc.) defined in other modules. For each reference, the linkage editor finds the corresponding definition and determines its address.

3. **Relocation**: Like a relocating loader, the linkage editor might also perform relocation, adjusting addresses within the code and data sections to reflect the combined layout of the modules. However, the relocation performed by a linkage editor is often more about adjusting relative addresses *within* the combined output rather than for a specific load address in memory (which is more the concern of a loader).

4. **Outputting the Executable**: The linkage editor produces a single output file, which is typically an executable file ready to be loaded into memory and run.Alternatively, it can create a new object file, which might be further processed by another linkage editor or loader.

In contrast to a loader, which primarily focuses on placing code into memory for execution, a linkage editor prepares the code *before* it is loaded. It creates a unified, coherent executable image. The resulting executable typically contains all the necessary code and data from the input object files, linked together so that they can function as a single program. The functions of a linkage editor are critical for modular programming, where programs are divided into smaller, manageable pieces that can be developed and compiled independently. The linkage editor brings these pieces together into a working whole.

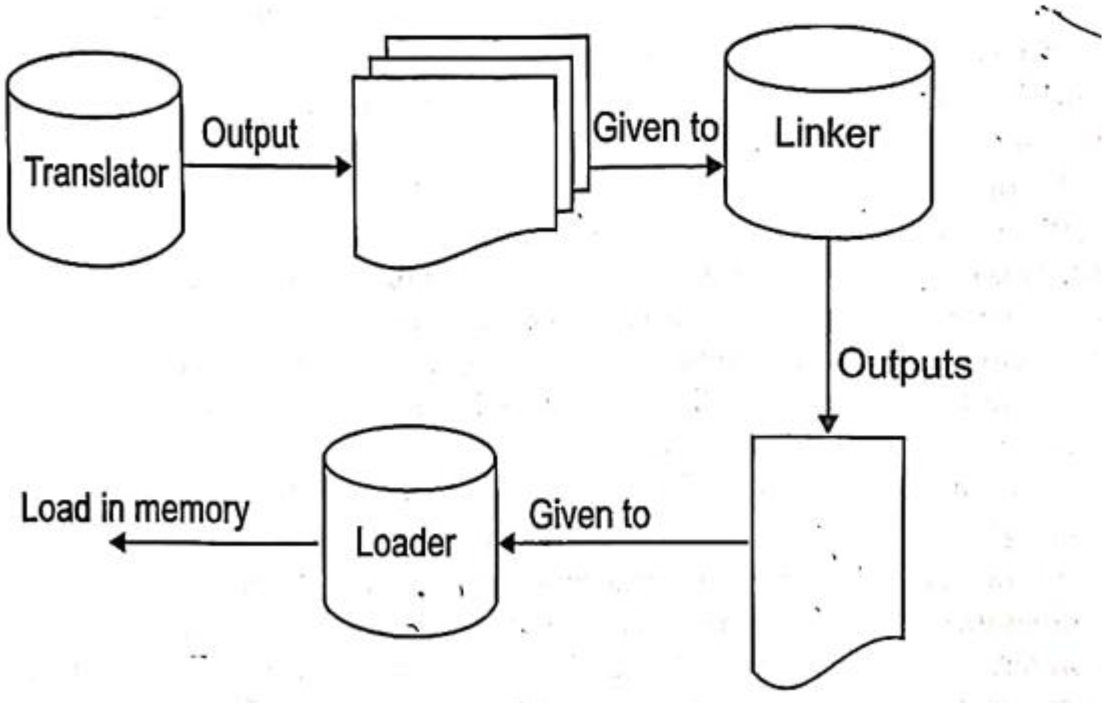


Figure 6.12: Process of linking a program

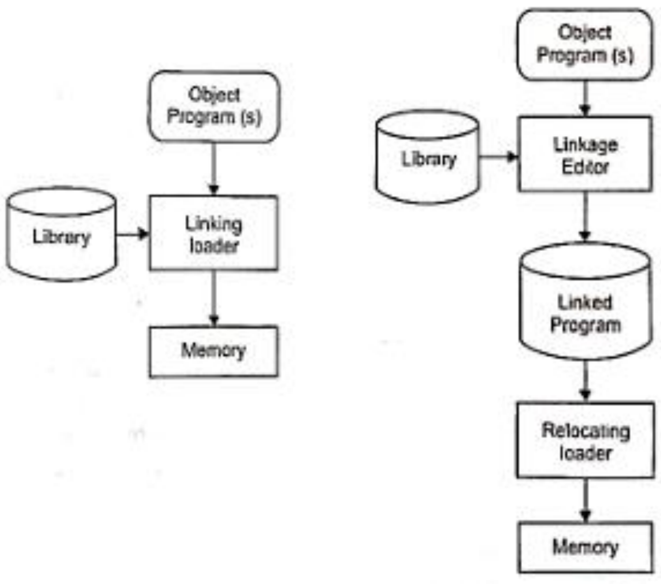


Figure 6.23: Processing of an Object Program using Linking Loader and Linkage Editor

Dynamic Linking

Dynamic linking – linking postponed until execution time.

A small piece of code, called a **stub**, is used to locate the appropriate memory-resident library routine. The stub replaces itself with the address of the routine and then executes the routine. The operating system checks if the routine is in the process's memory address. If it's not in the address space, it's added. Dynamic linking is particularly useful for libraries, also known as shared libraries.

How Dynamic Linking Works

When a program is compiled to use shared libraries, these libraries are dynamically linked by default in many systems. Instead of embedding the library code directly into the executable, the linker places information into the executable that tells the **runtime linker** (or dynamic linker), which is often a special part of the operating system, where to find the necessary shared object modules and how to bind the references.

