CHAPTER 1

INTRODUCTION

1.1 Loaders

In computer systems a loader is the part of an operating system that is responsible for loading programs and libraries. It is one of the essential stages in the process of starting a program, as it places programs into memory and prepares them for execution. Loading a program involves reading the contents of the executable file containing the program instructions into memory, and then carrying out other required preparatory tasks to prepare the executable for running. Once loading is complete, the operating system starts the program by passing control to the loaded program code.

All operating systems that support program loading have loaders, apart from highly specialized computer systems that only have a fixed set of specialized programs. Embedded systems typically do not have loaders, and instead the code executes directly from ROM. In order to load the operating system itself, as part of booting, a specialized boot loader is used. In many operating systems the loader is permanently resident in memory, although some operating systems that support virtual memory may allow the loader to be located in a region of memory that is pageable.

The functions of the loader are to allocate the space for program in the memory, to resolve the symbolic references between the object modules by assigning all the user subroutine and library subroutine addresses, to perform relocation for address dependent instructions and to place all the machine instructions into the memory.

1.2 Types of Loaders

There are several different types of loaders. They are classified into absolute loader and relocatable loader. By making the loader more sophisticated, more complex functions can be done by the loader, resulting in even better utilization of the computer.

1.2.1 Absolute Loader

An absolute loader is the simplest of loaders. Its function is simply to take the output of the assembler and load it into memory. The output of the assembler can be stored on any machine-readable form of storage, but most commonly it is stored on punched cards or magnetic tape, disk, or drum.

This Loader does not need to perform functions such as linking and program relocation, its operation is very simple. All functions are accomplished in a single pass. The Header record is checked to verify that the correct program has been presented for lading. As each text record is read, the object code it contains is moved to the indicated address in memory when the end record is encountered, the loader jumps to the specified address to begin execution of the loaded program.

Absolute loaders have a number of advantages: they are small, fast and simple. But they have a number of disadvantages, too. The major problem deals with the need to assemble an entire program all at once. Since the addresses for the program are determined at assembly time, the entire program must be assembled at one time in order for proper addresses to be assigned to the different parts. This means that a small change to one subroutine requires reassembly of the entire program. Also, standard subroutines, which might be kept in a library of useful subroutines and functions, must be physically copied and added to each program which uses them

1.2.2 Relocatable Loader

Loaders that allow for program relocation are called *relocating loaders* or *relative loaders*. There are two methods for specifying relocation as part of the object program.

In first method, a Modification record is used to describe each part of the object code that must be changed when the program is relocated. If the portions in assembled program use relative or immediate addressing then their values will not affected by relocation, only values of those portions that contain actual addresses (extended format instructions) are affected. There is one Modification record for each value that must be changed during relocation. Each Modification record specifies the starting address and length of the field whose value is to be altered. Ot then describes the modification to be performed.

The Modification record scheme is a convenient means for specifying program relocation; however, it is not well suited for use with all machine architectures. On a machine that primarily uses direct addressing and has a fixed instruction format, it is often more efficient to specify relocation using a different technique.

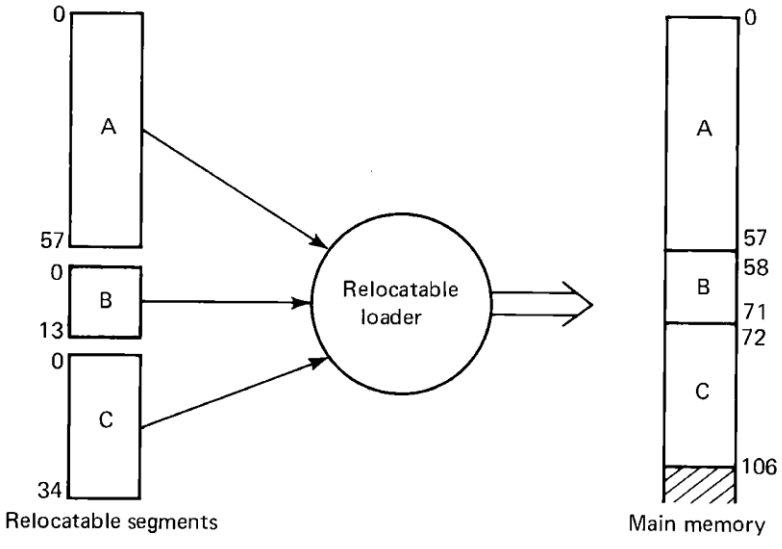

Figure1.1: Relocatable loader

CHAPTER 2

SYSTEM REQUIREMENTS

Software Requirement Specification (SRS) is a complete description of the behaviour of the system to be developed. It includes the system requirements which consist of functional and non-functional requirements, user requirements and hardware and software requirements. The functional requirements indicate what functions the software should perform and non-functional requirements are related to overall properties such as performance and scalability. This section of SRS contain all the software requirements to a level of detail sufficient to enable designers to design a system to satisfy those requirements and it also help to design their test cases to verify them to check whether system satisfies those requirements or not.

