BHARATI VIDYAPEETH DEEMED UNIVERSITY

COLLEGE OF ENGINEERING, PUNE - 43

DEPARTMENT OF COMPUTER ENGINEERING

Laboratory Manual

Systems Programming

Vision of the Institute
“To be World Class Institute for Social Transformation through Dynamic Education”

Mission of the Institute
A. To provide quality technical education with advanced equipment, qualified faculty members, infrastructure to meet needs of profession and society.
B. To provide an environment conducive to innovation, creativity, research, and entrepreneurial leadership.
C. To practice and promote professional ethics, transparency and accountability for social community, economic and environmental conditions. VISION OF THE DEPARTMENT To pursue and excel in the endeavour for creating globally recognised Computer Engineers

VISION OF THE DEPARTMENT

“To pursue and excel in the endeavour for creating globally recognized computer engineers through quality education.”

Mission of the Department

- To impart engineering knowledge and skills confirming to a dynamic curriculum.
- To develop professional, entrepreneurial & research competencies encompassing continuous intellectual growth.
- To produce qualified graduates exhibiting societal and ethical responsibilities in working environment

PROGRAM EDUCATIONAL OBJECTIVES(PEOs):
1. Demonstrate technical and professional competencies by applying engineering fundamentals, computing principles and technologies.
2. Learn, Practice, and grow as skilled professionals/ entrepreneur/researchers adapting to the evolving computing landscape.
3. Demonstrate professional attitude, ethics, understanding of social context and interpersonal skills leading to a successful career.

ROGRAM SPECIFIC OUTCOMES(PSO)s:
PSO 1: To design, develop and implement computer programs on hardware towards solving problems.
PSO 2: To employ expertise and ethical practise through continuing intellectual growth and adapting to the working environment.

PROGRAMME OUTCOMES(POs):
Upon completion of the course the graduate engineers will be able to:
1. Apply the knowledge of mathematics, science, engineering fundamentals, and computing for the solution of complex engineering problems.
2. Identify, formulate, research literature, and analyse complex engineering problems reaching substantiated conclusions using computer engineering foundations, principles, and technologies.
3. Design solutions for complex engineering problems and design system components or processes that meet the specified needs with appropriate consideration for public health and safety, and cultural, societal, and environmental considerations.
4. Use research-based knowledge and research methods including design of experiments, analysis and interpretation of data, and synthesis of the information to provide valid conclusions.
5. Create, select, and apply appropriate techniques, resources, and modern engineering and IT tools including prediction and modelling to complex engineering activities with an understanding of the limitations.
6. Apply reasoning informed by the contextual knowledge to assess societal, health, safety, legal, and cultural issues, and the consequent responsibilities relevant to the professional engineering practice.
7. Understand the impact of the professional engineering solutions in societal and environmental contexts, and demonstrate the knowledge of, and the need for sustainable development.
8. Apply ethical principles while committed to professional responsibilities and norms of the engineering practice.
9. Function effectively as an individual, and as a member or leader in diverse teams, and in multidisciplinary settings
10.Communicate effectively on complex engineering activities with the engineering community and with the society at large, such as being able to comprehend and write effective reports and design documentation, make effective presentations, and give and receive clear instructions.
11.Apply the engineering and management principles to one's work, as a member and leader in a team.
12.Recognise the need for and have the preparation and ability to engage in independent and life-long learning in the broadest context of technological change.

General Instructions (Laboratory rules/ Practical oriented rules)

General Instructions

- Equipment in the lab is meant for the use of students. Students need to maintain a proper decorum in the computer lab. Students must use the equipment with care.
- Students are required to carry their reference materials, files and records with completed assignment while entering the lab.
- Students are supposed to occupy the systems allotted to them and are not supposed to talk or make noise in the lab.
- All the students should perform the given assignment individually.
- Lab can be used in free time/lunch hours by the students who need to use the systems should take prior permission from the lab in-charge.
- All the Students are instructed to carry their identity cards when entering the lab.
- Lab files need to be submitted on or before date of submission.
- Students are not supposed to use pen drives, compact drives or any other storage devices in the lab.
- For Laboratory related updates and assignments students should refer to the notice board in the Lab

Class: B.Tech Computer Sem IV
Course Name: SystemsProgramming
Course Code:
Contact hours: 2 hrs/Week

Weekly Plan

Week	**Topic to be Covered**
Week 1	Introduction to Language translators
Week 2	Assemblers Data structures, Design
Week 3	Algorithm for singlepass and two pass assembler
Week 4	Macroprocessor Macro expansion Data structures
Week 5	Functions of loaders and Linkers,Evolution of loaders and Linkers
Week 6	Use of Dynamic link libraries
Week 7	Ciompiler concept,Phases of compilet
Week 8	Psrdsing
Week 9	Operating System ,Structure of Opeating system,Types of Shell
Week 10	Shell Script, Shell Programming
Week 11	Device drivers,Types,Installation for line printer

Examination Scheme

TW: 25marks
Practical Examination: 25 marks

Marking Scheme

Term work marks (25) for each assignment is evaluated in three parts as follows

Sr. No	Evaluation Criteria	Marks for each Criteria	Rubrics
1	**Timely Submission**	**07**	➢ Punctuality reflects the work ethics. Students should reflect that work ethics by completing the lab assignments and reports in a timely manner without being reminded or warned.
2	**Presentation**	**06**	➢ Student are expected to write the technical document (lab report) in their own words. The presentation of the contents in the lab report should be complete, unambiguous, clear, and understandable. The report should document approach/algorithm/design and code with proper explanation.

3	**Understanding**	**12**	➢ Correctness and Robustness of the code is expected. The Learners should have an in-depth knowledge of the practical assignment performed.The learner should be able to explain methodology used for designing and developing the program/solution. He/she should clearly understand the purpose of the assignment and its outcome.

Oral Examination Marks (25) evaluated as follows

- Overall Knowledge : 05 Marks
- Argument/Analytical abilities:05 Marks
- Reasoning : 15 Marks
- Total Marks:50

<u>Laboratory Usage</u>–

Name of the laboratory: System software
Number of machines: 20

Details of machines:

Configuration	**Details**	**Quantity**
Hardware	Processor: Intel(R), Core i5-3470 CPU@ 3.20GHz Installed memory(RAM): 4GB HDD: 500GB	20
Software	Operating System: Windows 8.1	
	Software: Turbo C, TASM, JDK, rational rose	

<u>Practical Pre-requisite</u>

1. Knowledge of Assembly language
2. C-programming skills
3. Knowledge of Data structures

Course Objectives:

1. To help the students understand functioning of various system programs
2. To initiate an understanding of design of language translators and brief about phases of compiler
3 To create awareness to use dynamic link libraries.
4. To introduce the students the basics of shell programming

Course Outcomes:

1 Interpret theoretical and practical aspects of language translation.

2. Examine working and design of assemblers and microprocessors

3. Relate the concept of memory allocation, relocation and the functions of loaders, linkers

4. Examine the phases of compiler and its working.

5. Interpret various operating system concepts

6. Demonstrate use of device drivers

Outcome	Assignment Number	Level	Proficiency evaluated by
Interpret theoretical and practical aspects of language translation	1,2,3,5	Synthesis	Report
Examine working and design of assemblers and microprocessors	1,2,3,5	Synthesis	Experiment aim
Relate the concept of memory allocation, relocation and the functions of loaders, linkers	4	Analysis	Setting up experiments
Examine the phases of compiler and it's working	5	Synthesis	Conducting experiments and reporting results
Interpret various operating system concepts	6,7,8	Synthesis	Conducting experiments and reporting results
Demonstrate use of device drivers	9,10	Analysis	Setting up experiments

LIST OF PRACTICALAssignments:

1. Develop a single pass Assembler
2. Develop a two pass Assembler
3. Develop a two e Microprocessor
4. Demonstrate use of Dynamic Link Libraries.

5. Develop a lexical analyzer using Lex Compiler.+

6. Study of Shell programming basics

7. Demonstrate the use of VI editor commands.

8. Use Linux and compute average of N numbers.

9. Study of device drivers.

10. Demonstrate use of device drivers.

Assignment 1

Aim:
Develop a single pass assembler.

Procedure:

Step no	Details of the step
1	Begin
2	Read first input line
3	if OPCODE='START' then a. save #[operand] as starting address b. initialize LOCCTr as starting address c. read next input line end
4	else initialize LOCCTR to 0
	while OPCODE != 'END' do

5	d. if there is not a comment line then e. if there is a symbol in the LABEL field then i. search SYMTAB for LABEL ii. if found then 1. if symbol value as null 2. set symbol value as LOCCTR and search the linked list with the corresponding operand 3. PTR addresses and generate operand addresses as corresponding symbol values 4. set symbol value as LOCCTR in symbol table and delete the linked list iii. end iv. else insert (LABEL,LOCCTR) into SYMTAB v. end
6	search OPTAB for OPCODE
7	if found then search SYMTAB for OPERAND address
8	if found then

	f. if symbol value not equal to null then i) store symbol value as operand address else insert at the end of the linked list with a node with address as LOCCTR
9	else insert (symbol name, null) add 3 to LOCCTR.
10	elseif OPCODE='WORD' then add 3 to LOCCTR & convert comment to object code
11	elseif OPCODE = 'RESW' then add 3 #[OPERND] to LOCCTR
12	elseif OPCODE = 'RESB' then add #[OPERND] to LOCCTR
13	elseif OPCODE = 'BYTE' then g. find length of the constant in bytes h. add length to LOCCTR convert constant to object code
14	if object code will not fit into current text record then i. write text record to object program j. initialize new text record o. add object code to text record
15	write listing line
16	read next input line
17	write last text record to object program
18	write end record to object program
19	write last listing line
20	End

INPUT FILES:

input1.txt

```
**          START       6000
**          JSUB        CLOOP
**          JSUBRLOOP
ALPHA       WORD        23
BETA        RESW        3
GAMMA       BYTE        C'Z'
DELTA       RESB        4
CLOOP       LDA         ALPHA
RLOOP       STA         BETA
**          LDCH        GAMMA
```

```
**          STCH        DELTA
**          END**
```

optab1.txt

```
START        *
JSUB         48
LDA          14
STA          03
LDCH         53
STCH         57
END          *
```

OUTPUT FILES:

```
CLOOP    6023
RLOOP    6026
ALPHA     6006
BETA     6009
GAMMA    6018
DELTA  6019
```

spout.txt

```
**          START       6000        0
**          JSUB        CLOOP       486023
**          JSUB        RLOOP       486026
ALPHA       WORD              23          23
BETA        RESW        3           0
GAMMA       BYTE        C'Z'        90
DELTA       RESB        4           0
CLOOP       LDA         ALPHA       146006
RLOOP       STA         BETA        036009
**          LDCH        GAMMA       536018
**          STCH        DELTA       576019
**          END         **          0
```

e) Result:

Thus the program for single pass assembler is implemented and the output is verified accordingly.

Questions:

1. Define Assembler and Disassembler
2. List the problems in developing assembler in single pass
3. List and explain types of assembly language statements

Assignment 2

Aim: Develop a two pass assembler

OBJECTIVE:

To understand the basic elements of assembly language.

To learn the different types ofassembly language statements.

To study and understand the phases structure of an assembler.

To understand what are forward references.

To study the difference between two-pass and single pass assembly.

To know about the specification of all the databases with their format

To understand the different types of intermediate code representations.

To understand the flowchart for first pass of assembler

THEORY:

Assembler: Software that translates assembly language into machine language.

Basic elements of Assembly language:

- Mnemonic Operation Codes: Eliminates the need to memorize numeric operation codes.Enables the assembler to provide helpful diagnostic message
  - Symbolic Operands- Can be associated with data or instructions. Can be used as operands in assembly statements. Assembler performs memory bindings to these names.
  - Data Declarations-Can be declared in variety of notations.

Assembly language statements:

- Imperative Statements-Indicate and action to be performed during the execution of assembled program. Each statement translates into one machine instruction.
- Declarative Statements:

  Syntax

  ```
  [Label]        DS     [Constant]
  [Label]        DC     '<Value>'
  ```

  DS(Declare Storage) statement reserves areas of memory and associates names with them.

  Eg. A DS 5 reserves memory area of 5 words and associates the name A with it.

DC (Declare Constant) statement constructs memory words containing constants. It merely initializes memory words to given values, these values are not protected and can be changed by moving a new value to the memory word.

1. Design of Pass 1 of a two pass assembler

Eg. ONE   DC   ‘1’ associates the name ONE with a memory word containing value ‘1’A literal is an operand with the syntax='<value>.It differs from aconstant because its location cannot be specified in the assemblyprogram. This helps to ensure that its value is not changed duringexecution of a program

Eg.ADD AREG ,   ='5'

- Assembler Directives: Instructs the assembler to perform certain actions during the assembly of a program.
  - Start- START <constant>

    This directive indicates that the first word of the target programgenerated by the assembler should be placed in the memory word withaddress< constant>.
  - END – END [<operand spec>]

    This directive indicates the end of the source program. The optional <operand spec> indicates the address of the instruction where the execution of the program should begin. (By default, execution begins with the first instruction of the assembled program
  - ORIGIN-The syntax of this directive is

    ORIGIN <address spec>

    where<address spec> is an<operand spec> or<constant>.This directive indicates that LC should be set to the address given by <address spec>. The ORIGIN statement is useful when the targetprogram does not consist of consecutive memory words.

    Eg. Origin Loop+2

    Assuming Loop has the address 200, now using the Origin directive, LC is set to 204.

    EQU-The EQU statement has the syntax

    <symbol>EQU<address spec>

    where<address spec> is an<operand spec> or< constant>.The EQU statement defines the symbol to represent<addressspec>. This differs from the DC/DS statement as no LCprocessing is implied.

    BACK   EQU   LOOP

    introduces the symbol BACK to represent the operand LOOP
  - LTORG- LTORG statement permits a programmer to specifywhere literals should be placed. By default, assembler places the literalsafter the END statement.Atevery LTORG statement, as also at the ENDstatement, theassembler

allocatesmemory to the literals of a literal pool. The poolcontains all literals used in the program since the start of the programor since the last LTORG statement.