When the program starts:

- The operating system kernel loads the executable file.
- It then identifies the path of the dynamic linker embedded within the executable.
- The dynamic linker takes over, loading the initial executable image and all the dynamically linked libraries it depends on.
- It resolves the symbolic references (e.g., function calls, variable accesses) between the program and these shared libraries. This binding happens at runtime.
- In some cases, a technique called "lazy linking" or "lazy binding" is used, where calls to functions in a library are linked to their implementations only when the function is first called.

Implementations vary across operating systems. For example, Microsoft Windows uses **Dynamic-Link Libraries (DLLs)** with file extensions like .dll, .ocx, or .drv. Unix-like systems, often using the ELF format, embed the path of the dynamic linker in the executable image, and the kernel loads this dynamic linker first, which then handles the loading and binding of shared libraries.

Advantages of Dynamic Linking

Dynamic linking offers several significant benefits:

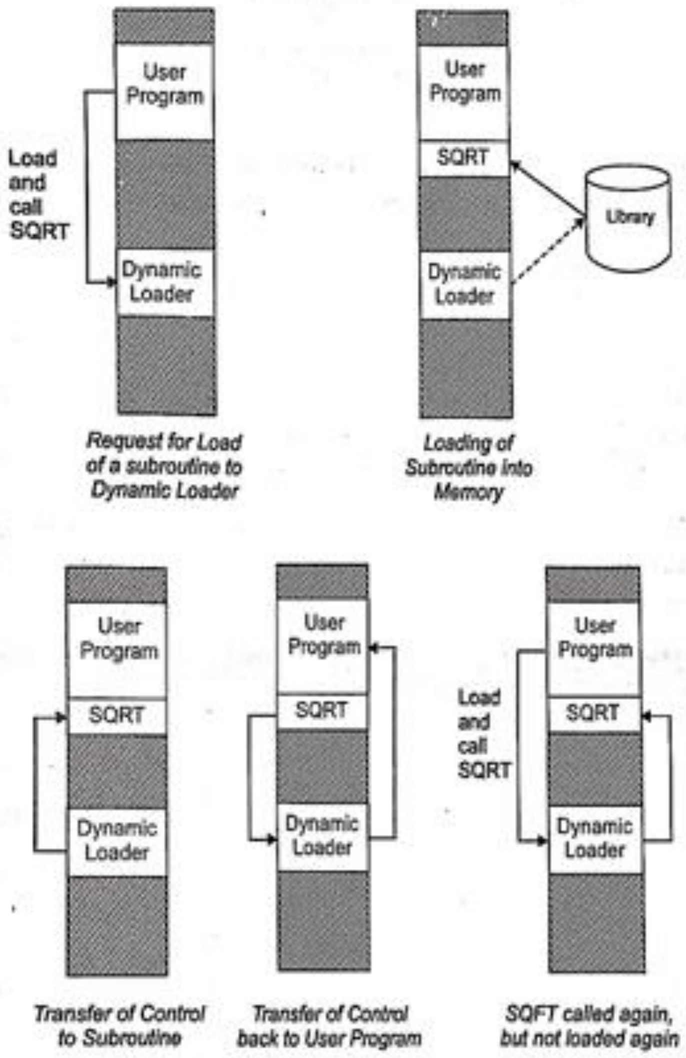
1. **Reduced Executable Size**: The code for shared libraries is not duplicated in every executable that uses it. Instead, the executable only contains a reference to the library, resulting in smaller executable files on disk.
2. **Efficient Memory Usage**: Shared libraries are loaded into memory only once, typically in a shared segment, and can be shared by multiple processes concurrently. If several applications or multiple instances of the same application use the same shared library, they all access the single copy in memory, thereby conserving virtual and physical memory. This can also lead to fewer page faults.
3. **Easier Updates and Maintenance**: Shared libraries can be updated or replaced without requiring recompilation or relinking of the applications that use them. This allows for easier deployment of bug fixes, security patches, or new features in libraries without distributing new versions of all dependent applications.
4. **Modularity**: It promotes modular programming, allowing large applications to be broken down into smaller, independently developed and deployed modules.
5. **Reduced Load Time (in some cases)**: If a shared library is already loaded into memory by another running program, the load time for a new program that uses the same library can be reduced.