2.1 Overall Description

The functions of the product, its intended users and details of its operating environment are listed below.

2.1.1 Product Perspective

The software performs loading after the generation of the machine code by the assembler. It performs various functions such as linking, loading, relocating in order to load the executable machine code into the memory. This software may form part of a longer pipeline. Therefore, it is evident that this software is made up of various phases and subsystems.

2.1.2 Product Functions

The software performs functions like relocation, linking and loading. Some SIC/XE instructions are needed to be modified by the loader when they are loaded into the starting address provided by the OS. Modification records are used to perform this change of address into the memory.

Relocation bits are used to perform the same task in SIC programs which consists of large number of instructions to be modified.

2.1.3 User Classes and Characteristics

The different classes of users who will this software include end-users who will use this software to load their files, and developers interested in improving the loader implementation or redefining the loader.

2.1.4 Operating Environment

The software will operate on Ubuntu 14.04 LTS 64 bit operating system. The hardware it is tested on includes an Intel core I5 processor and 4GB RAM. The loader will work in parallel with other programs which process the same input or which use the output of the loader to do further processing.

2.1.5 Design and Implementation Constraints

The loader must be efficient and robust to errors. There should be sufficient space in the memory for loading the object programs into it. Linking the libraries to the programs to be correctly so that no error is generated.

2.2 Hardware Requirements

The basic hardware requirements needed to be able to run the loader are listed below.

2.2.1 Processor Requirements

The program will run on very basic processors (even single core). However, it has been designed and tested on a computer using an Intel core I5 processor with 4 cores.

2.2.2 Memory Requirements

The software is designed on a system with 4 GB of RAM. It is recommended that at least 1 GB of RAM be present, so that all relevant data (external symbol table, input files, etc.) may be kept in memory simultaneously.

2.2.3 Disk Requirements

GCC (for the C compiler) requires at least 250 MB of disk space (including runtime libraries). In addition, the flex program, grammar rules and input files must also be stored, so a minimum of 1 GB disk space is recommended.

2.3 Software Requirements

Given below are the basic software requirements for the solution

2.3.1 Operating system

The operating system on which the application is built on a Windows 8.1 64-bit operating system.

2.3.2 C Compiler

A C compiler is needed to compile the external symbol table generator source code, written in C. We use the GCC compiler system which supports the C language. The code is written in C and requires a GCC compiler to be compiled.

2.4 Functional Requirements

The solution has separate components capable of performs many functions. The basic functions of the solution can be specified as follows.

2.4.1 Relocation of source code

The loader can load the code at the runtime dynamically taking into account the memory available at that point of time. The SIC/XE code has the property that the code addresses are relative to each other, thus our program must be able to handle that dynamic relocation of code.

2.4.2 Address modification for extended instructions

The loader should be able to handle the records which define the location of the extended instruction in the SIC/XE instruction set. These instructions should contain the absolute address instead of the relative address like the normal instruction. Since the assembler has no idea about the final location of the code, it provides a relative code for this instruction as well, which should be handled and made absolute.

2.4.3 Formatted output generation

The loader should be able to simulate the loading of program into the memory thus it should show the exact location of each instruction at the end of the program. This output file in HEX format should consider each byte of data that's written into the memory when the program is actually loaded into the memory.

2.5 Non-functional Requirements

The loader must fulfil the functional requirements listed above. In addition, the system must meet the required non-functional requirements, especially those regarding robustness and error-recovery. In addition, non-functional requirements related to performance and maintainability requirements should be met. These are described below.

2.5.1 Software Quality Attributes

The following qualities should be satisfied by the overall system.

2.5.1.1 Robustness

The loader should be able to recover from error and continue with its task of parsing the input. For example, panic mode recovery is a simple but robust method to recover for error-recovery. It is essential that the loader does not halt on seeing a single erroneous input sequence, but that it continues parsing so that more errors can be discovered and reported.

2.5.1.2 Maintainability

The system should be such that it is possible in the future to correct any defects or lacuna that may be discovered. This may necessitate the modification of productions or the expressions used in lexical analysis.

2.5.2 Performance Requirements

The loader must be able to perform well and load the given input at a reasonable rate. The process of loading the object programs may take longer as it is not necessary to do this for each run on the loader .Sufficient space should be available in the memory for the object programs to be loaded and it is allocated by the OS.

CHAPTER 3.