Phase Structure of Assembler

Synthesis Phase

Analysis Phase

Forward Reference- Reference to an entity which precedes its definition in the program is called forward reference.

Eg.

```
    Mover AREG, ONE
        :
    ONE DS 5
```

Pass Structure of an Assembler

Pass: A pass of a language processor means a complete scan of the source program

- Two pass translation: Can handleforward references easily. LC processing is performed in the first passand symbols defined in the program are entered into the symbol table.The second pass synthesizes the target form using the addressinformation found in the symbol table. In effect, the first pass performsanalysis of the source program while the second pass performssynthesis of the target program. The first pass constructs an intermediate representation (IR) of the source program for use by the second pass. This representation consists of two main components—data structures, e.g. the symbol table, and a processedform of the source program. The latter component is called intermediate code(IC).
- Single pass translation: LC processing and construction of the symbol table proceed as intwo pass translation. The problem of forward references is tackledusing a process called back patching. The operand field of aninstruction containing a forward reference is left blank initially.The address of the forward referenced symbol is put into this fieldwhen its definition is encountered. The need for inserting the operand's address at a later stage can be indicated by adding an entry to the Table of Incomplete Instructions (TOII). This entry is a pair (instruction address>, <symbol>.By the time the END statement is processed, the symbol tablewould contain the addresses of all symbols defined in the sourceprogram and TOII would contain information describing all forwardreferences. The assembler can now process each entry in TOII tocomplete the concerned instruction.

Databases and their formats

MNEMONIC TABLE

Mnemonic	Opcode	Length	Class

SYMBOL TABLE

Sr. No	Symbol	Length	Value	Address

LITERAL TABLE

Sr. No	Literal	Address

POOL TABLE

Sr. No	Pool Start Index

Intermediate Code Representations:

The intermediate code consists of a set of IC units, each IC unitconsisting of the following three fields:

1.Address

2.Representation of the mnemonic opcode

3.Representation of operands
Mnemonic Field contains a pair of the form (statement class, code) where statement class can one of the IS, DL and AD representing Imperative Statements, Declarative Statements and Assembler directives respectively.

1 Design of Pass 1 of a two pass assembler
Code is the instruction opcode in machine language.

- Variant I- First Operand is represented by single digit number for the register and the second operand is represented by a pair of the form (operand class, code) where operand class is one of C,S and L standing for constant, symbol and literal resp. For constant, the code field contains the internal representation of the constant and for the symbol, literal, it contains the ordinal number of the operand's entry in the symbol or literal table. Entries are made in the symbol table as soon as a symbol is encountered. In case of forward references, the size and length field cannot be filled, as a result two kinds of entries exist in the symbol table, for defined symbols and for forward references,
- Variant II- Here, for declarative and assembler directives, processing of the operand field takes place. For imperative statements, the operand field is processed only if it contains literals.

ALGORITHM

```
1.   Loc_cntr=0(default value)
     Pooltab_ptr=1; POOLTAB [1]=1;
     Littab_ptr=1;
2.   While next statement is not an END statement
a.   If label is present then
     This_label=symbol in label field;
     Enter(this_label,loc_cntr) in SYMTAB.
b.   If an LTORG statement then
i.   Process literals LITTAB[POOLTAB[pooltab_ptr]…LITTAB[littab_ptr-1] to allocate
     memory and put the addrsss in the address field. Update loc_cntr accordingly.
ii.  Pooltab_ptr=pooltab_ptr+1;
iii. POOLTAB[pooltab_ptr]=littab_ptr;
c.   If a START or ORIGIN statement then
     Loc_cntr=value specified in operand field
d.   If an EQU statement then
i.   This_adddr=value of <address spec>
ii.  Correct the symtab entry for this_label to (this_label, this_addr)
e.   If a declaration statement then
i.   Code=code of the declaration statement;
ii.  Size=size of memory area required by DS/DC
iii. Loc_cntr=loc_cntr+size;
iv.  Generate IC(DL,code)…
f.   If an imperative statement then
i.   Code=machine Opcode from OPTAB;
ii.  Loc_cntr=loc_cntr+instruction length from OPTAB
iii. If operand is a literal then
     This_literak=literal in operand field
     LITTAB[littab_ptr]=this_literal
     Littab_ptr=littab_ptr+1
```

3.(Processing of END statement)

a. Perform step 2(b)

b. Generate IC

c. Go to Pass II

INPUT:

MNEMONIC TABLE

Mnemonic	Opcode	Length	Class
START	01	0	AD
END	02	0	AD
EQU	03	0	AD
LTORG	04	0	AD
ORIGIN	05	0	AD
DS	01	0	DL
DC	02	0	DL
MOVER	01	02	IS
ADD	02	01	IS
SUB	03	02	IS

SOURCE CODE

Label	Mnemonic	Operand 1	Operand 2
	START	100	
Loop	MOVER	AREG	TEMP
	ADD	BREG	AREG
	MOVER	AREG	COUNT
	ADD	AREG	=1
	ORIGIN	Loop	
	LTORG		
	ADD	BREG	=1
	ADD	BREG	=2
	TEMP	DS	5
	COUNT	DC	3
	END		

OUTPUT:

SYMBOL TABLE

Sr. No	Symbol	Length	Value	Address
1	Loop	0	0	100
2	TEMP	5		104
3	COUNT	1	3	109

LITERAL TABLE

Sr. No	Literal	Address
1	=1	100
2	=1	110
3	=2	111

POOL TABLE

Sr. No	Pool Index
1	#1
2	#2

INTERMEDIATE CODE

```
            (AD,01)     (C,100)
100         (IS,01)     AREG        TEMP
102         (IS,02)     BREG        AREG
103         (IS,01)     AREG        COUNT
105         (IS,02)     AREG        (L,01)
            (AD,05)
            (AD,04)
102         (IS,02)     BREG        (L,02)
103         (IS,02)     BREG        (L,03)
104         (DL,01)     (C,5)
109         (DL,02)     (C,3)
            (AD,02)
```

Design of Pass 2 of a two pass assembler

OBJECTIVE: To understand how the intermediate code is converted to machine Language code by the assembler.

THEORY:

ALGORITHM:

```
1.   Code_area_address=address of code_area
     Pooltab_ptr=1;
     Loc_cntr=0;
2.   While next statement is not an END statement
a.   Clear  machine_code_buffer
b.   If an LTORG statement
i.   Process literals in
     LITTAB[POOLTAB[pooltab_ptr]….LITATB[POOLTAB[pooltab_ptr+1]]-1 similar to
     processing of constants in a DC statement i.e. assemble the literals in
     machine_code_buffer;
ii.  Size=size of memory area required for literals;
iii. Pooltab_ptr=pooltab_ptr+1;
c.   If a START or ORIGIN statement then
i.   Loc_cntr=value specified in operand field;
ii.  Size=0;
d.   If a declaration statement
i.   If a DC statement then
     Assemble the constant in machine_code_buffer;
ii.  Size=size=size of memory area required by DC/DS
e.   If an imperative statement
i.   Get operand address from SYMTAB or LITTAB
ii.  Assemble instruction in machine_code_buffer
iii. Size=size of  instruction
f.   If size not equal to 0 then
i.   Move contents of machine_code_buffer to the address code_area_address+loc_cntr
ii.  Loc_cntr=loc_cntr+size
```

g. (Processing of END statement)

i. Perform steps 2b and 2f

ii. Write code_area into output file

INPUT:

SYMBOL TABLE

Sr. No	Symbol	Length	Value	Address
1	Loop	0	0	100
2	TEMP	5		104
3	COUNT	1	3	109

LITERAL TABLE

Sr. No	Literal	Address
1	=1	100
2	=1	110
3	=2	111

POOL TABLE

Sr. No	Pool Index
1	#1
2	#2

INTERMEDIATE CODE

```
            (AD,01)     (C,100)
100         (IS,01)     AREG        TEMP
102         (IS,02)     BREG        AREG
103         (IS,01)     AREG        COUNT
105         (IS,02)     AREG        (L,01)
            (AD,05)
            (AD,04)
102         (IS,02)     BREG        (L,02)
103         (IS,02)     BREG        (L,03)
104         (DL,01)     (C,5)
109         (DL,02)     (C,3)
            (AD,02)
```

OUTPUT:

```
100         01          #1          104
102         02          #2          #1
103         01          #1          109
105         02          #1          100
100         =1
102         02          #2          110
103         02          #2          111
104
```

109	3
110	=1
111	=2

Assignment 3

AIM: Implementation of pass1 of a macro processor.

OBJECTIVE:

To understand the concept of Macro.

To study the Macro definition, Macro Call, Macro Expansion.

To know the different features of Macro Facility.

To understand the different databases used by macro processor along with their specifications

To understand how macro processor generates intermediate code from the source code.

THEORY:

Macro: Single line abbreviation for a group of instructions. Macro facility permits us to attach a name to a sequence of instructions and use it in their place.

Macro Definition

```
Start of Macro -------------------------→      MACRO
Macro Name ---------------------------→     [          ]
: Sequence to be abbreviated----------→          :
End of Macro---------------------------→     MEND
```

MACRO pseudo Opcode is the first line which identifies the following line as macro instruction name. Following is the sequence of instructions to be abbreviated followed by MEND which indicates the end of macro definition.

```
Eg.                     MACRO
                         INCR
                        ADD AREG, DATA
                        ADD BREG, DATA
                        ADD CREG, DATA
                        MEND
                        :
                        :
                        :
                        INCR
                        :
                        DATA DC '5'
                        :
                        :
        ADD CREG, DATA
```

This process of replacement of macro name with the sequence of instructions is called macro expansion.

Features of Macro Facility

1. Macro with Arguments

   All calls to the macro are replaced with identical blocks. To modify the coding during macro call, macro with arguments can be specified. Arguments are provided on the macro name line and precede with an ‘&’. They are called as dummy arguments. Arguments can be provided to a macro using two ways

   - Keyword arguments
     - Keyword arguments allow reference to dummy arguments by name as well as position.

```
MACRO
INCR &ARG1=DATA3,&ARG2-DATA1,&ARG3=DATA2
ADD AREG,&ARG1
ADD BREG,&ARG2
ADD CREG,&ARG3
MEND
        :
        :
INCR DATA1, DATA2, DATA3
        :
        :
INCR DATA3, DATA1, DATA2
```

   - Positional arguments
     - Arguments are matched with the dummy arguments according to the order in which they appear.

       Eg:

```
MACRO
INCR &ARG1,&ARG2,&ARG3
ADD AREG,&ARG1
ADD BREG,&ARG2
ADD CREG,&ARG3
MEND
        :
        :
INCR DATA1,DATA2,DATA3
```

```
    :
    :
INCR DATA3,DATA1,DATA2
```

Here when the macro is called for the first time, ARG1 is replaced with DATA1, ARG2 with DATA2 and ARG3 with DATA3,while in the second call ARG1 is replaced with DATA3, ARG2 with DATA1 and ARG2 respectively. Accordingly, while expansion the code is replaced in the expanded code.

2. Macro call within macro

```
MACRO
INCR&DATA1
ADD AREG,&DATA1
ADD BREG,&DATA1
MEND
MACRO
INCR1  &DATA1,&DATA2
INCR &DATA1
INCR &DATA2
MEND
:
:
INCR1 COUNT, COUNT1
:
:
COUNT DC 3
COUNT DC 4
:
:
```

Here, when INCR1 is called, inside its body there is one more call to INCR macro. So INCR will first get expanded.After encountering MEND for INCR, it will continue with the expansion of INCR1. Next line is again a call to INCR , so same will repeat and then after encountering MEND of INCR1 , it will terminate. So the final expanded code after expansion of INCR1 will be as

```
ADD AREG, COUNT
ADD BREG, COUNT
ADD AREG, COUNT1
ADD BREG, COUNT1
```

3. Macro Instruction Defining Macro

```
MACRO
OUTER &INNER
MACRO
&INNER  &ARG1
ADD AREG,&ARG1
SUB BREG, AREG
MEND
ADD AREG, BREG
MEND
    :
    :
OUTER FIRST
   :
   :
FIRST DATA1
   :
DATA1 DC 2
  :
```

Here, when OUTER macro is called, it will lead to the definition of the inner macro with the name FIRST and its entry will be made in the MDT and MNT.The instructions inside the inner macro will not appear in the expanded code when the OUTER macro is getting expanded. Only when FIRST macro is called, then its body will appear in the expanded code.

4. Conditional Macros

Macro facility provides two pseudo-ops for conditional reordering of the instruction sequence during expansion. This allows conditional selection of instructions that appear during expansion of macro call. The pseudo-op are

- AIF: it's a conditional pseudo-op that performs an arithmetic test and branches only if the tested condition is true.
- AGO:Unconditional branch pseudo-op or 'go to' statement.

Both the pseudo-op direct the macro processor to jump to a macro label called as Sequencing Symbol. The macro label begins with a period(.).

```
Eg.     MACRO
        INCR &COUNT,&ARG1,&ARG2,&ARG3
        MOVER AREG,&ARG1
        AIF (&COUNT EQ 1) .FINI
        MOVER AREG,&ARG2
        AGO .FINI
        MOVER BREG,&ARG3
.FINI   MEND
```

Here, if COUNT is set to 1,only MOVER AREG,&ARG1 instruction will appear in the expanded code, if COUNT is not 1, MOVER AREG,&ARG2 will also be appear in the expanded code, while MOVER BREG,&ARG3will never appear because of the AGO pseudo-op which is on the previous line.

