Chapter 3 – Loaders and Linkers

- Three fundamental processes:

  * **Loading** – brings the object program into memory for execution.
  * **Relocation** – modifies the object program so that it can be loaded at an address different from the location originally specified.
  * **Linking** – combines two or more separate object programs and supplies the information needed to allow references between them.

- A *loader* is a system program that performs the **loading** function. Many loaders also support **relocation** and **linking**. Some systems have a *linker* to perform the **linking** operations and a separate loader to handle **relocation** and **loading**.

### 3.1 Introduction

- The most fundamental functions of a loader – bringing an object program into memory and starting its execution.

#### 3.1.1 Design of an Absolute Loader

- An example object program is shown in Fig 3.1(a).
For a simple absolute loader, all functions are accomplished in a single pass as follows:

1) The **Header record** of object programs is checked to verify that the correct program has been presented for loading.

2) As each **Text record** is read, the object code it contains is moved to the indicated address in memory.

3) When the **End record** is encountered, the loader jumps to the specified address to begin execution of the loaded program.

Fig 3.1(b) shows a representation of the program from Fig 3.1(a) after loading.

Fig 3.2 shows an algorithm for the absolute loader we have discussed.
It is very important to realize that in Fig 3.1(a), each printed character represents one byte of the object program record. In Fig 3.1(b), on the other hand, each printed character represents one hexadecimal digit in memory (a half-byte).

Therefore, to save space and execution time of loaders, most machines store object programs in a binary form, with each byte of object code stored as a single byte in the object program.

3.1.2 A Simple Bootstrap Loader

When a computer is first turned on or restarted, a special type of absolute loader, called a bootstrap loader, is executed. This bootstrap loads the first program to be run by the computer – usually an operating system.

Fig 3.3 shows the source code for our bootstrap loader. The bootstrap itself begins at address 0 in the memory.
Note: each byte of object code to be loaded is represented on device F1 as two hexadecimal digits just as it is in a Text record of a SIC object program.

1) The object code from device F1 is always loaded into consecutive bytes of memory, starting at address 80. The
main loop of the bootstrap keeps the address of the next memory location to be loaded in register X.

2) After all of the object code from device F1 has been loaded, the bootstrap jumps to address 80, which begins the execution of the program that was loaded.

- Much of the work of the bootstrap loader is performed by the subroutine GETC.

GETC is used to read and convert a pair of characters from device F1 representing 1 byte of object code to be loaded. For example, two bytes =C“D8”Æ‘4438’H converting to one byte ‘D8’H.

The resulting byte is stored at the address currently in register X, using STCH instruction that refers to location 0 using indexed addressing.

The TIXR instruction is then used to add 1 to the value in X.

3.2 Machine-Dependent Loader Features

- The absolute loader has several potential disadvantages.

  One of the most obvious is the need for the programmer to specify the actual address at which it will be loaded into memory.

  Writing absolute programs also makes it difficult to use subroutine libraries efficiently. This could not be done effectively if all of the subroutines had pre-assigned absolute addresses.

- The need for program relocation is an indirect consequence of the change to larger and more powerful computers. The way relocation is implemented in a loader is also dependent upon machine characteristics.
3.2.1 Relocation

- Two methods for specifying relocation as part of the object program.

- *The first method:* A Modification record (The format is given in Section 2.3.5.) is used to describe each part of the object code that must be changed when the program is relocated.