Disadvantages of Dynamic Linking

Despite its advantages, dynamic linking also has some drawbacks:

1. **Runtime Overhead**: Dynamic linking introduces a performance overhead at runtime. The process of locating, loading, and resolving symbols for shared libraries requires CPU cycles. While lazy linking can reduce initial load time, subsequent function calls might still incur a slight overhead for symbol resolution if not already bound.
2. **Dependency Issues ("DLL Hell")**: Dynamically linked programs are highly dependent on the presence and compatibility of the shared libraries they use. If a library is changed (e.g., an incompatible new version is installed), removed from the system, or corrupted, programs relying on that specific version might cease to function, leading to issues commonly referred to as "DLL Hell" (on Windows) or similar dependency problems on other systems.
3. **Reduced Locality of Reference**: If a program uses only a few routines scattered widely within a large shared library, more memory pages might need to be touched to access these routines compared to if they were directly bound into the executable. This can potentially increase page faults and Translation Lookaside Buffer (TLB) misses, impacting performance if the program is the sole user of those routines.

4. **Portability Challenges**: Programs linked dynamically are not as easily portable to systems that lack the specific versions of the shared libraries they depend on, unlike statically linked programs which bundle all their dependencies.



User Program
Load and call SQRT
Dynamic Loader
Request for Load of a subroutine to Dynamic Loader
SQRT
Library
Loading of Subroutine into Memory
Transfer of Control to Subroutine
Transfer of Control back to User Program
SQFT called again, but not loaded again

Figure 6.24: Loading and calling of a subroutine using Dynamic Linking

Bootstrap Loader

A **bootstrap loader** is a specialized type of absolute loader that plays a critical role in the initial startup sequence of a computer system. It is the very first program to be executed when the computer is first turned on or restarted (booted up).Its fundamental purpose is to load the operating system, or at least the initial part of it, into memory so that the computer can begin its normal operations.

Function and Design Principles

The concept of a bootstrap loader is rooted in simplicity and essential functionality, as it operates on a "bare machine" with minimal existing software support. The following text illustrates a very simple bootstrap loader for the **SIC/XE** architecture, which effectively demonstrates the core logic and coding techniques typically employed in any absolute loader.

Key characteristics and operational details of such a loader include:

- **Starting Point**: A bootstrap loader usually begins execution at a predefined, fixed address in memory (e.g., address 0 for the SIC/XE example). This fixed starting point is crucial because, at boot time, there's no operating system or complex loader in place to determine where to begin execution. The hardware is configured to transfer control to this specific address upon power-on or reset.
- **Loading the Initial Program**: Its primary task is to load the initial program, typically the operating system, into a specific memory location.[5] In the SIC/XE example, the bootstrap loader loads the operating system starting at address 80 (hexadecimal).
- **Simple Format**: For simplicity and efficiency, especially given its early execution phase, the program being loaded by a bootstrap loader often lacks complex control information like Header Records or End Records found in more general object files. The object code is usually loaded into consecutive memory locations without needing intricate parsing. In the SIC/XE example, a single byte of object code is represented as two hexadecimal digits on an input device (F1), without any other control information.
- **Direct Execution Transfer**: After all the necessary data (the operating system or its initial part) has been loaded from the input device, the bootstrap loader transfers control directly to the starting address of the newly loaded program, initiating its execution. For the SIC/XE example, it jumps to address 80 to start the loaded program.