Databases and their formats

Macro Definition Table

Sr. No	Label	Mnemonic	Operands

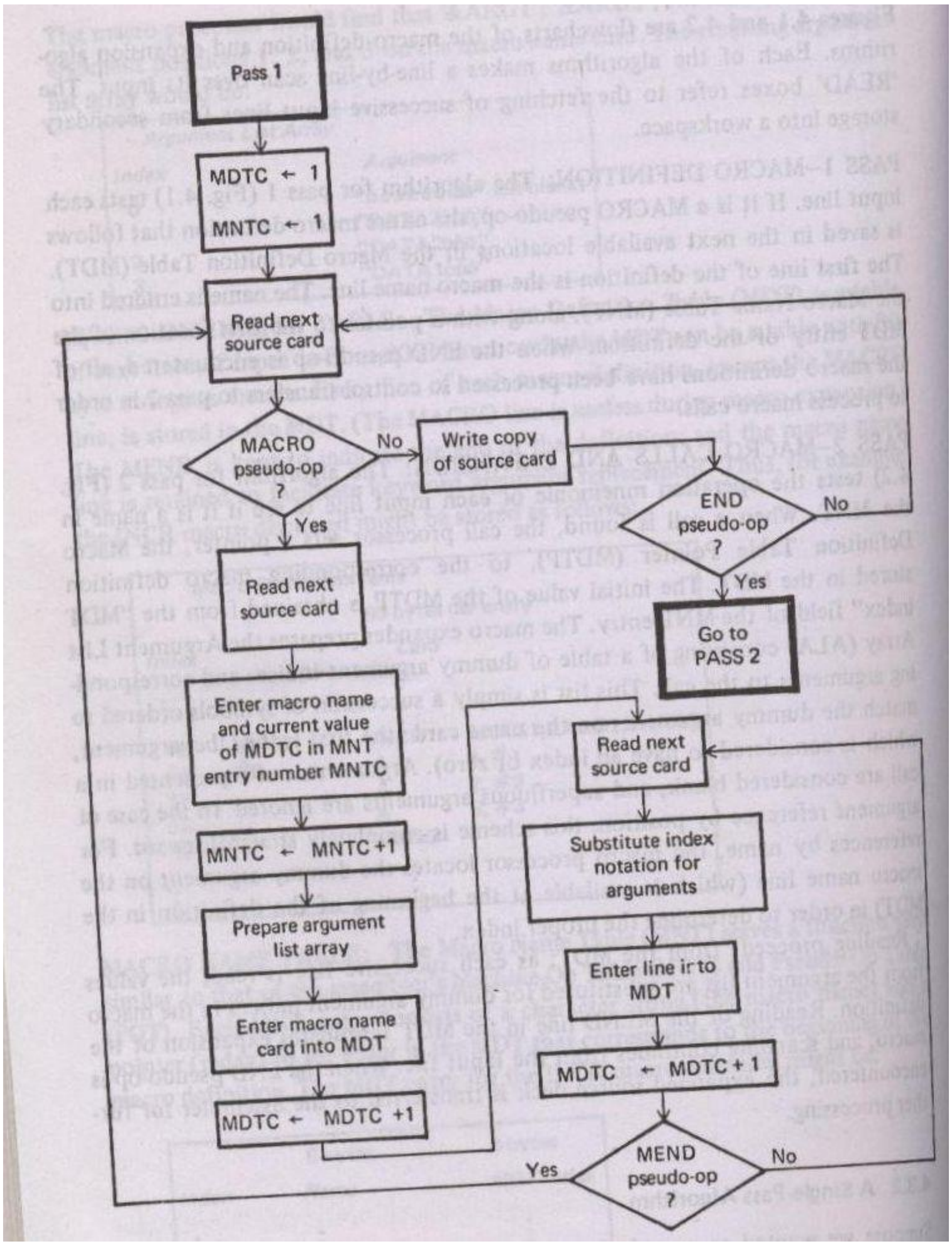
Macro Name Table

Sr. No	Macro Name	Index where Macro begins in MDT

ALA

Index	Arguments

flowchart



3Design of Pass1 of two pass macro

INPUT:

```
MACRO
FIRST
ADD AREG,BREG
MEND
MACRO
```

```
SECOND &ARG1,&ARG2
ADD AREG,&ARG2
SUB BREG,&ARG1
MEND
MACRO
THIRD &ARG1=DATA3,&ARG2=DATA1
ADD AREG,&ARG1
ADD BREG,&AGR2
SUB BREG,&ARG1
MEND
START
    :
    :
FIRST
    :
SECOND DATA1,DATA2
    :
    :
THIRD DATA2,DATA,DATA3
    :
DATA1       DS      3
DATA2       DS      2
DATA3       DC      '3'
 END
```

OUTPUT:

MACRO NAME TABLE (MNT)

Sr.No	Macro Name	Index
1	FIRST	1
2	SECOND	4
3	THIRD	8

3. Design of Pass1 of two pass macro processor

MACRO DEFINITION TABLE (MDT)

Sr. No	Label	Mnemonic	Operands	
1		FIRST		
2		ADD	AREG	BREG
3		MEND		

4		SECOND	&ARG1	&ARG2
5		ADD	AREG	#1
6		SUB	BREG	#2
7		MEND		
8		THIRD	&ARG1=DATA3	&ARG2=DATA1
9		ADD	AREG	#1
10		ADD	BREG	#2
11		SUB	BREG	#1
12		MEND		

INTERMEDIATE CODE

```
START
     :
     :
FIRST
     :

SECOND DATA1,DATA2
     :
     :
THIRD DATA2,DATA,DATA3


:
DATA1 DS       3
DATA2 DS       2
DATA3 DC       '3'
 END
```

Questions:

1. Compare process of developing an assembler in a single pass and using two passes.
2. Describe various databases used to developer a two pass assembler
3. 3. Distinguish between a pass and a phase

Assignment No:-3

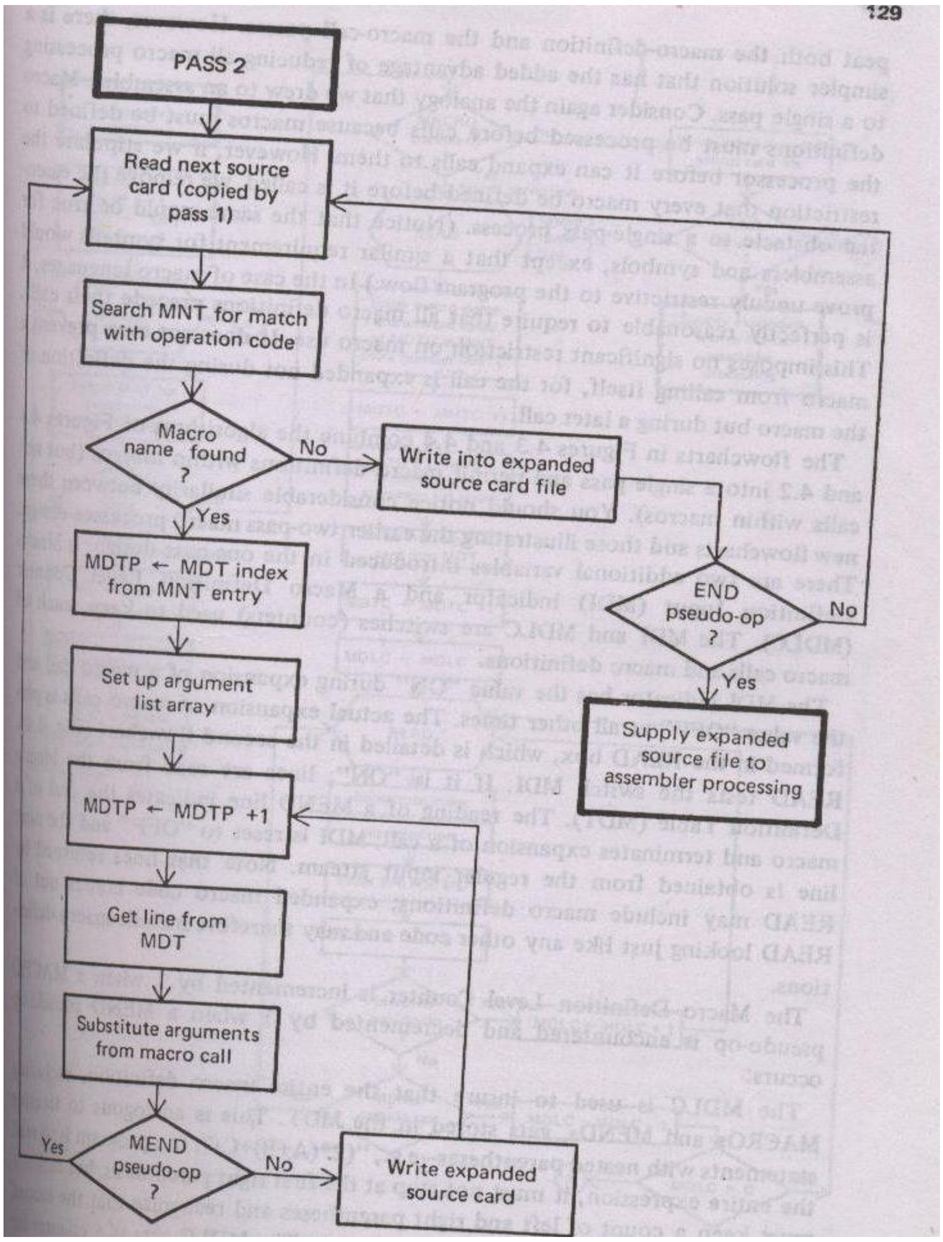
AIM:Implementation of pass2 of a macro processor.

OBJECTIVE:

To understand how macro processor generates expanded source code from intermediate code.

THEORY:

Flowchart Pass2


129
PASS 2
Read next source card (copied by pass 1)
Search MNT for match with operation code
Macro name found ?
No
Write into expanded source card file
Yes
MDTP ← MDT index from MNT entry
END pseudo-op ?
No
Yes
Set up argument list array
Supply expanded source file to assembler processing
MDTP ← MDTP +1
Get line from MDT
Substitute arguments from macro call
Yes
MEND pseudo-op ?
No
Write expanded source card

INPUT:

MACRO NAME TABLE (MNT)

Sr.No	Macro Name	Index
1	FIRST	1
2	SECOND	4
3	THIRD	8

MACRO DEFINITION TABLE (MDT)

Sr. No	Label	Mnemonic	Operands	
1		FIRST		
2		ADD	AREG	BREG
3		MEND		
4		SECOND	&ARG1	&ARG2
5		ADD	AREG	#1
6		SUB	BREG	#2
7		MEND		
8		THIRD	&ARG1=DATA3	&ARG2=DATA1
9		ADD	AREG	#1
10		ADD	BREG	#2
11		SUB	BREG	#1
12		MEND		

INTERMEDIATE CODE (IC)

```
START
    :
    :
FIRST
    :
SECOND DATA1,DATA2
    :
    :
THIRD DATA3,DATA1
    :
DATA1      DS     3
DATA2      DS     2
DATA3      DC     '3'
 END
```

OUTPUT:

```
START
:
:
 ADD          AREG            BREG
:

 ADD          AREG            DATA1
 SUB          BREG            DATA2
:
:
 ADD          AREG            DATA3
 ADD        BREG         DATA1
 SUB        BREG         DATA3

:
DATA1      DS    3
DATA2      DS    2
DATA3      DC    ‘3’
 END
```

Questions:

\\

1. List advantages of integrating a Microprocessor in pass1 of an assembler.
2. List features of MASM
3. Describe the role of intermediate file in developing a two pass microprocessor
4. Distinguish between a Macro and a Function.

Assignment No:-4

Aim: Demonstrate use of Dynamic Link Libraries

Objective:

- To write an application in VC++ to create a Dynamic Link Library (DLL).
- To understand how to use appwizard to create MFC based DLLs.
- To understand how to enable support using your DLL as an automation inpro C server.

THEORY:

In computer science, a **library** is a collection of resources used to developsoftware. These may includesubroutines, classes, values or typespecifications.

Any library is categorized into two types:

- Static Library.
- Dynamic Library.

Static Library:

Originally, only *static* libraries existed. A static library, also known as an *archive*, consists of a set of routines which are copied into a target application by the compiler, linker, or binder, producing object files and a stand-alone executable file. This process, and the stand-alone executable file, are known as a static build of the target application. Actual addresses for jumps and other routine calls are stored in a relative or symbolic form which cannot be resolved until all code and libraries are assigned final static addresses.

Dynamic Library or Dynamic Link Library:

A *dynamic-link library* (DLL) is a module that contains functions and data that can be used by another module (application or DLL).

DLLs provide a way to modularize applications so that their functionality can be updated and reused more easily. DLLs also help reduce memory overhead when several applications use the same functionality at the same time, because although each application receives its own copy of the DLL data, the applications share the DLL code.The advantage of DLL files is that, because they don't get loaded into random access memory (RAM) together with the main program, space is saved in RAM.

Types of dynamic linking:

- load-time dynamic linking
- run time dynamic linking

Load-time dynamic linking:

In load time dynamic linking, a module makes explicit calls to exported DLL functions. This requires you to link the module with the import library for the DLL. An import library supplies the system with the information needed to load the DLL and locate the exported DLL functions when the application is loaded

If the system cannot locate a required DLL, it terminates the process and displays a dialog box that reports the error to the user. Otherwise, the system maps the DLL into the virtual address space of the process and increments the DLL reference count.

Run-time dynamic linking:

In run-time dynamic linking, a module uses the load library function to load the DLL at run time. After the DLL is loaded the module calls the Get Proc Address functions to get the address of the exported DLL functions. The module calls the exported DLL functions using the function pointers returned by Get Proc Address. This eliminates the need for an import library.

Difference between load time dynamic linking and run time dynamic linking load-time linking is when symbols in the library, referenced by the executable (or another library) are handled when the executable/library is loaded into memory, by the operating system.

Run-time linking is when you use an API provided by the OS or through a library to load a DLL or DSO when you need it, and perform the symbol resolution then.