- Fig 3.4 shows a SIC/XE program we use to illustrate this first method of specifying relocation.
### Figure 3.4 Example of a SIC/XE program (from Fig. 2.6)

<table>
<thead>
<tr>
<th>Line</th>
<th>Loc</th>
<th>Source statement</th>
<th>Object code</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td>0000</td>
<td>COPY START 0</td>
<td>17202D</td>
</tr>
<tr>
<td>10</td>
<td>0000</td>
<td>FIRST STL RETADR</td>
<td>69202D</td>
</tr>
<tr>
<td>12</td>
<td>0003</td>
<td>LDB #LENGTH</td>
<td></td>
</tr>
<tr>
<td>13</td>
<td></td>
<td>BASE LENGTH</td>
<td></td>
</tr>
<tr>
<td>15</td>
<td>0006</td>
<td>CLOOP +JSUB RDREC</td>
<td>4B101036</td>
</tr>
<tr>
<td>20</td>
<td>000A</td>
<td>LDA LENGTH</td>
<td>032026</td>
</tr>
<tr>
<td>25</td>
<td>000D</td>
<td>COMP #0</td>
<td>250000</td>
</tr>
<tr>
<td>30</td>
<td>0010</td>
<td>JEQ ENDFIL</td>
<td>332007</td>
</tr>
<tr>
<td>35</td>
<td>0013</td>
<td>+JSUB WRREC</td>
<td>4B10105D</td>
</tr>
<tr>
<td>40</td>
<td>0017</td>
<td>J CLOOP</td>
<td>3F2FBC</td>
</tr>
<tr>
<td>45</td>
<td>001A</td>
<td>ENDFIL</td>
<td>3F2FBC</td>
</tr>
<tr>
<td>50</td>
<td>001D</td>
<td>LDA EOF</td>
<td>032010</td>
</tr>
<tr>
<td>55</td>
<td>0020</td>
<td>STA BUFFER</td>
<td>0F2016</td>
</tr>
<tr>
<td>60</td>
<td>0023</td>
<td>LDA #3</td>
<td>010003</td>
</tr>
<tr>
<td>65</td>
<td>0026</td>
<td>STA LENGTH</td>
<td>08200D</td>
</tr>
<tr>
<td>70</td>
<td>002A</td>
<td>J @RETADR</td>
<td>3B2003</td>
</tr>
<tr>
<td>80</td>
<td>002D</td>
<td>EOF BYTE C'ECF'</td>
<td>454F46</td>
</tr>
<tr>
<td>95</td>
<td>0030</td>
<td>RETADR RESW 1</td>
<td></td>
</tr>
<tr>
<td>100</td>
<td>0033</td>
<td>LENGTH RESW 1</td>
<td></td>
</tr>
<tr>
<td>105</td>
<td>0036</td>
<td>BUFFER RESB 4096</td>
<td></td>
</tr>
</tbody>
</table>

**SUBROUTINE TO READ RECORD INTO BUFFER**

<table>
<thead>
<tr>
<th>Line</th>
<th>Loc</th>
<th>Source statement</th>
<th>Object code</th>
</tr>
</thead>
<tbody>
<tr>
<td>125</td>
<td>1036</td>
<td>RDREC CLEAR X</td>
<td>B410</td>
</tr>
<tr>
<td>130</td>
<td>1038</td>
<td>CLEAR A</td>
<td>B400</td>
</tr>
<tr>
<td>132</td>
<td>103A</td>
<td>CLEAR S</td>
<td>B440</td>
</tr>
<tr>
<td>133</td>
<td>103C</td>
<td>+LDT #4096</td>
<td>75101000</td>
</tr>
<tr>
<td>135</td>
<td>1040</td>
<td>RLOOP TD INPUT</td>
<td>E32019</td>
</tr>
<tr>
<td>140</td>
<td>1043</td>
<td>JEQ RLOOP</td>
<td>3320PA</td>
</tr>
<tr>
<td>145</td>
<td>1046</td>
<td>RD INPUT</td>
<td>DB2013</td>
</tr>
<tr>
<td>150</td>
<td>1049</td>
<td>COMPR A,S</td>
<td>A004</td>
</tr>
<tr>
<td>155</td>
<td>104B</td>
<td>JEQ EXIT</td>
<td>332008</td>
</tr>
<tr>
<td>160</td>
<td>104E</td>
<td>STCH BUFFER,X</td>
<td>57C003</td>
</tr>
<tr>
<td>165</td>
<td>1051</td>
<td>TIXR T</td>
<td>B850</td>
</tr>
<tr>
<td>170</td>
<td>1053</td>
<td>EXIT STX LENGTH</td>
<td>134000</td>
</tr>
<tr>
<td>175</td>
<td>1056</td>
<td>INPUT RSUB</td>
<td>4F0000</td>
</tr>
<tr>
<td>180</td>
<td>1059</td>
<td>BYTE X'F1'</td>
<td>F1</td>
</tr>
<tr>
<td>185</td>
<td>105C</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**SUBROUTINE TO WRITE RECORD FROM BUFFER**