Example: SIC/XE Bootstrap Loader

The SIC/XE bootstrap loader example highlights its concise nature and dependence on a core subroutine:

- **Initialization**: The loader initializes a register (e.g., register X) to the starting memory address where the operating system will be loaded (e.g., HEX 80). This register then keeps track of the next memory location to be filled.
- **Input Handling (GETC Subroutine)**: Most of the heavy lifting is often delegated to a specialized subroutine, like GETC in the SIC/XE example. This subroutine is responsible for:
    - Reading a character from a designated input device (e.g., device F1).
    - Converting the ASCII character code of the input (e.g., hexadecimal digits '0'-'9', 'A'-'F') into its corresponding numeric hexadecimal value. This involves subtracting constants (e.g., 48 for '0'-'9', 55 for 'A'-'F').
    - Ignoring irrelevant input characters (e.g., those with ASCII codes less than hexadecimal 30).
    - Handling end-of-file conditions, typically by transferring control to the starting address of the newly loaded program (e.g., jumping to address 80).
- **Byte Assembly and Storage**: The main loop of the bootstrap loader uses the GETC subroutine to read pairs of hexadecimal characters. It then combines these two characters to form a single byte of machine code (e.g., by shifting the first digit left by 4 bits and adding the second). This resulting byte is then stored at the memory address currently pointed to by the designated register (e.g., STCH 0,X).

- **Address Increment**: After storing a byte, the memory address register is incremented (e.g., using TIXR), preparing it for the next byte to be loaded. This loop continues until an end-of-file condition is detected from the input device.

For simplicity, such bootstrap loaders often lack sophisticated error checking and assume the input to be correct. However, in real-world scenarios, a robust loader would include mechanisms to handle various error conditions that might occur during the loading process.

Editors & Debuggers: Introduction to a Text Editor & Its Types

Introduction to a Text Editor

The **interactive text editor** has become an indispensable component of almost any modern computing environment. Its role has evolved beyond merely being a tool for programmers; it is now widely recognized as the primary interface to the computer for all types of "knowledge workers." These individuals utilize text editors to compose, organize, study, and manipulate computer-based information, making it a fundamental utility for interacting with digital content.

An **interactive editor** is defined as a computer program that empowers a user to create and revise a target document. The term "document" in this context is broad, encompassing a wide array of objects such as computer programs, plain text, mathematical equations, tables, diagrams, line art, and photographs—essentially, anything that might appear on a printed page. Specifically, a **text editor** is a program where the primary elements manipulated and edited are the character strings that constitute the target text.

The **document-editing process** is fundamentally an interactive dialogue between the user and the computer, designed to accomplish four core tasks:

1. **Selection**: The user must be able to select the specific part of the target document that they wish to view and manipulate. This involves "traveling" through the document to locate the area of interest using operations like "next screenful," "bottom," or "find pattern." Once the area is located, "filtering" extracts the relevant subset, such as the next screenful of text or the next statement, for viewing and manipulation.
2. **Formatting and Display**: After selection, the system determines how this filtered view will be formatted for on-line presentation and how it will be displayed on a screen or other output device.
3. **Modification**: This is the actual **editing phase**, where the target document is created or altered. A set of fundamental operations is provided for this purpose, including insert, delete, replace, move, and copy. The functionalities of these editing operations are often specialized to operate on elements that are meaningful to the specific type of editor being used. For instance, a manuscript-oriented editor might operate on elements like single characters, words, lines, sentences, and paragraphs. Conversely, a program-oriented editor would typically operate on elements such as identifiers, keywords, and complete statements, reflecting the structured nature of code.
4. **View Update**: Finally, the system must appropriately update the displayed view to reflect the changes made during the editing phase. This ensures that the user always sees an accurate representation of the document's current state.