DESIGN

3.1 Data Flow Diagram

A data flow diagram (DFD) is a graphical representation of the "flow" of data through an information system, modelling its process aspects. A DFD is often used as a preliminary step to create an overview of the system without going into great detail, which can later be elaborated. DFDs can also be used for the visualization of data processing (structured design).

A DFD shows what kind of information will be input to and output from the system, how the data will advance through the system, and where the data will be stored. It does not show information about process timing or whether processes will operate in sequence or in parallel, unlike a traditional structured flowchart which focuses on control flow, or a UML activity workflow diagram, which presents both control and data flows as a unified model.

3.1.1 LEVEL 0 DFD

Different assemblers give distinct object codes, but the linkage loader needs to make sure that this is combined into a single symbol table that can be loaded into the memory.

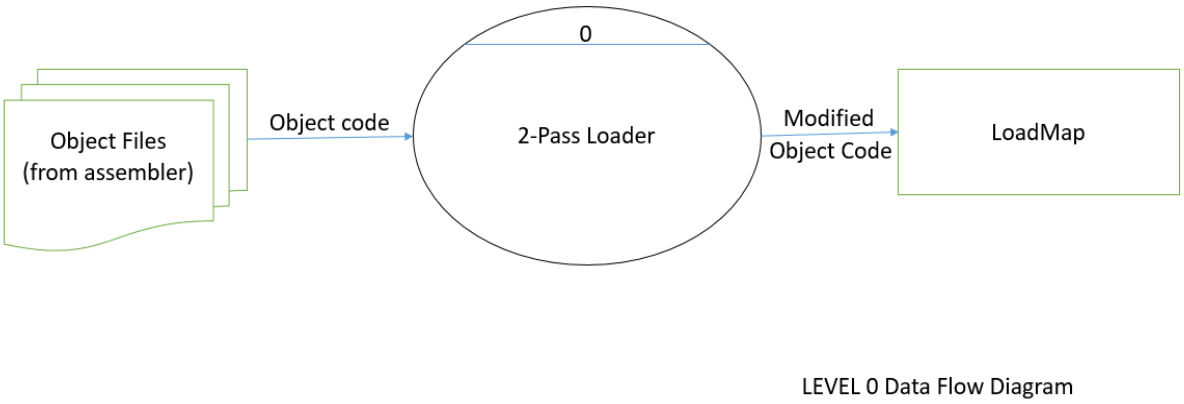

Figure 3.1: DFD Level 0

Level 0 provides a view of the system as a whole. The input object code is converted to an output in form of a load map.

3.1.2 LEVEL 1 DFD

Here, the flow of information is more specifically mentioned, showing each of the components involved with each process.

In first pass , an external symbol table (ESTAB) is populated, which is also accessible later by the 2nd pass. The first pass also throws errors when there are unresolved dependencies, which cannot be represented in the data flow diagram as it is an abnormal state of the system. Therefore first pass does 2 major functions: interprets the input and stores the records in intermediate storage; populates the external symbol table. The first pass is responsible for the program linking.

The second pass uses the external symbol table to apply the modification to the text records, which are then used to create the output load map. The second pass throws errors on corrupted modification

records and unmet external references. The second pass is also responsible for the actual program relocation.