<table>
<thead>
<tr>
<th>Line</th>
<th>Loc</th>
<th>Source statement</th>
<th>Object code</th>
</tr>
</thead>
<tbody>
<tr>
<td>210</td>
<td>105D</td>
<td>WRREC CLEAR X</td>
<td>B410</td>
</tr>
<tr>
<td>212</td>
<td>105F</td>
<td>LDT LENGTH</td>
<td>774000</td>
</tr>
<tr>
<td>215</td>
<td>1062</td>
<td>WLOOP TD OUTPUT</td>
<td>E32011</td>
</tr>
<tr>
<td>220</td>
<td>1065</td>
<td>JEQ WLOOP</td>
<td>3320PA</td>
</tr>
<tr>
<td>225</td>
<td>1068</td>
<td>LDXCH BUFFER,X</td>
<td>53C003</td>
</tr>
<tr>
<td>230</td>
<td>106B</td>
<td>WD OUTPUT</td>
<td>DF2008</td>
</tr>
<tr>
<td>235</td>
<td>106E</td>
<td>TIXR T</td>
<td>B850</td>
</tr>
<tr>
<td>240</td>
<td>1070</td>
<td>JLT WLOOP</td>
<td>3B2F3A</td>
</tr>
<tr>
<td>245</td>
<td>1073</td>
<td>RSUBkehr</td>
<td>4F0000</td>
</tr>
<tr>
<td>250</td>
<td>1076</td>
<td>OUTPUT BYTE X'05'</td>
<td>05</td>
</tr>
<tr>
<td>255</td>
<td></td>
<td>END FIRST</td>
<td></td>
</tr>
</tbody>
</table>
Most of the instructions in this program use *relative* or *immediate* addressing.

The only portions of the assembled program that contain actual addresses are the extended format instructions on lines 15, 35, and 65. Thus these are the only items whose values are affected by relocation.

- Fig 3.5 displays the object program corresponding to the source in Fig 3.4.

![Figure 3.5 Object program with relocation by Modification records.](image)

Each Modification record specifies the starting address and length of the field whose value is to be altered.

It then describes the modification to be performed.

In this example, all modifications add the value of the symbol COPY, which represents the starting address of the program.

- The Modification record is not well suited for use with all machine architectures. Consider, for example, the program in Fig 3.6. This is a *relocatable* program written for standard version for SIC.
The important difference between this example and the one in Fig 3.4 is that the standard SIC machine does not use relative addressing.

In this program the addresses in all the instructions except RSUB must modified when the program is
relocated. This would require 31 Modification records, which results in an object program more than twice as large as the one in Fig 3.5.

- The second method: Fig 3.7 shows this method applied to our SIC program example.

![Figure 3.7 Object program with relocation by bit mask.]

There are no Modification records.

The Text records are the same as before except that there is a relocation bit associated with each word of object code.

Since all SIC instructions occupy one word, this means that there is one relocation bit for each possible instruction.

- The relocation bits are gathered together into a **bit mask** following the length indicator in each Text record. In Fig 3.7 this mask is represented (in character form) as three hexadecimal digits.

- If the relocation bit corresponding to a word of object code is set to 1, the program’s starting address is to be added to this word when the program is relocated.

  A bit value of 0 indicates that no modification is necessary.