Types of Text Editors

Text editors come in various forms, each designed with different scopes of operation and user interaction models:

1. **Line Editors**:
    - In a line editor, the scope of edit operations is strictly confined to a single line of text.
    - A line is typically designated either positionally (e.g., by specifying its serial number within the text) or contextually (e.g., by specifying a unique string or pattern that identifies it).
    - The primary advantage of line editors lies in their simplicity, making them straightforward to implement and use for basic text manipulation.
2. **Stream Editors**:
    - A stream editor conceptualizes the entire text as a continuous stream of characters. This fundamental view allows edit operations to transcend individual line boundaries, providing greater flexibility than line editors.
    - These editors support commands that are oriented around characters, lines, and contexts, with operations often based on the current editing context, which is indicated by the position of a text pointer.
    - This text pointer can be manipulated using positioning commands (e.g., move forward/backward by a certain number of characters) or search commands (e.g., find a specific string).
    - Both line and stream editors maintain multiple representations of the text: a "display form" that shows the text as a sequence of lines, and an "internal form" used for performing edit operations. The internal form often includes special characters like end-of-line markers and other edit characters. The editor ensures that these two representations remain compatible and synchronized at all times.
3. **Screen Editors**:
    - Unlike line or stream editors, which do not necessarily display text in its final printed appearance, a screen editor adheres to the "what-you-see-is-what-you-get" (WYSIWYG) principle.
    - The editor displays a full "screenful" of text at a single instant, allowing the user to interact directly with the visible content.
    - Users can move a cursor anywhere on the screen, position it precisely at the desired point for editing, and then proceed with modifications directly.
    - A key advantage of screen editors is the immediate visual feedback; users can instantly see the effect of an edit operation on the screen, which is particularly useful for formatting text to produce printed documents.
4. **Word Processors**:

  - Word processors are specialized document editors that extend beyond basic text editing by incorporating additional sophisticated features designed to produce well-formatted hard copy outputs.
  - Essential features commonly found in word processors include commands for moving entire sections of text from one location to another, merging different text segments, and advanced search and replacement functionalities.
  - Many modern word processors also support supplementary features such as spell-checking, grammar-checking, and layout tools.
  - These editors are extensively used by a wide range of professionals, including authors, office personnel, and computer professionals, for creating polished documents.

5. **Structure Editors**:
    - A structure editor is distinct in that it incorporates an inherent awareness of the underlying structure of the document it is editing. This awareness is based on a predefined or user-specified grammar or schema for the document.
    - This structural understanding is highly beneficial for tasks such as browsing through a document; for example, a programmer can easily navigate to and edit a specific function within a program file by leveraging the editor's knowledge of the program's structure.
    - The structure is typically specified by the user during the creation or modification of the document. Editing requirements are then specified and performed in relation to this defined structure.
    - A specialized class of structure editors, known as **syntax-directed editors**, are particularly prevalent and useful in programming environments, where they understand the syntax of programming languages and can assist in writing syntactically correct code.

Interactive Debugging Systems

An **interactive debugging system** is a sophisticated software tool designed to provide programmers with comprehensive facilities that significantly aid in the testing and debugging of programs. These systems are crucial for identifying, diagnosing, and resolving errors within software, thereby improving program reliability and development efficiency.

Interactive debugging systems typically offer the following core facilities:

1. **Setting Breakpoints**: Programmers can define specific points in their program, known as breakpoints, where execution will be temporarily suspended.
2. **Initiating Debug Conversations**: When control reaches a breakpoint and execution is suspended, the debugging system initiates a "debug conversation," allowing the programmer to interact with the program's state.
3. **Displaying Variable Values**: During a debug conversation, programmers can inspect and display the current values of program variables.

4. **Assigning New Variable Values**: Debuggers allow programmers to dynamically assign new values to variables, enabling them to test different scenarios or correct erroneous states during execution.

5. **Testing Assertions and Predicates**: Programmers can define user-defined assertions and predicates (logical conditions) involving program variables. The debugging system can continuously evaluate these conditions, suspending execution when any of them become true.

Debugging Functions and Capabilities

To fulfill its purpose, a debugging system must possess a range of powerful capabilities:

1. **Execution Sequencing**: A fundamental requirement is the ability to observe and control the flow of program execution. This involves providing unit test functions that allow programmers to step through code, run to specific points, or execute sections.

2. **Breakpoints**: As mentioned, breakpoints are key. They cause program execution to suspend at a specified point. Once suspended, various debugging commands can be invoked to analyze the program's progress and diagnose detected errors. After analysis, execution can be resumed from the breakpoint.

3. **Conditional Expressions**: Beyond simple breakpoints, programmers can define conditional expressions that are continuously evaluated by the debugger during the debugging session. Program execution is suspended automatically when any of these conditions evaluates to true, allowing for targeted debugging of complex logic.