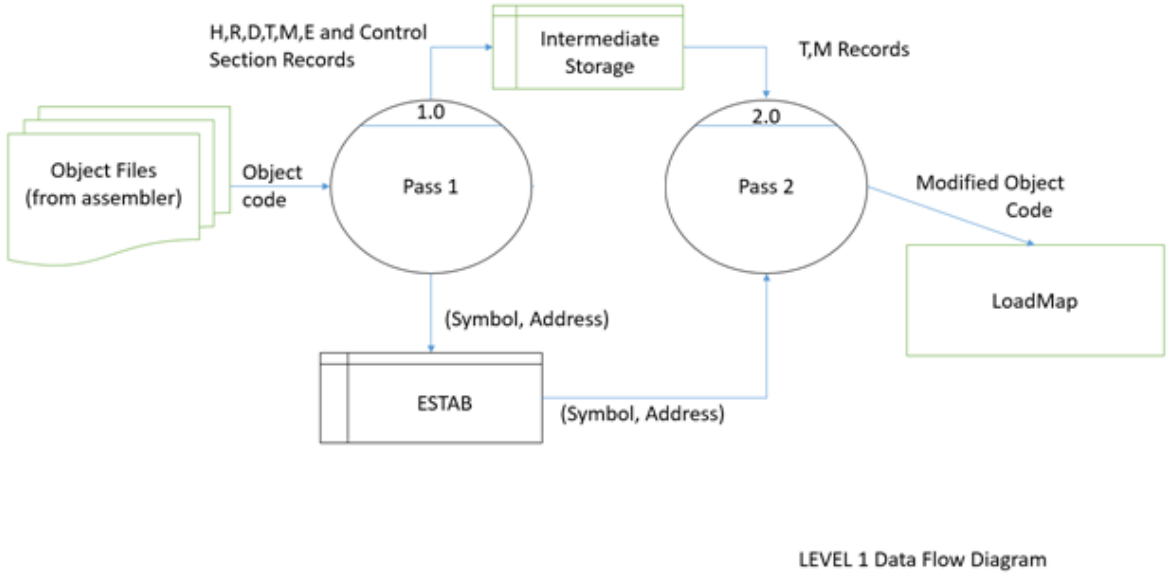

Figure 3.2: DFD Level 1

3.1.3 LEVEL 2 DFD

At the second level, individual components are looked at in detail. The two components represented by process 1.0 and 2.0 can be further split into sub components.

Here, the process *Pass 1 (1.0)* is looked at in detail. The first pass performs the function of parsing the input form the object files by first tokenizing the input and storing them in the record lists and then checking for syntax and handling multiple control sections. The records are stored in the intermediate storage, and then the define records of every control section is processed to populate the external symbol table (linking).

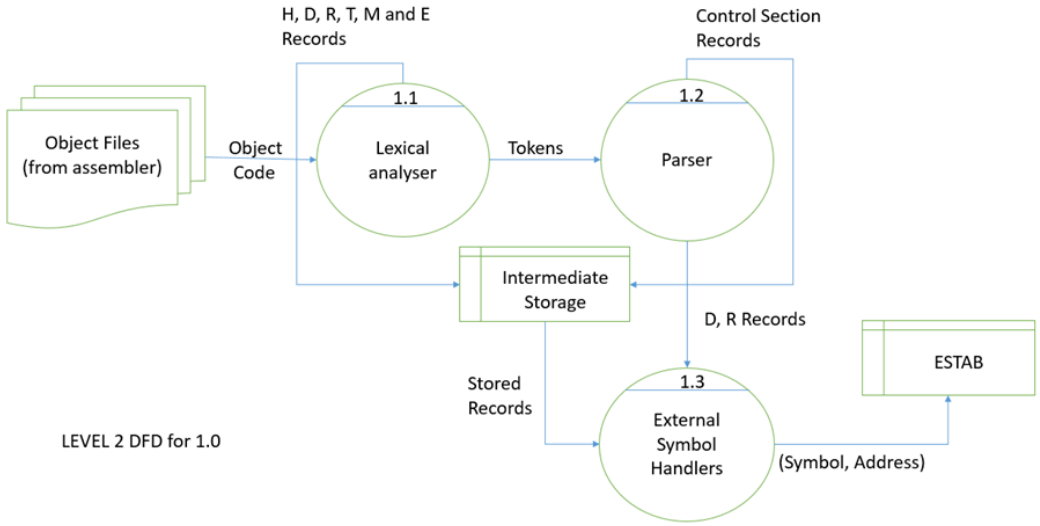

Figure 3.3: DFD level 2 for 1.0

Here, the actual relocation and the generation of the load map is done. The refer records are first processed to create a local symbol table, and then the text records are modified to reflect the program address provided by the OS. Finally, the modify records are processed to modify the text records and a load map is generated with the modified text records.