PROCEDURE:

1. Open Microsoft Visual Studio and click on File – New to open a new window.
2. Now click on Win32 Dynamic Link Library file.
3. Give a sample project name (eg. Test) and click OK. The next window of Win32 Dynamic Link Library will appear.
4. Click a simple DLL project option and click on finish button. The Win32 DLL file will be created.
5. Click OK in this window. The workspace for the sample project will be created.
6. At the end of test.cpp write following code :

```
int_stdcall sum(int x, int y)
{
returnx+y;
}
int_stdcall diff(int x, int y)
{
return x-y;
}
int_stdcallmult(int x, int y)
{
return x*y;
}
        int_stdcall division(int x, int y)
        {
        return x/y;
        }
```

10. Dynamic Link Library

7. This code will serve as a entry point for the DLL. The actual functions are defined in this file.
8. Now click on File – New, text file. Give a sample filename with extension .def (eg. Test.def). The file will be created . Then write the following code in it.

```
LIBRARY test
EXPORTS
sum @1
diff @2
mult @3
division  @4
```

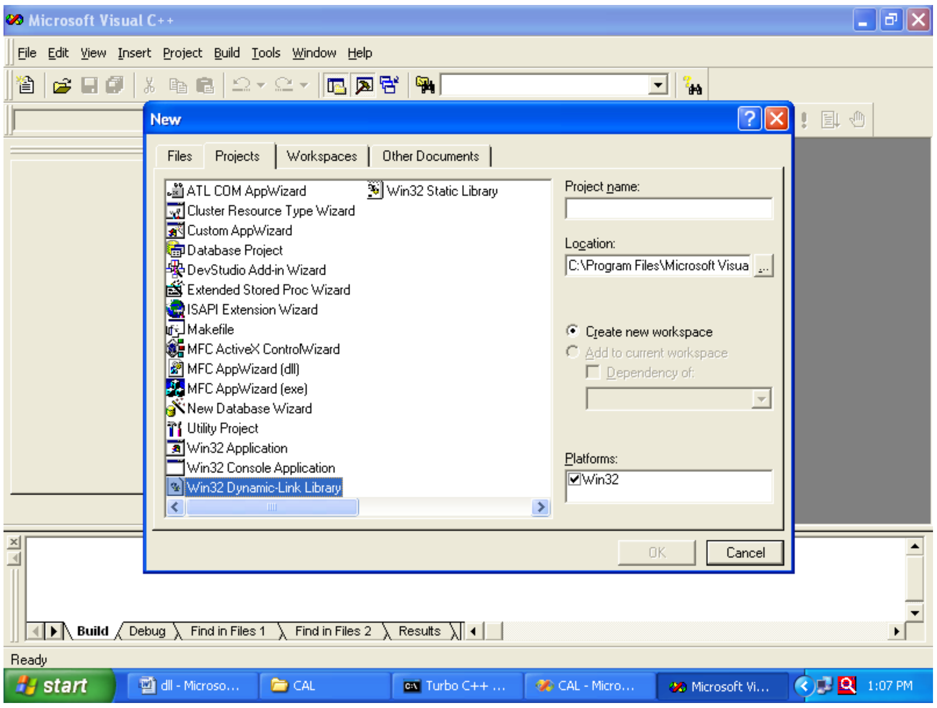
9. Click on build option of menu bar and click on build test.dll. The test.dll will get created. This dll file will be in test folder. Copy this test.dll to directory WINDOWS.
10. Close Microsoft VC++ and open Visual Basic. On the form place three text windows namely text1, text2, text3 and place four command buttons for arithmetic operations and one command button for exit.
11. Write the following code in VB

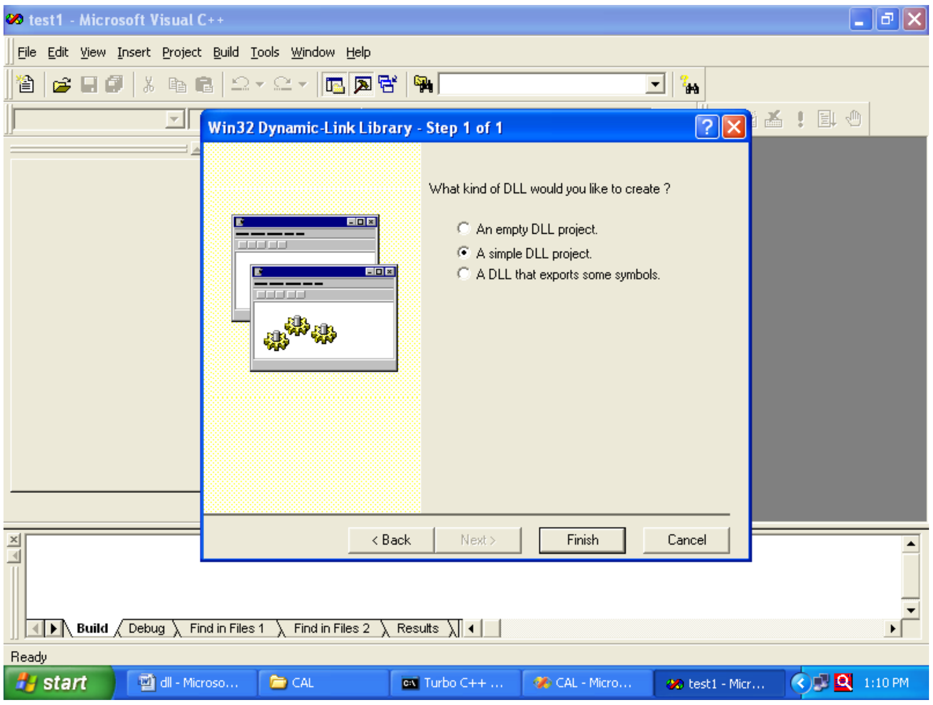
```
Private Declare Function sum Lib "test.dll"(ByVal x as Long, ByVal y as Long) As
Long
Private Declare Function diff  Lib "test.dll"(ByVal x as Long, ByVal y as Long) As
Long
Private Declare Function mult  Lib "test.dll"(ByVal x as Long, ByVal y as Long) As
Long
Private Declare Function division Lib "test.dll"(ByVal x as Long, ByVal y as Long)
As Long
Private Sub cmd_add_Click()
Text3.Text = Str(sum(Cint(Text1.Text), Cint(Text2.Text)))
End Sub
Private Sub cmd_add_Click()
Text3.Text = Str(diff(Cint(Text1.Text), Cint(Text2.Text)))
End Sub
Private Sub cmd_add_Click()
Text3.Text = Str(mult(Cint(Text1.Text), Cint(Text2.Text)))
End Sub
Private Sub cmd_add_Click()
```

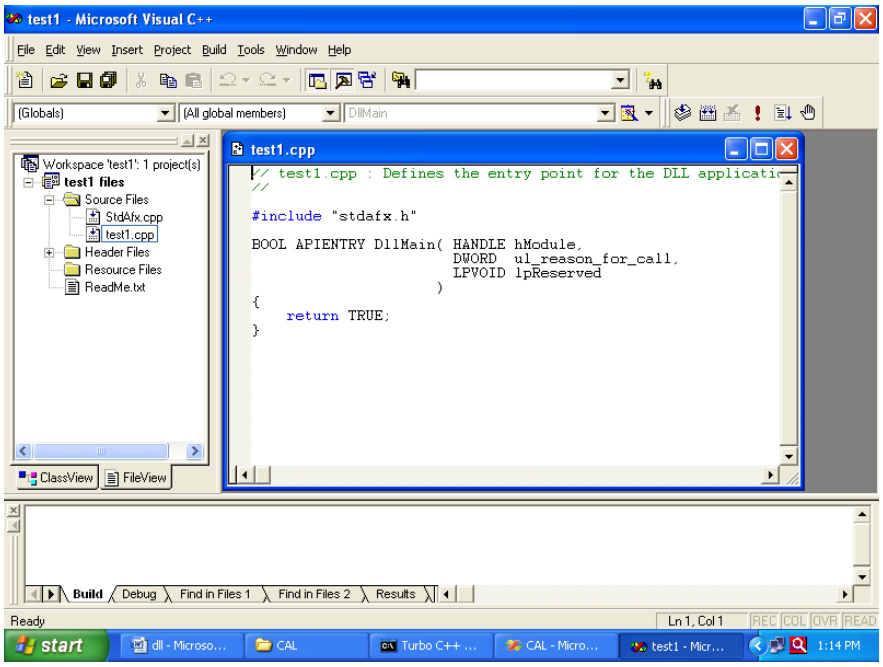
Text3.Text = Str(division(Cint(Text1.Text), Cint(Text2.Text)))
End Sub

12. Run VB program. Note thet the DLL has been created in VC++ and used in VB.

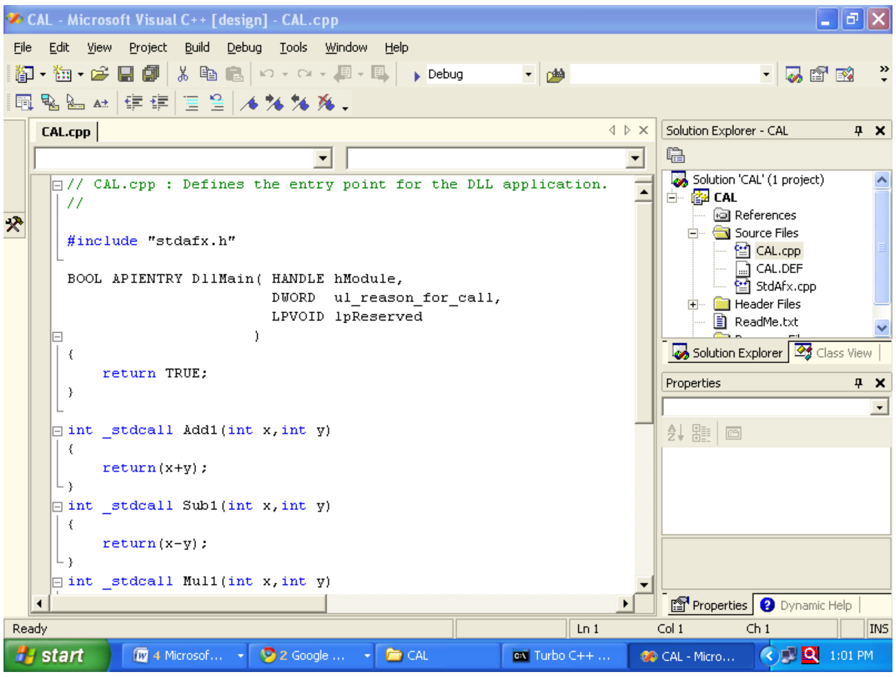
INPUT :


Microsoft Visual C++
File Edit View Insert Project Build Tools Window Help
New
Files Projects Workspaces Other Documents
ATL COM AppWizard
Cluster Resource Type Wizard
Custom AppWizard
Database Project
DevStudio Add-in Wizard
Extended Stored Proc Wizard
ISAPI Extension Wizard
Makefile
MFC ActiveX ControlWizard
MFC AppWizard (dll)
MFC AppWizard (exe)
New Database Wizard
Utility Project
Win32 Application
Win32 Console Application
Win32 Dynamic-Link Library
Win32 Static Library
Project name:
Location:
C:\Program Files\Microsoft Visua
Create new workspace
Add to current workspace
Dependency of:
Platforms:
Win32
OK
Cancel
Build Debug Find in Files 1 Find in Files 2 Results
Ready
start dll - Microso... CAL Turbo C++ ... CAL - Micro... Microsoft Vi... 1:07 PM

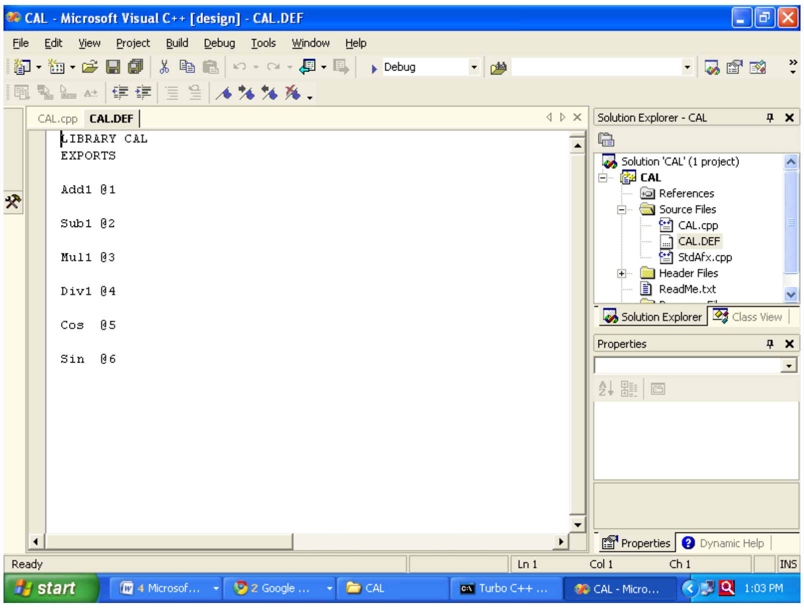

test1 - Microsoft Visual C++
File Edit View Insert Project Build Tools Window Help
Win32 Dynamic-Link Library - Step 1 of 1
What kind of DLL would you like to create ?
An empty DLL project.
A simple DLL project.
A DLL that exports some symbols.
< Back Next > Finish Cancel
Build Debug Find in Files 1 Find in Files 2 Results
Ready
start dll - Microso... CAL Turbo C++ ... CAL - Micro... test1 - Micr... 1:10 PM



```
// test1.cpp : Defines the entry point for the DLL applicati
//

#include "stdafx.h"

BOOL APIENTRY DllMain( HANDLE hModule,
                       DWORD  ul_reason_for_call,
                       LPVOID lpReserved
                     )
{
    return TRUE;
}
```

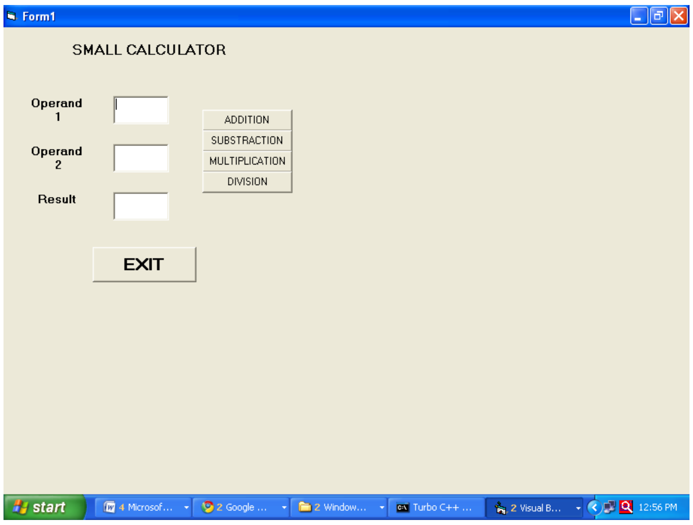


```
// CAL.cpp : Defines the entry point for the DLL application.
//

#include "stdafx.h"

BOOL APIENTRY DllMain( HANDLE hModule,
                       DWORD  ul_reason_for_call,
                       LPVOID lpReserved
                     )
{
    return TRUE;
}

int _stdcall Add1(int x,int y)
{
    return(x+y);
}
int _stdcall Sub1(int x,int y)
{
    return(x-y);
}
int _stdcall Mul1(int x,int y)
```


CAL - Microsoft Visual C++ [design] - CAL.DEF
File Edit View Project Build Debug Tools Window Help
CAL.cpp CAL.DEF
Solution Explorer - CAL
Solution 'CAL' (1 project)
CAL
References
Source Files
CAL.cpp
CAL.DEF
StdAfx.cpp
Header Files
ReadMe.txt
Solution Explorer Class View
Properties
Properties Dynamic Help
Ready Ln 1 Col 1 Ch 1 INS

```
LIBRARY CAL
EXPORTS

Add1 @1

Sub1 @2

Mul1 @3

Div1 @4

Cos  @5

Sin  @6
```

OUTPUT:



Questions:

1. Distinguish between Static linking and Dyanamic Linking.
2. List the benefits of using DLLs.
3. Define OLE

Assignment No.:5

Aim: Develop a lexical analyser using Lex.