4. **Tracing**: Tracing is used to track the flow of execution logic and monitor data modifications. Control flow can be traced at different levels of detail, such as at the procedure level, branch level, or even individual instruction level. Tracing can also be made conditional, activating only when specific expressions are met.

5. **Traceback**: The traceback capability provides a historical view of the program's execution path, showing how the current statement was reached. It can also indicate which statements were responsible for the modification of a given variable or parameter, which is invaluable for understanding data flow and pinpointing the source of errors.

6. **Program Display**: A robust debugging system must offer excellent program-display capabilities. This includes the ability to display the program being debugged, complete with statement numbers, and allowing the user to control the level of detail in this display. It is also highly beneficial for the system to support modification and incremental recompilation of the program *during* a debugging session, allowing for quick testing of fixes without restarting the entire process. Furthermore, the system should save all debugging specifications (e.g., breakpoints, traces) so that the programmer does not need to re-issue them in subsequent sessions. The ability to symbolically display or modify the contents of any variables and constants in the program, and then resume execution, is also a crucial feature.

Multi-Language Debugging and Debugger Context

For a debugging system to be effective in modern user environments and application systems, which often utilize several different programming languages, it must be a **multi-language debugging tool**. The commands used to initiate actions and gather data about a program's execution should ideally be common across all supported languages, providing a consistent user experience.

These requirements have significant implications for the debugger and other system software:

- **Temporary Suspension**: When control is transferred to the debugger, the execution of the program being debugged is temporarily suspended.
- **Language Determination and Context Setting**: The debugger must be capable of determining the language in which the program (or the current segment) is written and setting its context accordingly. This ensures that language-specific rules for symbol resolution, data interpretation, and instruction execution are correctly applied.
- **Context Switching**: The debugger should also be able to seamlessly switch its context when a program written in a different language is called from the currently executing program. Such context changes should be clearly communicated to the user to prevent confusion.

While the notations for various debugger functions may vary with the language of the program being debugged, the underlying functions are accomplished in a similar manner across programming languages. For the debugger to fully complete its functions, it must have access to information gathered by the language translator, such as the internal symbol table formats. Future compilers and assemblers are expected to provide consistent interfaces with the debugging system to facilitate this information exchange.

Dynamic Debugging Sequence

1. **Program Compilation with Debug Option**: The user first compiles the program with the debug option enabled. This process generates two key outputs: the compiled code file (the executable logic) and a debug information file (containing symbols, line numbers, etc., necessary for debugging).
2. **Debugger Activation**: The user then activates the debugging system and specifies the name of the program they intend to debug. In response, the debugging system opens both the compiled code file and the associated debug information file.
3. **Breakpoint Specification**: The user defines their debugging requirements, notably a list of breakpoints. The debugger then "instruments" the program by inserting necessary code or data structures (like a debug table containing pairs of statement numbers and corresponding debug actions) to manage these breakpoints.
4. **Instrumental Program Control**: When the program's execution reaches a breakpoint, an instrumental program (part of the debugger's runtime component) generates a software interrupt. This interrupt transfers control to the debugging system. The debugger then consults its debug table and performs the predefined debug actions for that specific breakpoint. At this point, a "debug conversation" is initiated, allowing the user to interact with the program, issue commands, or modify breakpoints and other debug actions.
5. **Execution Resumption**: After the user has completed their analysis or modifications within the debug conversation, control is returned to the instrumented program, allowing its execution to resume from the point of suspension.

6. **Looping**: Steps 4 and 5 are repeated iteratively until the entire debugging session is concluded, or the program terminates.

Relationship with Other Parts of the System

An interactive debugger does not operate in isolation; it needs to be closely integrated with other parts of the system to function effectively. This relationship is characterized by:

1. **Availability**: It is an important requirement for an interactive debugger to be always available and to function as an integral part of the system's run-time environment. Debugging becomes impossible if program failures cannot be reproduced under the debugger's control.
2. **Consistency**: There must be consistency between the debugging system and the source code, object code, and execution environment. Any changes to the source code must be consistent with the object code, and the debugger should also use the same symbol table formats generated by the compiler.
3. **Coordination**: Effective debugging necessitates proper coordination between the various system components. The debugger must be able to interact with the operating system for process control, with the compiler/assembler for symbol information, and potentially with loaders/linkers for understanding memory layouts.