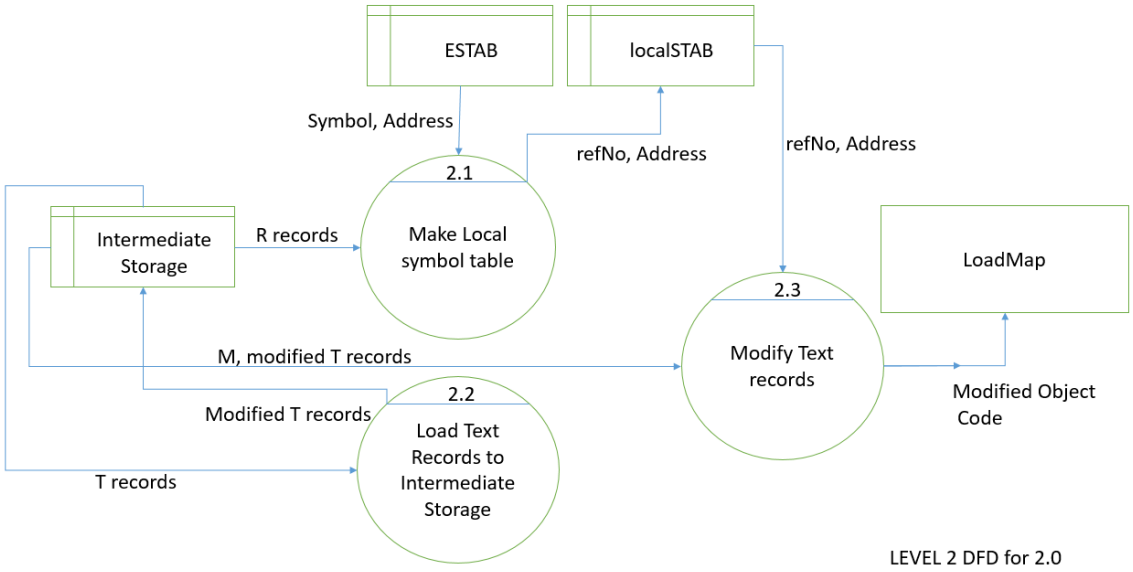

Figure 3.4: DFD Level 2, for 2.0

3.2 Data Structures

External symbol tables (ESTAB)

(i). Like SYMTAB, store the name and address of each external symbol in the set of control sections being loaded.

(ii) It needs to indicate in which control section the symbol is defined

ESTAB is used to store the name and address of each external symbol in the set of control sections being loaded. It has two variables PROGADDR and CSADDR.

PROGADDR is the beginning address in memory where the linked program is to be loaded, and CSADDR contains the starting address assigned to the control section currently being scanned by the loader.

A reference number, is used in Modification records, like the number 01 to the control section name. The main advantage of this reference-number mechanism is that it avoids multiple searches of ESTAB for the same symbol during the loading of a control section.

CHAPTER 4

CONCLUSION

Two Pass loader being one of the most used software, has been undergoing several changes and enhancements over the years to incorporate more features and make it user friendly. This project aims to add more features to the existing ones of the text editor, to make it more robust and exhaustive.

4.1 Limitations of the project

Some of the limitations found in the project are:

- It is currently functioning only on operating system with C installed because other language like Python, Java do not support the use of dictionaries for easy implementation.
- It assumes that an assembler generated object code used as input is syntactically and semantically correct. Only basic syntax checking has been implemented in this project.

4.2 Future Enhancements

- Integrating performance enhancing features like using multicore programming constructs to construct multi pass loader.
- Moving towards platform independence and program portability by removing platform specific components.

CHAPTER 5

REFERENCES

[1] David Salomon (1993). Assemblers and Loaders.

[2] Beck, Leland L. (1996). "2". System Software: An Introduction to Systems Programming. Addison Wesley.

[3] Hyde, Randall. "Chapter 12 – Classes and Objects". The Art of Assembly Language and loader, 2nd Edition. No Starch Press. © 2010.

APPENDIX A

PROGRAM SOURCE CODE

A.1 Pass-1 Source Code (pass1.c)

```
#include<stdio.h>
#include<string.h>
#include<stdlib.h>
struct estab
{
  char csname[10];
  char extsym[10];
  int address;
  int length;
}es[20];
char ref[20][20];
char def[20][20];
int refn=0,defn=0;
void main()
{
        char input[10],name[10],symbol[10],ch;  int count=0,progaddr,csaddr,add,len;
        int flag=0;
    int i,j;
        FILE *fp1,*fp2;
        fp1=fopen("INPUT.DAT","r");
        fp2=fopen("ESTAB.DAT","w");
        printf("\n\nEnter the address where the program has to be loaded : ");
        scanf("%x",&progaddr);
        if(progaddr<0 && progaddr>20000){
      printf("Invalid address\n");
                exit(0);
        }
        csaddr=progaddr;
        fscanf(fp1,"%s",input);
        while(strcmp(input,"END")!=0)
        {
                if(strcmp(input,"H")==0)
```

```
		{
			fscanf(fp1,"%s",name);
			strcpy(es[count].csname,name);
			strcpy(es[count].extsym,"  ");
			fscanf(fp1,"%x",&add);
			es[count].address=add+csaddr;
			fscanf(fp1,"%x",&len);
			es[count].length=len;
			fprintf(fp2,"%s ** %x
%x\n",es[count].csname,es[count].address,es[count].length);
			count++;
		}
		else if(strcmp(input,"D")==0)
		{
			fscanf(fp1,"%s",input);
			while(strcmp(input,"R")!=0)
			{
				strcpy(es[count].csname,"  ");
				if(input[0]>=65 && input[0]<=90){
				strcpy(es[count].extsym,input);
	strcpy(def[defn++],es[count].extsym);
				fscanf(fp1,"%x",&add);
				es[count].address=add+csaddr;
				es[count].length=0;
				fprintf(fp2,"** %s %x\n",es[count].extsym,es[count].address);
				count++;
				fscanf(fp1,"%s",input);
				}

				else{
					printf("\nERROR: Incorrect Format. Expecting the Label
name.\n");
					exit(1);
			}
			}
			csaddr=csaddr+len;
			fscanf(fp1,"%s",input);
```

```
                while(strcmp(input,"T")!=0)
                {
                        strcpy(ref[refn++],input);
                        fscanf(fp1,"%s",input);
                }
        }
        else if(strcmp(input,"T")==0)
        {
                while(strcmp(input,"E")!=0)
                fscanf(fp1,"%s",input);
        }
        fscanf(fp1,"%s",input);
    }



    for(i=0;i<refn;i++){
        flag=0;
        for(j=0;j<defn && flag==0;j++){
                if(strcmp(ref[i],def[j])==0){
                        flag=1;
                }
        }
        if(flag==0){
                printf("\nERROR: Referred external symbol \"%s\" not defined\n",ref[i]);
                exit(1);
                }
}
fclose(fp1);
fclose(fp2);

fp2=fopen("ESTAB.DAT","r");
ch=fgetc(fp2);
while(ch!=EOF)
{
 printf("%c",ch);
 ch=fgetc(fp2);
```

```
  }
  fclose(fp2);
}
```