Objective:

- To write a program in 'C' language to implement Lexical Analyzer for the subset of 'C' language.
- To study the role and significance of the lexical analyzer in compiler analysis phase.
- To implement a simple lexical analyzer in 'C' language to perform a lexical analysis of a sample C program.

THEORY:

Lexical Analysis:

Introduction

Lexical analysis is the first stage in the compilation of a source program written in high-level language like C or C++. The lexical analyzer reads the input source program and produces as output, a sequence of tokens that the parser uses for syntax analysis.

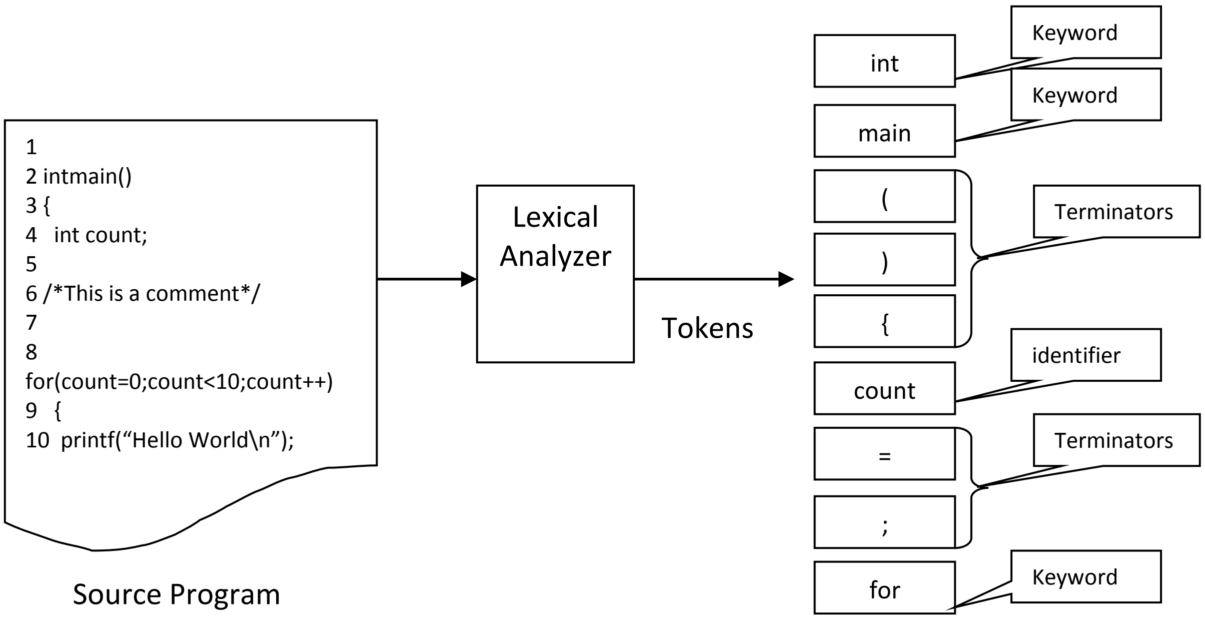


Fig 4.1 Lexical Analysis

For example consider a simple C program as input to the lexical analyzer. The lexical analyzer separates the input C program into various types of tokens like keywords, identifiers, operators, and so on as shown in the above figure 4.1.

6. Lexical Analyser

Comments and white space (like tab, blank, new line) do not influence code generation. The lexical analyzer strips out the comments and white space in the source program.

Token: A token is a string of characters, categorized according to the rules as a symbol (e.g. IDENTIFIER, NUMBER, COMMA, etc.). A token is a substring of the input string that represents a basic element of the language. The process of forming tokens from an input stream of characters is called tokenization and the lexer categorizes them according to a symbol type. A token can look like anything that is useful for processing an input text stream or text file.

Lexeme: The part of the input stream that qualifies for a certain type of token is called as lexeme.

Procedure

The action of parsing the source program into proper syntactic classes is known as lexical analysis. The source program is scanned sequentially. The basic elements or tokens are delineated by blanks, operators, and special symbols and thereby recognized as identifiers, literals, or terminal symbols (operators, keywords, etc).

The basic elements identifiers and literals are placed into Identifier table and Literal table respectively. As other phases recognize the use and meaning of the elements, further information is added into these tables (e.g., precision, data type, length, etc).

Uniform symbol table is created which contains of uniform symbol. Uniform symbol is conversion the source string which contains information about the basic elements. Uniform symbols are of fixed size and consist of the syntactic class and pointer to the table entry of the associated basic element. The uniform symbol is the same length whether the token is 1 or 31 characters long. Other phases of compiler deal mainly with the small uniform symbol, but they can access any attribute of the token by following the pointer. Figure 4.2 depicts the uniform symbol for users.

CLASS	PTR

CLASSES OF UNIFORM SYMBOLS:

IDENTIFIER (IDN)
TERMINAL SYMBOL (TRM)
LITERAL (LIT)

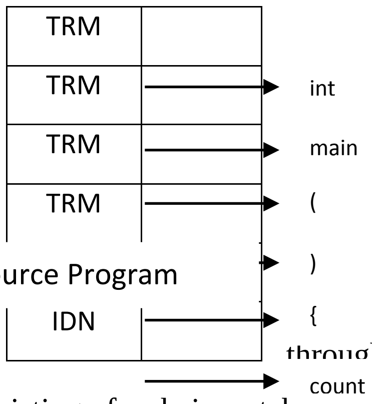


Fig 4.2 Uniform Symbols of Example Source Program

6. Lexical Anal

This lexical process can be done in one continuous pass through the data by creating an intermediate form of the program consisting of a chain or table of tokens.

Tasks

The three task of lexical analysis phase are:

1. To parse the source program into the basic elements or tokens of the language.
2. To build a literal table and an identifier table.
3. To build a uniform symbol table.

Databases

These tasks involve manipulation of five databases. Possible forms of these are:

1. Source Program- original form of program; appears to the compiler as a string of characters.
2. Terminal Table- a permanent database that has an entry for each terminal symbol (e.g., arithmetic operators, keywords, symbols).

Symbol	Indicator	Precedence

3. Literal Table- created by lexical analysis to describe all literals used in the source program. There is only one entry of each literal, consisting of a value, a number of attributes, an address denoting the location of the literal at execution time and other information (if needed).

Literal	Base	Scale	Precision	Other Information	Address

4. Identifier Table- created by lexical analysis to describe all identifiers used in the source program. There is one entry for each identifier. Lexical analysis creates the entry and places the name of the identifier into that entry. Later phases will fill in the data attributes and address of each identifier.

Name	Data attributes	Address

5. Uniform Symbol Table- created by lexical analysis to represent the program as a string of tokens rather than of individual characters. Spaces and comments in the source code are not represented by uniform symbols and are not used by future phases. There is one uniform symbol for every token in the program. Each uniform symbol contains the identification of the table of which the token is a member (e.g., a pointer to the table or a code) and its index within that table.

Table	Index

Algorithm

The first task of the lexical analysis algorithm is to parse the input character string into tokens. The second task is to make the appropriate entries in the table. Implementation of this phase is described below:

- The input string is separated into tokens by break characters or blank space. Blank space may serve as break characters but are otherwise ignored. Break characters are permanently defined in the terminal table. While scanning the characters of the input string are stored in a temporary array until they match a rule of token.
    - Lexical analysis recognizes three types of tokens: terminal symbols, possible identifiers, and literals. It checks all tokens by first comparing them with the entries in the terminal table. Once a match is found, the token is classified s a terminal symbol and lexical analysis creates a uniform symbol of type 'TRM', and inserts it in the uniform symbol table. If a token is not a terminal symbol, lexical analysis proceeds to classify it as a possible identifier or literal. Those tokens that satisfy the lexical rules for identifiers are classified as "possible identifiers." If a token does not fit into the rule, it is an error and is flagged as such.
    - After a token is classified as a "possible identifier," the identifier table is examined. If this particular token is not in the table, a new entry is made. At this phase only name

of the identifier goes into the table, the remaining information is discovered and inserted by later phases. Regardless of whether or not an entry had to be created in uniform symbol table, lexical analysis creates a uniform symbol of type 'IDN' and inserts it into the uniform symbol table.

- Numbers, quoted character strings, and other self-defining data are classifies as "literals." After a token has been classified as such, the literal table is examined. If the literal is not yet there, a new entry is made. Regardless of whether or not an entry had to be created in uniform symbol table, lexical analysis creates a uniform symbol of type 'LIT' and inserts it into the uniform symbol table.

The lexical analysis makes one complete pass over the source code and produces the entire uniform symbol table.

Assumption for implementation purpose

- Source Program- Simple 'C' language program is take as input. The source program is saved in text file.
- Terminal Table-

Index	Symbol

- Literal Table-

Index	Literal

- Identifier Table-

Index	Name

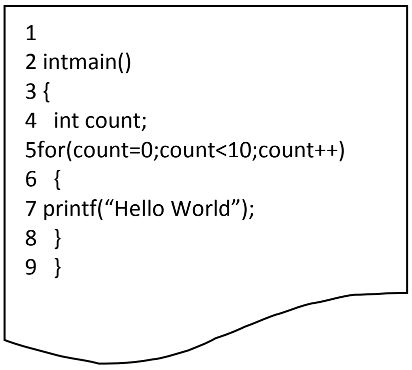
- Uniform Symbol Table-

Token	Class	Index

INPUT:

1. A simple 'C' language program saved in text file is given as input to lexical analyzer program.
2. Terminal table permanently saved for the use of lexical analysis program.

Example:



```

intmain()
{
  int count;
for(count=0;count<10;count++)
  {
printf("Hello World");
  }
  }
```

Source program saved as text file

Index	Symbol
1	Int
2	Main
3	(
4	)
5	for
6	=
7	;
8	<
9	++
10	{
11	}
12	printf
13	“”

Terminal Table

OUTPUT:

Output of the lexical analysis program will be three tables:

1. Identifier table
2. Literal table
3. Uniform symbol table

Example:

Index	Name
1	count

Identifier Table

Index	Literal
1	0
2	10
3	“Hello World”

Literal Table

Token	Class	Index
Int	TRM	1
Main	TRM	2
(	TRM	3
)	TRM	4

{	TRM	10
int	TRM	1
count	IDN	1
;	TRM	7
for	TRM	5
(	TRM	3
count	IDN	1
=	TRM	6
0	LIT	1
;	TRM	7

Uniform Symbol Table

Questions:

1. Describe the phases of compiler.
2. Describe the significance of Lex and YACC to develope a compiler.
3. List advantages of using Intermediate Code in development of compller
4. Explain the terms compiler-compiler

Assignment No:6

Aim: Study of Shell Programming basics

Shell: A*shell* (or *command interpreter*, or *command prompt*) is a program that lets you interact with the operating system by issuing text-based commands. It is called a shell, as it protects or shields y from interacting with the operating system directly. In addition to a set of commands, a shell also comes with its scripting language to write shell scripts (or shell programs), which could be a sequence of commands for automating system administration tasks.

There are many shell programs: from the legacy and obsoleted sh (the original Bourne Shell), csh (C Shell), ksh (Korn Shell), zsh (Z Shell), to newer bash (Bourne Again Shell), tcsh (Tenex C Shell) and tsh (T Shell); and Windows cmd shell.

Bash shell supports *thousands* of commands - the philosophy of Unix is using small programs that perform one task, but performed it well. Bash shell is often identified by a "$" sign in the command prompt.+

Shell Command

A program that interprets commands

Allows a user to execute commands by typing them manually at a terminal, or automatically in programs called shell scripts.

A shell is not an operating system. It is a way to interface with the operating system and run commands.

Bootstrap loader small piece of code locates the kernel, loads it into memory, and starts it Sometimes 2-step process is used instead 1. Simple bootstrap loader in ROM loads a more comp

Types of Shell

The UNIX shell is of two major types

- BASH Cell
- BASH = Bourne Again Shell

Bash is a shell written as a free replacement to the standard Bourne Shell (/bin/sh) originally written by Steve Bourne for UNIX systems.

It has all of the features of the original Bourne Shell, plus additions that make it easier to program with and use from the command line.

Shell Scripts

A **shell script** is a computer program designed to be run by the **Unix shell**, a command-line interpreter. The various dialects of **shell scripts** are considered to be **scripting** languages. Typical operations performed by **shell scripts** include file manipulation, program execution, and printing text.

Shell programming is a basic skill that every UNIX System Administrator should have. The Systems Administrator must be able to read and write shell programs because

- There are many tasks that can and should be quickly automated by using shell programs,
- Many software products come with install scripts that have to be modified for your system before they will work.

Shell Program

?