  If a Text record contains fewer than 12 words of object code, the bits corresponding to unused words are set to 0.
For example, the bit mask FFC (representing the bit string 111111111100) in the first Text record specifies that all 10 words of object code are to be modified during relocation.

- Example: note that the LDX instruction on line 210 (Fig 3.6) begins a new Text record. If it were placed in the preceding Text record, it would not be properly aligned to correspond to a relocation bit because of the 1-byte data value generated from line 185.

### 3.2.2 Program Linking

- Consider the three (separately assembled) programs in Fig 3.8, each of which consists of a single control section.
Consider first the reference marked REF1.

For the first program (PROGA), (1) REF1 is simply a reference to a label within the program. (2) It is assembled in the usual way as a PC relative instruction. (3) No modification for relocation or linking is necessary.

In PROGB, the same operand refers to an external symbol. (1) The assembler uses an extended-format instruction with address field set to 00000. (2) The object program for PROGB (Fig 3.9) contains a Modification record instructing the loader to add the value of the symbol LISTA to this address field when the program is linked.

![Figure 3.8 Sample programs illustrating linking and relocation.](image-url)
For PROGC, REF1 is handled in exactly the same way.

- The reference marked REF2 is processed in a similar manner.

- REF3 is an immediate operand whose value is to be the difference between ENDA and LISTA (that is, the length of the list in bytes).

In PROGA, the assembler has all of the information necessary to compute this value.

During the assembly of PROGB (and PROGC), the values of the labels are unknown. In these programs, the expression must be assembled as an external reference (with two Modification records) even though the final result will be an absolute value independent of the locations at which the programs are loaded.

- Consider REF4.

The assembler for PROGA can evaluate all of the
expression in REF4 except for the value of LISTC. This results in an initial value of ‘000014’H and one Modification record.

The same expression in PROGB contains no terms that can be evaluated by the assembler. The object code therefore contains an initial value of 000000 and three Modification records.

For PROGC, the assembler can supply the value of LISTC relative to the beginning of the program (but not the actual address, which is not known until the program is loaded). The initial value of this data word contains the relative address of LISTC (‘000030’H). Modification records instruct the loader to add the beginning address of the program (i.e., the value of PROGC), to add the value of ENDA, and to subtract the value of LISTA.

- Fig 3.10(a) shows these three programs as they might appear in memory after loading and linking. PROGA has been loaded starting at address 4000, with PROGB and PROGC immediately following.
For example, the value for reference REF4 in PROGA is located at address 4054 (the beginning address of PROGA plus 0054). Fig 3.10(b) shows the details of how this value is computed.

The initial value (from the Text record) is 000014. To this is added the address assigned to LISTC, which 4112 (the beginning address of PROGC plus 30).

3.2.3 Algorithm and Data Structures for a Linking Loader

- The algorithm for a *linking loader* is considerably more complicated than the *absolute loader* algorithm discussed in Section 3.1.

- A linking loader usually makes two passes over its input, just as an assembler does. In terms of general function,
the two passes of a linking loader are quite similar to the two passes of an assembler:

Pass 1 assigns addresses to all external symbols.
Pass 2 performs the actual loading, relocation, and linking.

- The main data structure needed for our linking loader is an external symbol table ESTAB.

This table, which is analogous to SYMTAB in our assembler algorithm, is used to store the name and address of each external symbol in the set of control sections being loaded.

A hashed organization is typically used for this table.

Two other important variables are PROGADDR (program load address) and CSADDR (control section address).

PROGADDR is the beginning address in memory where the linked program is to be loaded. Its value is supplied to the loader by the OS.

CSADDR contains the starting address assigned to the control section currently being scanned by the loader. This value is added to all relative addresses within the control section to convert them to actual addresses.

- The algorithm is presented in Fig 3.11.

- During Pass 1 (Fig 3.11(a)), the loader is concerned only with Header and Define record types in the control sections.
1) The beginning load address for the linked program (PROGADDR) is obtained from the OS. This becomes the starting address (CSADDR) for the first control section in the input sequence.