A.2 Pass-2 Source Code (pass2.c)

```
#include<stdio.h>
#include<string.h>
#include<stdlib.h>
struct exttable
{
  char cextsym[20], extsym[20];
  int address,length;
}estab[20];
struct objectcode
{
 unsigned char code[15];
 int add;
}obcode[500];
void main()
{
 char temp[10];
 FILE *fp1,*fp2,*fp3;
 int i,j,x,y,pstart,exeloc,start,textloc,loc,textlen,length,location,st,s;
 int n=0,num=0,inc=0,count=0,record=0,mloc[30],mlen[30];
 signed long int newadd;
 char operation,lbl[10],input[10],label[50][10],opr[30],ch,*add1,address[10];
 fp1=fopen("INPUT.DAT","r");
 fp2=fopen("ESTAB.DAT","r");
 fp3=fopen("OUTPUT.DAT","w");
 while(!feof(fp2))
 {
        fscanf(fp2,"%s %s %x %x", estab[num].cextsym, estab[num].extsym, &estab[num].address,
                                &estab[num].length);
        num++;
 }
 exeloc=estab[0].address;
```

```
loc=exeloc;
start=loc;
st=start;
while(!feof(fp1))
{
	fscanf(fp1,"%s",input);
	if(strcmp(input,"H")==0)
	{
		fscanf(fp1,"%s",input);
		for(i=0;i<num;i++)
		if(strcmp(input,estab[i].cextsym)==0)
		{
			pstart=estab[i].address;
			break;
		}
		while(strcmp(input,"T")!=0)
			fscanf(fp1,"%s",input);
	}
	do
	{

		if(strcmp(input,"T")==0)
		{
			fscanf(fp1,"%x",&textloc);
			textloc=textloc+pstart;
			for(i=0;i<(textloc-loc);i++)
			{
				strcpy(obcode[inc].code,"..");
				obcode[inc++].add=start++;
			}
			fscanf(fp1,"%x",&textlen);
			loc=textloc+textlen;
		}
		else if(strcmp(input,"M")==0)
		{
			fscanf(fp1,"%x",&mloc[record]);
			mloc[record]=mloc[record]+pstart;
```

```
                fscanf(fp1,"%x",&mlen[record]);
                fscanf(fp1,"%s",label[record++]);
        }
        else
        {
                length=strlen(input);
                x=0;
                for(i=0;i<length;i++)
                {
                        obcode[inc].code[x++]=input[i];
                        if(x>1)
                        {
                                obcode[inc++].add=start++;
                                x=0;
                        }
                }
        }
        fscanf(fp1,"%s",input);
    }while(strcmp(input,"E")!=0);
    if(strcmp(input,"E")==0)
    fscanf(fp1,"%s",input);
}
    for(n=0;n<record;n++)
    {
    operation=label[n][0];
    length=strlen(label[n]);
    for(i=1;i<length;i++)
    {
        lbl[i-1]=label[n][i];
    }
    lbl[length-1]='\0';
    length=0;
    strcpy(address,"\0");
    location=mloc[n]-exeloc;
    loc=location;
    count=0;
    while(length<mlen[n])
```

```
        {
                strcat(address,obcode[location++].code);
                count++;
                length+=2;
        }
        for(i=0;i<num;i++)
        {
                if(strcmp(lbl,estab[i].cextsym)==0)
                break;
                if(strcmp(lbl,estab[i].extsym)==0)
                break;
        }
        switch(operation)
        {
                case '+':
                                newadd=strtol(address,&add1,16)+(long int)estab[i].address;
                                break;
                case '-':
                                newadd=strtol(address,&add1,16)-(long int)estab[i].address;
                                break;
        }
 ltoa(newadd,address,16);
 x=0; y=0;
 while(count>0)
 {
  obcode[loc].code[x++]=address[y++];
  if(x>1)
  {
                x=0; loc++;
                count--;
  }
 }
 }
count=0;
n=0;
s=st-16;
fprintf(fp3,"%x\t",s);
```

```
for(i=1;i<=16;i++)
{
 fprintf(fp3,"xx");
 if(i==4||i==8||i==12)
 {
  fprintf(fp3,"\t");
 }
}
fprintf(fp3,"\n\n%x\t",obcode[0].add);
for(i=0;i<inc;i++)
{
        fprintf(fp3,"%s",obcode[i].code);
        n++;
        if(n>3)
        {
                fprintf(fp3,"\t");
                n=0;
                count++;
        }
        if(count>3)
        {
                fprintf(fp3,"\n\n%x\t",obcode[i+1].add);
                count=0;
        }
 }
fclose(fp1);
fclose(fp2);
fclose(fp3);
 printf("\n\t***** PASS TWO OF A DIRECT-LINKING LOADER *****\n");
 printf("\nThe contents of the output file :");
 printf("\n----------------------------------------------------------");
 printf("\nAddress\t\t\t\tContents");
 printf("\n----------------------------------------------------------\n");
 fp3=fopen("OUTPUT.DAT","r");
 ch=fgetc(fp3);
 while(ch!=EOF)
 {
```

```
  printf("%c",ch);
  ch=fgetc(fp3);
 }
 fclose(fp3);

}
```