Simply put, a shell program (sometimes called a shell script) is a text file that contains standard UNIX and shell commands. Each line in a shell program contains a single UNIX command exactly as if you had typed them in yourself. The difference is that you can execute all the commands in a shell program simply by running the shell program (rather than typing all the commands within).

Shell programs are interpreted and not compiled programs. This means when you run a shell program a child shell is started. This child shell reads each line in the shell program and carries out the command on that line. When it has read all the commands the child shell finishes.

Support for programming

To be truly useful a programming language must provide the following services

- comments,
- variables,
- conditional commands, and
- Repeated action commands.

These extra services are provided by the shell. Different shells use different syntax for these services. This means that a shell program written for the Bourne shell cannot be run by the C shell (though it can be run by the `bash` shell.)

For portability, we will use only the Bourne shell syntax.

Creating a shell program

To create a shell program you need to complete the following steps:

- create a text file to hold the shell program
- decide which shell you will use
- add the required commands to the file
- save the file
- change the permissions on the file so it is executable
- run the shell program

Which shell?

The first step is to decide which shell to use. You decision here will control the syntax of the shell commands you use. For the purposes of this subject you should always use the Bourne shell. The standard Linux shell bash supports the Bourne shell syntax.

Once you've made that decision the first line of your shell program should indicate which shell should be used to interpret your shell program. This is done by having the **first line** of the shell program have the following: #!*path_to_shell*
For example, suppose that on my machine I want to use the Bourne shell (sh). The full path to the sh program on my system is /bin/sh (this will be the same on virtually all Unix systems). So the first line of every shell program I write will be: #!/bin/sh

Add the commands

The next step is to add the commands you wish the shell program to perform. The commands can be any standard UNIX or shell command. Of course any shell commands you use must be supported by the shell that will interpret your shell program.

Make the file executable

Once you've added all the commands you can then save the file. In order to execute a shell program both the read and execute permissions must be set. By default when you save a text file the execute file permission will not be set. You will have to set this manually

For example

I want to find out the number of files and directories in the current directory so I have created the following shell program called files

```
#!/bin/sh
ls | wc -w
```

Create the files shell program and execute it.

Running a shell program

By default a shell reads commands from standard input. This is the mode of operation you are used to. It is also possible for the shell to read commands from a file. For example
unx1:~$ **sh files**will start a new shell but rather than reading commands from standard input the shell will read its commands from the file called files. (Created in the example in the previous section).

Executing a shell program

The other method to execute a shell program is to enter its name in the same way as you would execute any other command. For example to run the files shell script you would simply type

unx1:~$ **files**

Which shell

The first line of the files shell program indicates which shell should be used to interpret the commands. However it is not compulsory to include this line.

If you do not specify which shell should be used your current shell will be used. If the first line of your shell script is blank, most versions of Unix will use the Bourne shell.

Shell program names

A common mistake for many new shell programmers is to create a shell program called test and then wonder why it doesn't work as expected.

Execute the following command

/usr/bin/which test

Do you see the problem? There is already a command called test and it is the command that is usually found first when you try to run a command called test. Use which when you have a question as to whether you are running the correct shell program.

File permissions

Anyone you wish to execute your shell programs should have at least read permission on the shell program's file. They must have read permission so that a shell they run can read the file in order to interpret the commands.

If they wish to execute the shell program simply by typing its name they must also have execute permission. It is possible to execute a shell program using the *shell filename* format without execute permission.

Questions:

1. Draw and Explain structure of operating system
2. Define shell. List and explain different types of shell
3. Describe evolution of operating system.

Assignment No:7

Aim: Demonstrate the use of VI editor commands.

What is vi?

The default editor that comes with the UNIX operating system is called vi (**vi**sual editor). Editors for UNIX environments include pico and emacs, a product of GNU.] The UNIX vi editor is a full screen editor and has two modes of operation:

1. *Command mode* commands which cause action to be taken on the file, and
2. *Insert mode* in which entered text is inserted into the file.

In the command mode, every character typed is a command that does something to the text file being edited; a character typed in the command mode may even cause the vi editor to enter the insert mode. In the insert mode, every character typed is added to the text in the file; pressing the <Esc> (*Escape*) key turns off the Insert mode.
While there are a number of vi commands, just a handful of these is usually sufficient for beginning vi users. To assist such users, this Web page contains a sampling of basic vi commands. The most basic and useful commands are marked with an asterisk (* or star) in the tables below. With practice, these commands should become automatic.
NOTE: Both UNIX and vi are **case-sensitive**. Be sure not to use a capital letter in place of a lowercase letter; the results will not be what you expect.

To Get Into and Out Of vi

To Start vi

To use vi on a file, type in vi filename. If the file named filename exists, then the first page (or screen) of the file will be displayed; if the file does not exist, then an empty file and screen are created into which you may enter text.

*	**vi filename**	*edit filename starting at line 1*
	vi -r filename	*recover filename that was being edited when system crashed*

To Exit vi

Usually the new or modified file is saved when you leave vi. However, it is also possible to quit vi without saving the file.
Note: The cursor moves to bottom of screen whenever a colon (:) is typed. This type of command is completed by hitting the <Return> (or <Enter>) key.

*	**:x<Return>**	*quit vi, writing out modified file to file named in original invocation*
	:wq<Return>	*quit vi, writing out modified file to file named in original invocation*
	:q<Return>	*quit (or exit) vi*
*	**:q!<Return>**	*quit vi even though latest changes have not been saved for this vi call*

Moving the Cursor

Unlike many of the PC and MacIntosh editors, **the mouse does not move the cursor** within the vi editor screen (or window). You must use the the key commands listed below. On some UNIX platforms, the arrow keys may be used as well; however, since

vi was designed with the Qwerty keyboard (containing no arrow keys) in mind, the arrow keys sometimes produce strange effects in vi and should be avoided.
If you go back and forth between a PC environment and a UNIX environment, you may find that this dissimilarity in methods for cursor movement is the most frustrating difference between the two.
In the table below, the symbol ^ before a letter means that the <Ctrl> key should be held down while the letter key is pressed.

* **j*or*<Return> [*or* down-arrow]**	*move cursor down one line*
* **k [*or* up-arrow]**	*move cursor up one line*
* **h*or*<Backspace> [*or* left-arrow]**	*move cursor left one character*
* **l*or*<Space> [*or* right-arrow]**	*move cursor right one character*
* **0 (zero)**	*move cursor to start of current line (the one with the cursor)*
* **$**	*move cursor to end of current line*
W	*move cursor to beginning of next word*
B	*move cursor back to beginning of preceding word*
:0<Return>*or* 1G	*move cursor to first line in file*
:n<Return>*or* nG	*move cursor to line n*
:$<Return>*or* G	*move cursor to last line in file*

Screen Manipulation

The following commands allow the vi editor screen (or window) to move up or down several lines and to be refreshed.

^f	*move forward one screen*
^b	*move backward one screen*
^d	*move down (forward) one half screen*
^u	*move up (back) one half screen*
^l	*redraws the screen*
^r	*redraws the screen, removing deleted lines*

Adding, Changing, and Deleting Text

Unlike PC editors, you cannot replace or delete text by highlighting it with the mouse. Instead use the commands in the following tables.
Perhaps the most important command is the one that allows you to back up and *undo* your last action. Unfortunately, this command acts like a toggle, undoing and redoing your most recent action. You cannot go back more than one step.

*** u** *UNDO WHATEVER YOU JUST DID; a simple toggle*

The main purpose of an editor is to create, add, or modify text for a file.

Inserting or Adding Text

The following commands allow you to insert and add text. Each of these commands puts the vi editor into insert mode; thus, the <Esc> key must be pressed to terminate the entry of text and to put the vi editor back into command mode.

*** i**	*insert text before cursor, until <Esc> hit*
I	*insert text at beginning of current line, until <Esc> hit*
*** a**	*append text after cursor, until <Esc> hit*
A	*append text to end of current line, until <Esc> hit*
*** o**	*open and put text in a new line below current line, until <Esc> hit*
*** O**	*open and put text in a new line above current line, until <Esc> hit*

Changing Text

The following commands allow you to modify text.

*** R**	*replace single character under cursor (no <Esc> needed)*
R	*replace characters, starting with current cursor position, until <Esc> hit*
cw	*change the current word with new text, starting with the character under cursor, until <Esc> hit*
cNw	*change N words beginning with character under cursor, until <Esc> hit; e.g., c5w changes 5 words*
C	*change (replace) the characters in the current line, until <Esc> hit*
Cc	*change (replace) the entire current line, stopping when <Esc> is hit*
Ncc*or*cNc	*change (replace) the next N lines, starting with the current line, stopping when <Esc> is hit*

Deleting Text

The following commands allow you to delete text.

*** X**	*delete single character under cursor*
Nx	*delete N characters, starting with character under cursor*
dw	*delete the single word beginning with character under cursor*
dNw	*delete N words beginning with character under cursor; e.g., d5w deletes 5 words*
D	*delete the remainder of the line, starting with current cursor position*
*** Dd**	*delete entire current line*
Ndd*or*dNd	*delete N lines, beginning with the current line; e.g., 5dd deletes 5 lines*

Cutting and Pasting Text

The following commands allow you to copy and paste text.

Yy	*copy (yank, cut) the current line into the buffer*
Nyy*or*yNy	*copy (yank, cut) the next N lines, including the current line, into the buffer*
P	*put (paste) the line(s) in the buffer into the text after the current line*

Other Commands

Searching Text

A common occurrence in text editing is to replace one word or phrase by another. To locate instances of particular sets of characters (or strings), use the following commands.

/string	*search forward for occurrence of string in text*

?string	*search backward for occurrence of string in text*
n	*move to next occurrence of search string*
N	*move to next occurrence of search string in opposite direction*

Determining Line Numbers

Being able to determine the line number of the current line or the total number of lines in the file being edited is sometimes useful.

:.=	*returns line number of current line at bottom of screen*
:=	*returns the total number of lines at bottom of screen*
^g	*provides the current line number, along with the total number of lines, in the file at the bottom of the screen*

Saving and Reading Files

These commands permit you to input and output files other than the named file with which you are currently working.

:r filename<Return>	*read file named filename and insert after current line (the line with cursor)*
:w<Return>	*write current contents to file named in original vi call*
:w newfile<Return>	*write current contents to a new file named newfile*
:12,35w smallfile<Return>	*write the contents of the lines numbered 12 through 35 to a new file named smallfile*
:w! prevfile<Return>	*write current contents over a pre-existing file named prevfile*

Questions:

1. Define Vi editor and Explain it's modes of operation.
2. Explain screen manipulation commands
3. Use vi commands to create a sample file and perform read /write operations.

Assignment No:8

Aim:Use Linux and compute average of N numbers.

Code for Write a shell script to find the average of the numbers entered in command line in Unix / Linux / Ubuntu

```
sum=0
for i in $*
do
sum=`expr $sum + $i`
done
avg=`expr $sum / $n`
echo Average=$avg
```

Questions:

1. Develop a program to check whether the number is even or odd.
2. Develop a to convert the temperature from Degree Celsius to Fahrenheit

Assignment No: 9

Aim: Study of device drivers.

The CPU is not the only intelligent device in the system, every physical device has its own hardware controller. The keyboard, mouse and serial ports are controlled by a SuperIO chip, the IDE disks by an IDE controller, SCSI disks by a SCSI controller and so on. Each hardware controller has its own control and status registers (CSRs) and these differ between devices. The CSRs for an Adaptec 2940 SCSI controller are completely different from those of an NCR 810 SCSI controller. The CSRs are used to start and stop the device, to initialize it and to diagnose any problems with it. Instead of putting code to manage the hardware controllers in the system into every application, the code is kept in the Linux kernel. The software that handles or manages a hardware controller is known as a device driver. The Linux kernel device drivers are, essentially, a shared library of privileged, memory resident, low level hardware handling routines. It is Linux's device drivers that handle the peculiarities of the devices they are managing.

There are many different device drivers in the Linux kernel (that is one of Linux's strengths) but they all share some common attributes:

Kernel code

Device drivers are part of the kernel and, like other code within the kernel, if they go wrong they can seriously damage the system. A badly written driver may even crash the system, possibly corrupting file systems and losing data.

Kernel interfaces

Device drivers must provide a standard interface to the Linux kernel or to the subsystem that they are part of. For example, the terminal driver provides a file I/O interface to the Linux kernel and a SCSI device driver provides a SCSI device interface to the SCSI subsystem which, in turn, provides both file I/O and buffer cache interfaces to the kernel.

Kernel mechanisms and services

Device drivers make use of standard kernel services such as memory allocation, interrupt delivery and wait queues to operate.

Loadable

Most of the Linux device drivers can be loaded on demand as kernel modules when they are needed and unloaded when they are no longer being used. This makes the kernel very adaptable and efficient with the system's resources.

Configurable

Linux device drivers can be built into the kernel. Which devices are built is configurable when the kernel is compiled.

Dynamic

As the system boots and each device driver is initialized it looks for the hardware devices that it is controlling. It does not matter if the device being controlled by a particular device driver does not exist. In this case the device driver is simply redundant and causes no harm apart from occupying a little of the system's memory.

Direct Memory Access (DMA)

Using interrupts driven device drivers to transfer data to or from hardware devices works well when the amount of data is reasonably low. For example a 9600 baud modem can transfer approximately one character every millisecond (1/1000 'th second). If the interrupt latency, the amount of time that it takes between the hardware device raising the interrupt and the device driver's interrupt handling routine being called, is low (say 2 milliseconds) then the overall system impact of the data transfer is very low. The 9600 baud modem data transfer would only take 0.002% of the CPU's processing time. For high speed devices, such as hard disk controllers or ethernet devices the data transfer rate is a lot higher. A SCSI device can transfer up to 40 Mbytes of information per second.

Direct Memory Access, or DMA, was invented to solve this problem. A DMA controller allows devices to transfer data to or from the system's memory without the intervention of the processor. A PC's ISA DMA controller has 8 DMA channels of which 7 are available for use by the device drivers. Each DMA channel has associated with it a 16 bit address register and a 16 bit count register. To initiate a data transfer the device driver sets up the DMA channel's address and count registers together with the direction of the data transfer, read or write. It then tells the device that it may start the DMA when it wishes. When the transfer is complete the device interrupts the PC. Whilst the transfer is taking place the CPU is free to do other things.