2) The control section name from **Header record** is entered into ESTAB, with value given by CSADDR. All **external symbols** appearing in the Define record for the control section are also entered into ESTAB. Their addresses are obtained by adding the value specified in the **Define record** to CSADDR.

3) When the **End record** is read, the control section length
CSLTH (which was saved from the End record) is added to CSADDR. This calculation gives the starting address for the next control section in sequence.

- At the end of Pass 1, ESTAB contains all external symbols defined in the set of control sections together with the address assigned to each.

- Many loaders include as an option the ability to print a load map that shows these symbols and their addresses. For the example of Figs 3.9 and 3.10, such a load map might look like as shown on the top of page 143.

- Pass 2 (Fig 3.11(b)) of our loader performs the actual loading, relocation, and linking of the program.

```
Pass 2:

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
                if record type = 'T' then
                    begin
                        (if object code is in character form, convert
                        into internal representation)
                        move object code from record to location
                        (CSADDR + specified address)
                    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 address)
                        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 BOP)
            end (begin)
        end (while)
    end (begin)
jump to location given by EXECADDR (to start execution of loaded program)
end (Pass 2)
```

*Figure 3.11(b)* Algorithm for Pass 2 of a linking loader.
1) As each Text record is read, the object code is moved to the specified address (plus the current value of CSADDR).

2) When a Modification record is encountered, the symbol whose value is to be used for modification is looked up in ESTAB.

3) This value is then added to or subtracted from the indicated location in memory.

4) The last step performed by the loader is usually the transferring of control to the loaded program to begin execution.

- The End record for each control section may contain the address of the first instruction in that control section to be executed. Our loader takes this as the transfer point to begin execution.

If more than one control section specifies a transfer address, the loader arbitrarily uses the last one encountered.

If no control section contains a transfer address, the loader uses the beginning of the linked program (i.e., PROGADDR) as the transfer point.

Normally, a transfer address would be placed in the End record for a main program, but not for a subroutine.

- This algorithm can be made more efficient. Assign a reference number, which is used (instead of the symbol name) in Modification records, to each external symbol referred to in a control section.

Suppose we always assign the reference number 01 to the control section name.

Fig 3.12 shows the object programs from 3.9 with this
change.

Figure 3.12 Object programs corresponding to Fig. 3.8 using reference numbers for code modification. (Reference numbers are underlined for easier reading.)
3.3 Machine-Independent Loader Features

- Loading and linking are often thought of as OS service functions. Therefore, most loaders include fewer different features than are found in a typical assembler. They include the use of an automatic library search process for handling external reference and some common options that can be selected at the time of loading and linking.

3.3.1 Automatic Library Search

- Many linking loaders can automatically incorporate routines from a subprogram library into the program being loaded.
- Linking loaders that support *automatic library search* must keep track of external symbols that are referred to, but not defined, in the primary input to the loader.
- At the end of Pass 1, the symbols in ESTAB that remain undefined represent unresolved external references.

The loader searches the library or libraries specified for routines that contain the definitions of these symbols, and processes the subroutines found by this search exactly as if they had been part of the primary input stream.

Note that the subroutines fetched from a library in this way may themselves contain external references. It is therefore necessary to repeat the library search process until all references are resolved.

If unresolved external references remain after the library search is completed, these must be treated as errors.

3.3.2 Loader Options

- Many loaders allow the user to specify options that modify the standard processing described in Section 3.2.
• Typical loader option 1: allows the selection of alternative sources of input. Ex.,

    INCLUDE    program-name (library-name)

might direct the loader to read the designated object program from a library and treat it as if it were part of the primary loader input.

• Loader option 2: allows the user to delete external symbols or entire control sections. Ex.,

    DELETE    csect-name

might instruct the loader to delete the named control section(s) from the set of programs being loaded.

    CHANGE    name1, name2

might cause the external symbol name1 to be changed to name2 wherever it appears in the object programs.