APPENDIX B

SNAPSHOTS

```
1  H PROGA 000000 000063
2  D LISTA 000054
3  R LISTB ENDB LISTC ENDC
4  T 000020 0A 03201D 77100004 050014
5  T 000054 0F 100014 000008 004051 000004 100000
6  M 000024 05 +LISTB
7  M 000054 06 +LISTC
8  M 000060 06 +LISTB
9  M 000060 06 -LISTA
10 E 000020
11
12 H PROGB 000000 00007F
13 D LISTB 000060 ENDB 000070
14 R LISTA LISTC ENDC
15 T 000036 0B 03100000 772027 05100000
16 T 000070 0F 100000 000008 004051 000004 100060
17 M 000037 05 +LISTA
18 M 00003E 05 -LISTA
19 M 000070 06 -LISTA
20 M 000070 06 +LISTC
21 M 00007C 06 +PROGB
22 M 00007C 06 -LISTA
23 E 000000
24
25 H PROGC 000000 0000051
26 D LISTC 000030 ENDC 000042
27 R LISTA LISTB ENDB
28 T 000018 0C 03100000 77100004 05100000
29 T 000042 0F 100030 000008 004051 000004 100000
30 M 00001D 05 +LISTB
31 M 000021 05 -LISTA
32 M 000042 06 -LISTA
33 M 000042 06 +PROGC
34 M 00004E 06 +LISTB
35 M 00004E 06 -LISTA
36 E
37 END
```

Figure B.1 Input consisting of three object programs

```
C:\Users\Pavankumar\Desktop\sd>gcc try1.c -o pass1

C:\Users\Pavankumar\Desktop\sd>pass1


Enter the address where the program has to be loaded : 2000
PROGA ** 2000 63
** LISTA 2054
PROGB ** 2063 7f
** LISTB 20c3
** ENDB 20d3
PROGC ** 20e2 51
** LISTC 2112
** ENDC 2124

C:\Users\Pavankumar\Desktop\sd>
```