Device drivers have to be careful when using DMA. First of all the DMA controller knows nothing of virtual memory, it only has access to the physical memory in the system. Therefore the memory that is being DMA'd to or from must be a contiguous block of physical memory. This means that you cannot DMA directly into the virtual address space of a process. You can however lock the process's physical pages into memory, preventing them from being swapped out to the swap device during a DMA operation. Secondly, the DMA controller cannot access the whole of physical memory. The DMA channel's address register represents the first 16 bits of the DMA address, the next 8 bits come from the page register. This means that DMA requests are limited to the bottom 16 Mbytes of memory.

DMA channels are scarce resources, there are only 7 of them, and they cannot be shared between device drivers. Just like interrupts, the device driver must be able to work out which DMA channel it should use. Like interrupts, some devices have a fixed DMA channel. The floppy device, for example, always uses DMA channel 2. Sometimes the DMA channel for a device can be set by jumpers; a number of ethernet devices use this technique. The more flexible devices can be told (via their CSRs) which DMA channels to use and, in this case, the device driver can simply pick a free DMA channel to use.

Linux tracks the usage of the DMA channels using a vector of dma_chan data structures (one per DMA channel). The dma_chan data structure contains just two fields, a pointer to a string describing the owner of the DMA channel and a flag indicating if the DMA channel is allocated or not. It is this vector of dma_chan data structures that is printed when youcat`/proc/dma`.

Interfacing Device Drivers with the Kernel

The Linux kernel must be able to interact with them in standard ways. Each class of device driver, character, block and network, provides common interfaces that the kernel uses when requesting services from them. These common interfaces mean that the kernel can treat often very different devices and their device drivers absolutely the same. For example, SCSI and IDE disks behave very differently but the Linux kernel uses the same interface to both of them.

Linux is very dynamic, every time a Linux kernel boots it may encounter different physical devices and thus need different device drivers. Linux allows you to include device drivers at kernel build time via its configuration scripts. When these drivers are initialized at boot time they may not discover any hardware to control. Other drivers can be loaded as kernel modules when they are needed. To cope with this dynamic nature of device drivers, device drivers register themselves with the kernel as they are initialized. Linux maintains tables of registered device drivers as part of its interfaces with them. These tables include pointers to routines and information that support the interface with that class of devices.

Character Devices

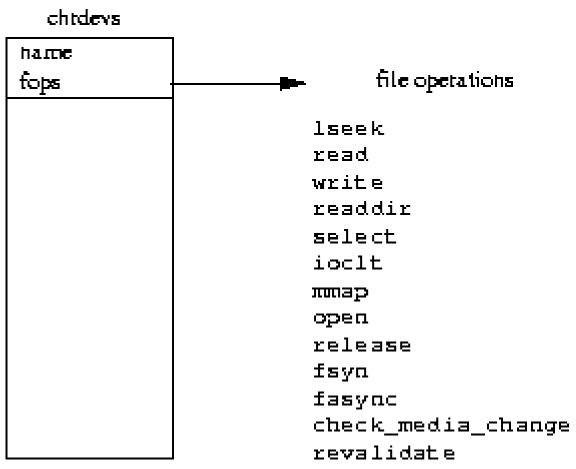


Figure 8.1: Character Devices

Character devices, the simplest of Linux's devices, are accessed as files, applications use standard system calls to open them, read from them, write to them and close them exactly as if the device were a file. This is true even if the device is a modem being used by the PPP daemon to connect a Linux system onto a network. As a character device is initialized its device driver registers itself with the Linux kernel by adding an entry into the chrdevs vector of device_struct data structures. The device's major device identifier (for example 4 for the tty device) is used as an index into this vector. The major device identifier for a device is fixed.

Block Devices

Block devices also support being accessed like files. The mechanisms used to provide the correct set of file operations for the opened block special file are very much the same as for character devices. Linux maintains the set of registered block devices as the blkdevs vector. It, like the chrdevs vector, is indexed using the device's major device number. Its entries are also device_struct data structures. Unlike character devices, there are classes of block devices. SCSI devices are one such class and IDE devices are another. It is the class that registers itself with the Linux kernel and provides file operations to the kernel. The device drivers for a class of block device provide class specific interfaces to the class. So, for example, a SCSI device driver has to provide interfaces to the SCSI subsystem which the SCSI subsystem uses to provide file operations for this device to the kernel.

Every block device driver must provide an interface to the buffer cache as well as the normal file operations interface. Each block device driver fills in its entry in the blk_dev vector

of blk_dev_struct data structures . The index into this vector is, again, the device's major number. The blk_dev_struct data structure consists of the address of a request routine and a pointer to a list of request data structures, each one representing a request from the buffer cache for the driver to read or write a block of data.

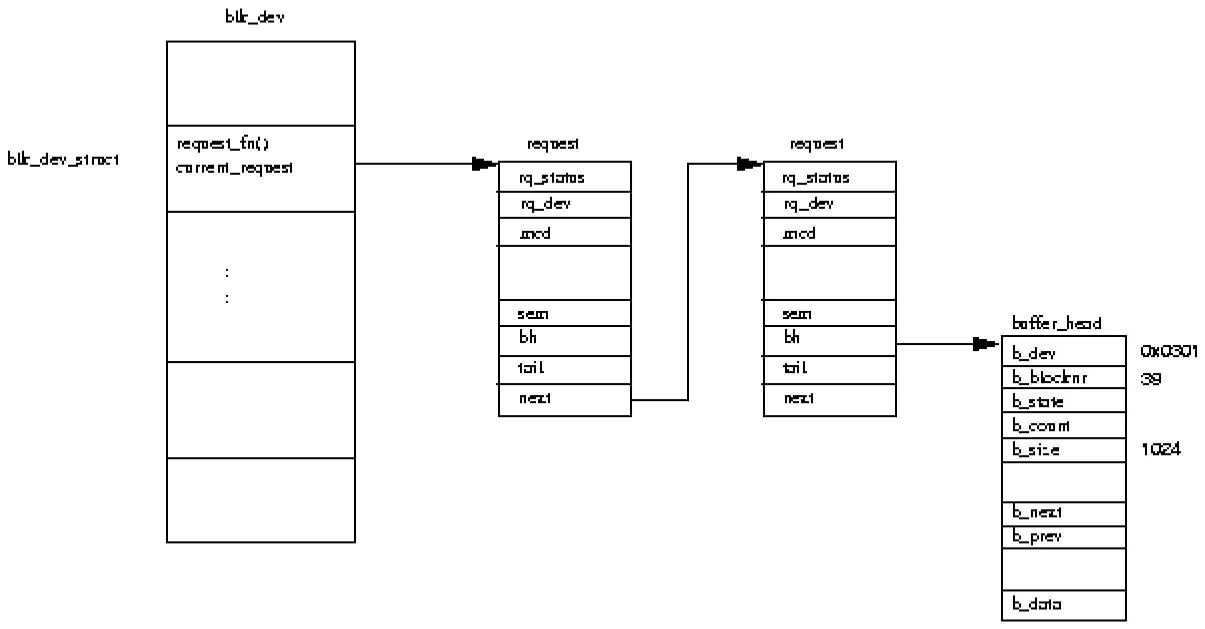


Figure 8.2: Buffer Cache Block Device Requests

Hard Disks

Hard disks can be further subdivided into *partitions*. A partition is a large group of sectors allocated for a particular purpose. Partitioning a disk allows the disk to be used by several operating system or for several purposes. A lot of Linux systems have a single disk with three partitions; one containing a DOS filesystem, another an EXT2 filesystem and a third for the swap partition. The partitions of a hard disk are described by a partition table; each entry describing where the partition starts and ends in terms of heads, sectors and cylinder numbers. For DOS formatted disks, those formatted by fdisk, there are four primary disk partitions. Not all four entries in the partition table have to be used. There are three types of partition supported by fdisk, primary, extended and logical. Extended partitions are not real partitions at all, they contain any number of logical paritions. Extended and logical partitions were invented as a way around the limit of four primary partitions. The following is the output from fdisk for a disk containing two primary partitions:

```
Disk /dev/sda: 64 heads, 32 sectors, 510 cylinders
Units = cylinders of 2048 * 512 bytes

   Device Boot   Begin    Start      End   Blocks   Id  System
/dev/sda1           1        1      478   489456   83  Linux native
/dev/sda2         479      479      510    32768   82  Linux swap

Expert command (m for help): p

Disk /dev/sda: 64 heads, 32 sectors, 510 cylinders

Nr AF  Hd Sec  Cyl  Hd Sec  Cyl   Start    Size ID
 1 00   1   1    0  63  32  477      32  978912 83
 2 00   0   1  478  63  32  509  978944   65536 82
 3 00   0   0    0   0   0    0       0       0 00
 4 00   0   0    0   0   0    0       0       0 00
```

IDE Disks

The most common disks used in Linux systems today are Integrated Disk Electronic or IDE disks. IDE is a disk interface rather than an I/O bus like SCSI. Each IDE controller can support up to two disks, one the master disk and the other the slave disk. The master and slave functions are usually set by jumpers on the disk. The first IDE controller in the system is known as the primary IDE controller, the next the secondary controller and so on. IDE can manage about 3.3 Mbytes per second of data transfer to or from the disk and the maximum IDE disk size is 538Mbytes. Extended IDE, or EIDE, has raised the disk size to a maximum of 8.6 Gbytes and the data transfer rate up to 16.6 Mbytes per second. IDE and EIDE disks are cheaper than SCSI disks and most modern PCs contain one or more on board IDE controllers.

Initializing the IDE Subsystem

IDE disks have been around for much of the IBM PC's history. Throughout this time the interface to these devices has changed. This makes the initialization of the IDE subsystem more complex than it might at first appear.

The maximum number of IDE controllers that Linux can support is 4. Each controller is represented by an ide_hwif_t data structure in the ide_hwifs vector. Each ide_hwif_t data structure contains two ide_drive_t data structures, one per possible supported master and slave IDE drive. During the initializing of the IDE subsystem, Linux first looks to see if there is information about the disks present in the system's CMOS memory. This is battery backed memory that does not lose its contents when the PC is powered off. This CMOS memory is actually in the system's real time clock device which always runs no matter if your PC is on or off. The CMOS memory locations are set up by the system's BIOS and tell Linux what IDE controllers and drives have been found. Linux retrieves the found disk's geometry from BIOS and uses the information to set up the ide_hwif_t data structure for this drive. More modern PCs use PCI chipsets such as Intel's 82430 VX chipset which includes a PCI EIDE controller. The IDE subsystem uses PCI BIOS callbacks to locate the PCI (E)IDE controllers in the system. It then calls PCI specific interrogation routines for those chipsets that are present.

SCSI Disks

The SCSI (Small Computer System Interface) bus is an efficient peer-to-peer data bus that supports up to eight devices per bus, including one or more hosts. Each device has to have a unique identifier and this is usually set by jumpers on the disks. Data can be transfered synchronously or asynchronously between any two devices on the bus and with 32 bit wide data transfers up to 40 Mbytes per second are possible. The SCSI bus transfers both data and state information between devices, and a single transaction between an *initiator* and a *target* can involve up to eight distinct phases. You can tell the current phase of a SCSI bus from five signals from the bus. The eight phases are:

BUS FREE
No device has control of the bus and there are no transactions currently happening,

ARBITRATION
A SCSI device has attempted to get control of the SCSI bus, it does this by asserting its SCSI identifer onto the address pins. The highest number SCSI identifier wins.

SELECTION
When a device has succeeded in getting control of the SCSI bus through arbitration it must now signal the target of this SCSI request that it wants to send a command to it. It does this by asserting the SCSI identifier of the target on the address pins.

RESELECTION
SCSI devices may disconnect during the processing of a request. The target may then reselect the initiator. Not all SCSI devices support this phase.

COMMAND
6,10 or 12 bytes of command can be transfered from the initiator to the target,

DATA IN, DATA OUT
During these phases data is transfered between the initiator and the target,

STATUS

This phase is entered after completion of all commands and allows the target to send a status byte indicating success or failure to the initiator,

MESSAGE IN, MESSAGE OUT

Additional information is transfered between the initiator and the target.

The Linux SCSI subsystem is made up of two basic elements, each of which is represented by data structures:

Host

A SCSI host is a physical piece of hardware, a SCSI controller. The NCR810 PCI SCSI controller is an example of a SCSI host. If a Linux system has more than one SCSI controller of the same type, each instance will be represented by a separate SCSI host. This means that a SCSI device driver may control more than one instance of its controller. SCSI hosts are almost always the *initiator* of SCSI commands.

Device

The most common set of SCSI device is a SCSI disk but the SCSI standard supports several more types; tape, CD-ROM and also a generic SCSI device. SCSI devices are almost always the *targets* of SCSI commands. These devices must be treated differently, for example with removable media such as CD-ROMs or tapes, Linux needs to detect if the media was removed. The different disk types have different major device numbers, allowing Linux to direct block device requests to the appropriate SCSI type.

Initializing the SCSI Subsystem

Initializing the SCSI subsystem is quite complex, reflecting the dynamic nature of SCSI buses and their devices. Linux initializes the SCSI subsystem at boot time; it finds the SCSI controllers (known as SCSI hosts) in the system and then probes each of their SCSI buses finding all of their devices. It then initializes those devices and makes them available to the rest of the Linux kernel via the normal file and buffer cache block device operations. This initialization is done in four phases:

First, Linux finds out which of the SCSI host adapters, or controllers, that were built into the kernel at kernel build time have hardware to control. Each built in SCSI host has a Scsi_Host_Template entry in the builtin_scsi_hosts vector The Scsi_Host_Template data structure contains pointers to routines that carry out SCSI host specific actions such as detecting what SCSI devices are attached to this SCSI host. These routines are called by the SCSI subsystem as it configures itself and they are part of the SCSI device driver supporting this host type. Each detected SCSI host, those for which there are real SCSI devices attached, has its Scsi_Host_Template data structure added to the scsi_hosts list of active SCSI hosts. Each instance of a detected host type is represented by a Scsi_Host data structure held in the scsi_hostlist list. For example a system with two NCR810 PCI SCSI controllers would have two Scsi_Host entries in the list, one per controller. Each Scsi_Host points at the Scsi_Host_Template representing its device driver.