• Loader option 3: involves the automatic inclusion of library routines to satisfy external references. Ex.,

    LIBRARY    MYLIB

Such user-specified libraries are normally searched before the standard system libraries. This allows the user to use special versions of the standard routines.

    NOCALL    STDDEV, PLOT, CORREL

To instruct the loader that these external references are to remain unresolved. This avoids the overhead of loading and linking the unneeded routines, and saves the memory space that would otherwise be required.

3.4  Loader Design Options

• Linking loaders perform all linking and relocation at load
time. There are two alternatives: **Linkage editors**, which perform linking prior to load time, and **dynamic linking**, in which the linking function is performed at execution time.

- Precondition: The source program is first assembled or compiled, producing an object program.

  A *linking loader* performs all linking and relocation operations, including automatic library search if specified, and loads the linked program directly into memory for execution.

  A *linkage editor* produces a linked version of the program (load module or executable image), which is written to a file or library for later execution.

- The essential difference between a *linkage editor* and a *linking loader* is illustrated in Fig 3.13.

![Diagram of linking loader and linkage editor](image_url)
3.4.1 Linkage Editors

- The *linkage editor* performs relocation of all control sections relative to *the start of the linked program*. Thus, all items that need to be modified at load time have values that are relative to the start of the linked program.

This means that the loading can be accomplished in one pass with no external symbol table required.

If a program is to be executed many times without being reassembled, the use of a linkage editor substantially reduces the overhead required.

- Linkage editors can perform many useful functions besides simply preparing an object program for execution. Ex., a typical sequence of linkage editor commands used:
  
  INCLUDE PLANNER (PROGLIB)
  DELETE PROJECT {delete from existing PLANNER}
  INCLUDE PROJECT (NEWLIB) {include new version}
  REPLACE PLANNER (PROGLIB)

- Linkage editors can also be used to build packages of subroutines or other control sections that are generally used together. This can be useful when dealing with subroutine libraries that support high-level programming languages.

- Linkage editors often include a variety of other options and commands like those discussed for linking loaders. Compared to linking loaders, *linkage editors in general* tend to offer more flexibility and control.

3.4.2 Dynamic Linking

- *Linkage editors* perform linking operations before the program is loaded for execution.

  *Linking loaders* perform these same operations at load
Dynamic linking, dynamic loading, or load on call postpones the linking function until execution time: a subroutine is loaded and linked to the rest of the program when it is first called.

- Dynamic linking is often used to allow several executing programs to share one copy of a subroutine or library, ex. run-time support routines for a high-level language like C.

- With a program that allows its user to interactively call any of the subroutines of a large mathematical and statistical library, all of the library subroutines could potentially be needed, but only a few will actually be used in any one execution.

  Dynamic linking can avoid the necessity of loading the entire library for each execution except those necessary subroutines.

- Fig 3.14 illustrates a method in which routines that are to be dynamically loaded must be called via an OS service request.
Figure 3.14 Loading and calling of a subroutine using dynamic linking.
Fig 3.14(a): Instead of executing a JSUB instruction referring to an external symbol, the program makes a load-and-call service request to OS. The parameter of this request is the symbolic name of the routine to be called.

Fig 3.14(b): OS examines its internal tables to determine whether or not the routine is already loaded. If necessary, the routine is loaded from the specified user or system libraries.

Fig 3.14(c): Control is then passed from OS to the routine being called

Fig 3.14(d): When the called subroutine completes its processing, it returns to its caller (i.e., OS). OS then returns control to the program that issued the request.

Fig 3.14(e): If a subroutine is still in memory, a second call to it may not require another load operation. Control may simply be passed from the dynamic loader to the called routine.

### 3.4.3 Bootstrap Loaders

- Given an idle computer with no program in memory, how do we get things started?
- On some computers, an absolute loader program is permanently resident in a read-only memory (ROM). When some hardware signal occurs, the machine begins to execute this ROM program. This is referred to as a bootstrap loader.

### 3.5 Implementation Examples

(Skip)