Figure B.2 Output given by pass-1

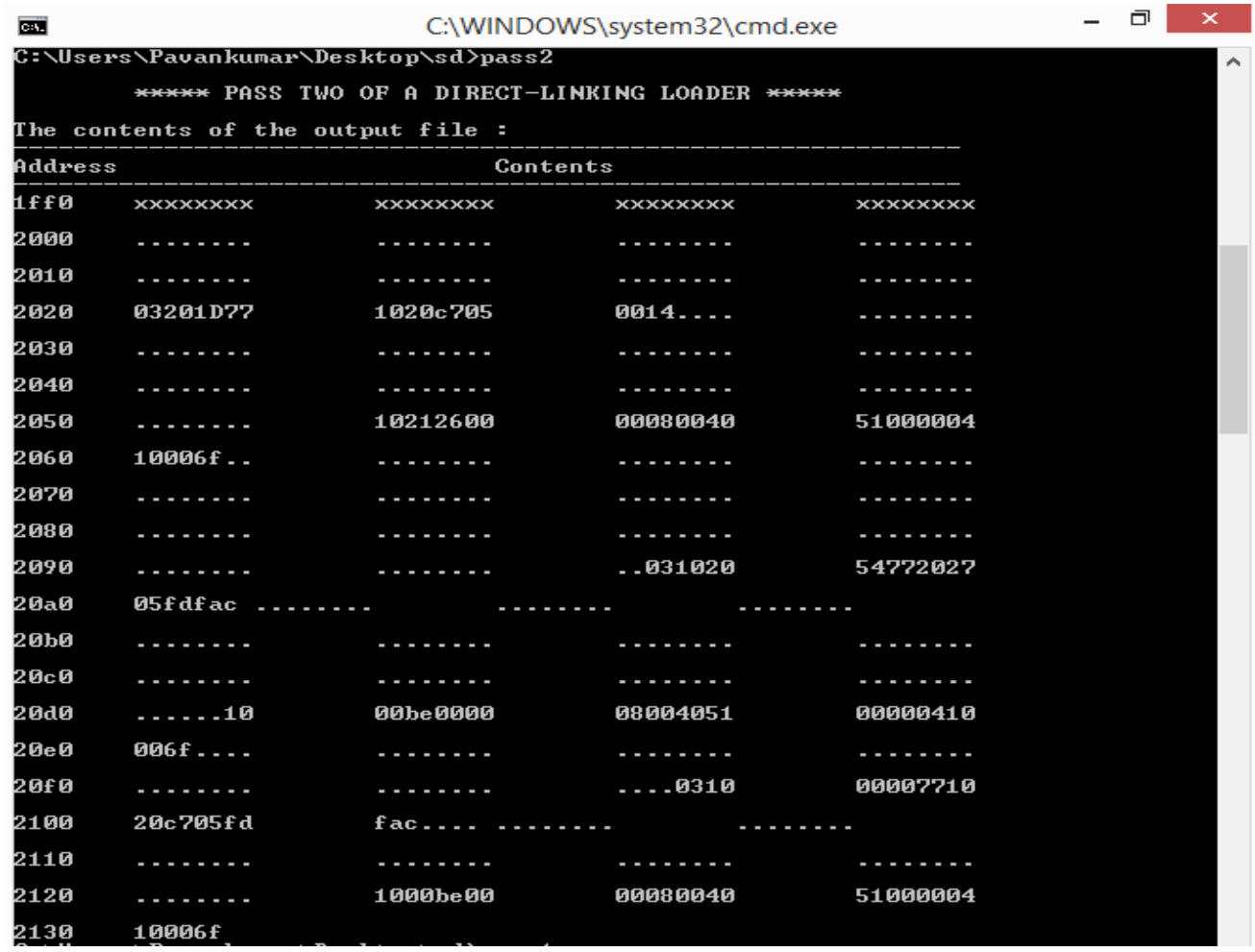

Figure B.3 Output given by pass-2

```
1   H PROGA 000000 000063
2   D LISTA 000054
3   R LISTZ ENDB LISTC ENDC
4   T 000020 0A 03201D 77100004 050014
5   T 000054 0F 100014 000008 004051 000004 100000
6   M 000024 05 +LISTB
7   M 000054 06 +LISTC
8   M 000060 06 +LISTB
9   M 000060 06 -LISTA
10  E 000020
11
12  H PROGB 000000 00007F
13  D LISTB 000060 ENDB 000070
14  R LISTA LISTC ENDY
15  T 000036 0B 03100000 772027 05100000
16  T 000070 0F 100000 000008 004051 000004 100060
17  M 000037 05 +LISTA
18  M 00003E 05 -LISTA
19  M 000070 06 -LISTA
20  M 000070 06 +LISTC
21  M 00007C 06 +PROGB
22  M 00007C 06 -LISTA
23  E 000000
24
25  H PROGC 000000 0000051
26  D LISTC 000030 ENDC 000042
27  R LISTA LISTB ENDB
28  T 000018 0C 03100000 77100004 05100000
29  T 000042 0F 100030 000008 004051 000004 100000
30  M 00001D 05 +LISTB
31  M 000021 05 -LISTA
32  M 000042 06 -LISTA
33  M 000042 06 +PROGC
34  M 00004E 06 +LISTB
35  M 00004E 06 -LISTA
36  E
37  END
```

Figure B.4 Input containing undefined label

```
C:\Users\Pavankumar\Desktop\sd>pass1

Enter the address where the program has to be loaded : 4000
ERROR: Referred external symbol "LISTZ" not defined
C:\Users\Pavankumar\Desktop\sd>
```

Figure B.5 Error message generated by the pass-1