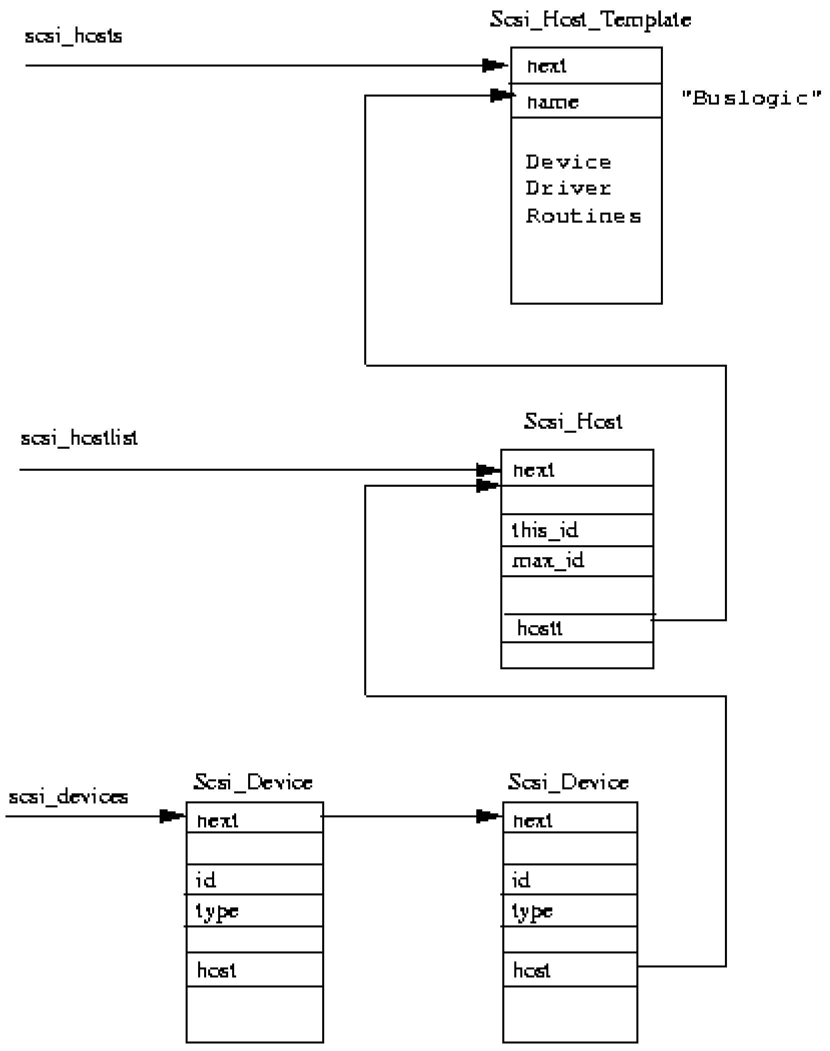


Figure 8.4: SCSI Data Structures

Now that every SCSI host has been discovered, the SCSI subsystem must find out what SCSI devices are attached to each host's bus. SCSI devices are numbered between 0 and 7 inclusively, each device's number or SCSI identifier being unique on the SCSI bus to which it is attached. SCSI identifiers are usually set by jumpers on the device. The SCSI initialization code finds each SCSI device on a SCSI bus by sending it a TEST_UNIT_READY command. When a device responds, its identification is read by sending it an ENQUIRY command. This gives Linux the vendor's name and the device's model and revision names. SCSI commands are represented by a Scsi_Cmnd data structure and these are passed to the device driver for this SCSI host by calling the device driver routines within its Scsi_Host_Template data structure. Every SCSI device that is found is represented by a Scsi_Device data structure, each of which points to its parent Scsi_Host. All of the Scsi_Device data structures are added to the scsi_devices list. Figure 8.4 shows how the main data structures relate to one another.

Network Devices

A network device is, so far as Linux's network subsystem is concerned, an entity that sends and receives packets of data. This is normally a physical device such as an ethernet card. Some network devices though are software only such as the loopback device which is used for sending data to yourself. Each network device is represented by a device data structure. Network device drivers register the devices that they control with Linux during network initialization at kernel boot time. The device data structure contains information about the device and the addresses of functions that allow the various supported network protocols to use the device's services. These functions are mostly concerned with transmitting data using the network device. The device uses standard networking support mechanisms to pass received data up to the appropriate protocol layer. All network data (packets) transmitted and received are represented by sk_buff data structures, these are flexible data structures that allow network protocol headers to be easily added and removed. How the network protocol layers use the network devices, how they pass data back and forth using sk_buff data structures is described in detail in the Networks chapter (Chapter networks-chapter). This chapter concentrates on the device data structure and on how network devices are discovered and initialized.

The device data structure contains information about the network device:

Name

Unlike block and character devices which have their device special files created using the mknod command, network device special files appear spontaniously as the system's network devices are discovered and initialized. Their names are standard, each name representing the type of device that it is. Multiple devices of the same type are numbered upwards from 0. Thus the ethernet devices are known as /dev/eth0,/dev/eth1,/dev/eth2 and so on. Some common network devices are:

/dev/ethN	Ethernet devices
/dev/slN	SLIP devices
/dev/pppN	PPP devices
/dev/lo	Loopback devices

Bus Information

This is information that the device driver needs in order to control the device. The *irq* number is the interrupt that this device is using. The *base address* is the address of any of the device's control and status registers in I/O memory. The *DMA channel* is the DMA channel number that this network device is using. All of this information is set at boot time as the device is initialized.

Interface Flags

These describe the characteristics and abilities of the network device:

IFF_UP	Interface is up and running,
IFF_BROADCAST	Broadcast address in device is valid
IFF_DEBUG	Device debugging turned on
IFF_LOOPBACK	This is a loopback device
IFF_POINTTOPOINT	This is point to point link (SLIP and PPP)
IFF_NOTRAILERS	No network trailers
IFF_RUNNING	Resources allocated

IFF_NOARP	Does not support ARP protocol
IFF_PROMISC	Device in promiscuous receive mode, it will receive all packets no matter who they are addressed to
IFF_ALLMULTI	Receive all IP multicast frames
IFF_MULTICAST	Can receive IP multicast frames

Protocol Information

Each device describes how it may be used by the network protocool layers:

mtu

The size of the largest packet that this network can transmit not including any link layer headers that it needs to add. This maximum is used by the protocol layers, for example IP, to select suitable packet sizes to send.

Family

The family indicates the protocol family that the device can support. The family for all Linux network devices is AF_INET, the Internet address family.

Type

The hardware interface type describes the media that this network device is attached to. There are many different types of media that Linux network devices support. These include Ethernet, X.25, Token Ring, Slip, PPP and Apple Localtalk.

Addresses

The device data structure holds a number of addresses that are relevent to this network device, including its IP addresses.

Packet Queue

This is the queue of sk_buff packets queued waiting to be transmitted on this network device,

Support Functions

Each device provides a standard set of routines that protocol layers call as part of their interface to this device's link layer. These include setup and frame transmit routines as well as routines to add standard frame headers and collect statistics. These statistics can be seen using the ifconfig command.

Initializing Network Devices

Network device drivers can, like other Linux device drivers, be built into the Linux kernel. Each potential network device is represented by a device data structure within the network device list pointed at by dev_base list pointer. The network layers call one of a number of network device service routines whose addresses are held in the device data structure if they need device specific work performing. Initially though, each device data structure holds only the address of an initialization or probe routine.

There are two problems to be solved for network device drivers. Firstly, not all of the network device drivers built into the Linux kernel will have devices to control. Secondly, the ethernet devices in the system are always called /dev/eth0, /dev/eth1 and so on, no matter what their underlying device drivers are. The problem of ``missing'' network devices is easily solved. As the initialization routine for each network device is called, it returns a status indicating whether or not it located an instance of the controller that it is driving. If the driver could not find any devices, its entry in the device list pointed at by dev_base is removed. If the driver could find a device it fills out the rest of the device data structure with information about the device and the addresses of the support functions within the network device driver.

Questions:

1 Define device drivers. Describes it’s different types.
2 Explain how an interface is created between a Kernel and a device driver in case of character device drivers.
3 Case study for device drivers used for external storage devices

Assignment No:10

Problem Statement: Demonstrate use of device drivers.

To install hardware devices on Windows servers, admins must install the appropriate device drivers first. Here are some guidelines for installing, configuring and troubleshooting device drivers.

System administrators frequently have to install hardware devices such as disk controllers and network cards on their Windows servers. To do so, they must install the appropriate device drivers on the server first.

Here are some guidelines for installing, configuring and troubleshooting device drivers, which provide the necessary interface between the application programs and the actual hardware, on Windows 2000 server system.

Methods of installing device drivers

Admins can install device drivers on a server in four ways.

1. **Running the Setup.** Device drivers are automatically installed for all detected devices during installation of Windows 2000 system on the server computer
2. **Starting the computer.** Whenever the server is started, new devices are detected and their device drivers are automatically installed
3. **Scanning for new hardware.**Add/Remove Hardware Wizard can be used to perform the hardware-detection process and desired device driver installation
4. **Manual installation.** You can use the Add/Remove Hardware Wizard to specify the device you want to install, or you can right-click the .INF file that comes with the driver and choose Install.

Note: You must have administrative privileges on the server to install device drivers

Configuring devices using Device Manager

Device Manager displays information about all devices installed on your server. It displays a list of devices that were either detected or for which drivers are installed. The icon for the device indicates whether the device is in proper working condition.

Normal icon indicates that the device driver has initialized successfully while an Exclamation point on icon would mean that device is incorrectly configured and a Stop sign on icon means that device could not be initialized due to hardware conflicts.

To resolve hardware conflicts and to override the resources or the device driver that were assigned to a device, follow these steps:

1. Start Device manager by typing 'Devmgmt.msc' in the Run dialog box.
2. Right-click the desired device and select Properties.
3. Click the Resources tab on the Properties dialog box.
4. Click the resource to be changed and then clear the "Use automatic settings" checkbox.

5. Click Change Setting and then change the value of the setting to the desired value.

However best thing to do is to let Plug and Play resolve conflicts whenever possible.

Changing the driver source location

While adding a device to your system, Windows 2000 provides a driver, which gets installed from the Driver.cab file in the systemroot\DriverCache\I386 folder. Tthis file is approximately 50 MB. If the driver cannot be found in Driver.cab, Windows 2000 will prompt you for the location of the driver.

It's a good idea to keep this file at a central location so that all clients install drivers from a consistent central source. To change the source path in the Registry and delete the local copy of Driver.cab, simply update the data for the "SourcePath" value in his key in the Registry: HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\Windows\CurrentVersion\Setup
This will also save space on the hard disk of client computers.

Updating and removing device drivers

Administrators often have to update the existing device drivers or remove them. To update a device driver:

1. Start Device Manager by typing 'Devmgmt.msc' in the Run dialog box.
2. Right-click the desired device and select Properties.
3. Click the 'Driver' tab and then click the Update Driver button.

To remove a device:

1. Start Device Manager by typing 'Devmgmt.msc' in the Run dialog box.
2. Right-click the desired device and select Properties.
3. Click the 'Driver' tab and then click Uninstall button

Note: Removing the device does not actually delete the device driver itself. What it does is remove references to the driver from the Registry so that the computer does not load the driver.

Upgrading from uniprocessor to multiprocessor

Many times the need arises to get your server upgraded in terms of software and hardware. If Windows 2000 was originally installed on a single processor PC and a second processor is to be added, you need to update the system software. To do so::

1. Start Device manager by typing 'Devmgmt.msc' in the Run dialog box.
2. Expand the Computer icon and make note of the computer model.
3. Right-click the computer model below the Computer icon and select Properties.
4. Begin the Upgrade Device Driver Wizard by clicking on the Update Driver button from the Driver tab.
5. Choose Display a List of Known Drivers For This Device and then click Next.

By selecting Show All Hardware of This Device Class, you can choose from a list of supplied multiprocessor drivers. If you have W2k-specific multiprocessor drivers, use the Have Disk button.

Driver signing

Driver signing ensures that the drivers have been certified by Windows Hardware Quality Labs (WHQL). It has been implemented in Windows 2000 to improve the quality of drivers and increase the overall stability of the Windows operating system.

Microsoft digitally signs all files and drivers on the Windows 2000 installation CD. But administrators often download drivers from the Internet or get them from various hardware vendors. To configure how these third-party drivers should be handled:

1. Open Control Panel.
2. Select the System Properties icon.
3. Switch to Hardware tab, and click the Driver Signing button.

A dialog box opens which has three checkboxes:

- Ignore (allows installing all files, regardless of the file signature)
- Warn (displays a message to the user before installing an unsigned file). This is usually the default setting.
- Block (restricts the installation of unsigned files).

Check the required checkbox and hit OK.

Disabling devices and services for a specific profile

Individual devices and services may be configured to load or not load as part of a Hardware Profile. To disable devices:

- Open Device Manager and select Properties of the desired device
- On the General tab under 'Device Usage' label select Do not use this device (disable) option
- Click OK.

To disable services:

- From Administrative Tools, select Services.
- Select the Log On tab from the Properties dialog of any of the Services
- Click on Enable or Disable for the selected Hardware Profile.

Driver installations: failure and recovery

Here are three problems that commonly occur during the installation of a device driver and their solutions:

Problem: Wrong driver is installed

Solution: Press F8 at startup and use the Last Known Good Configuration

Problem: Driver gets installed partially or some files are missed during installation.

Solution: Use a command-line utility called "System File Checker" that can verify the version of protected system files and revert to a previous version. Type sfc.exe /scannow on the command prompt for running the utility.

Problem: Files are corrupt during installation.

Solution: Uninstall or remove the device driver, restart the server and reinstall the driver.

Questions:
1 Describe the steps involved in installation of a scanner
2Case studydevice drives