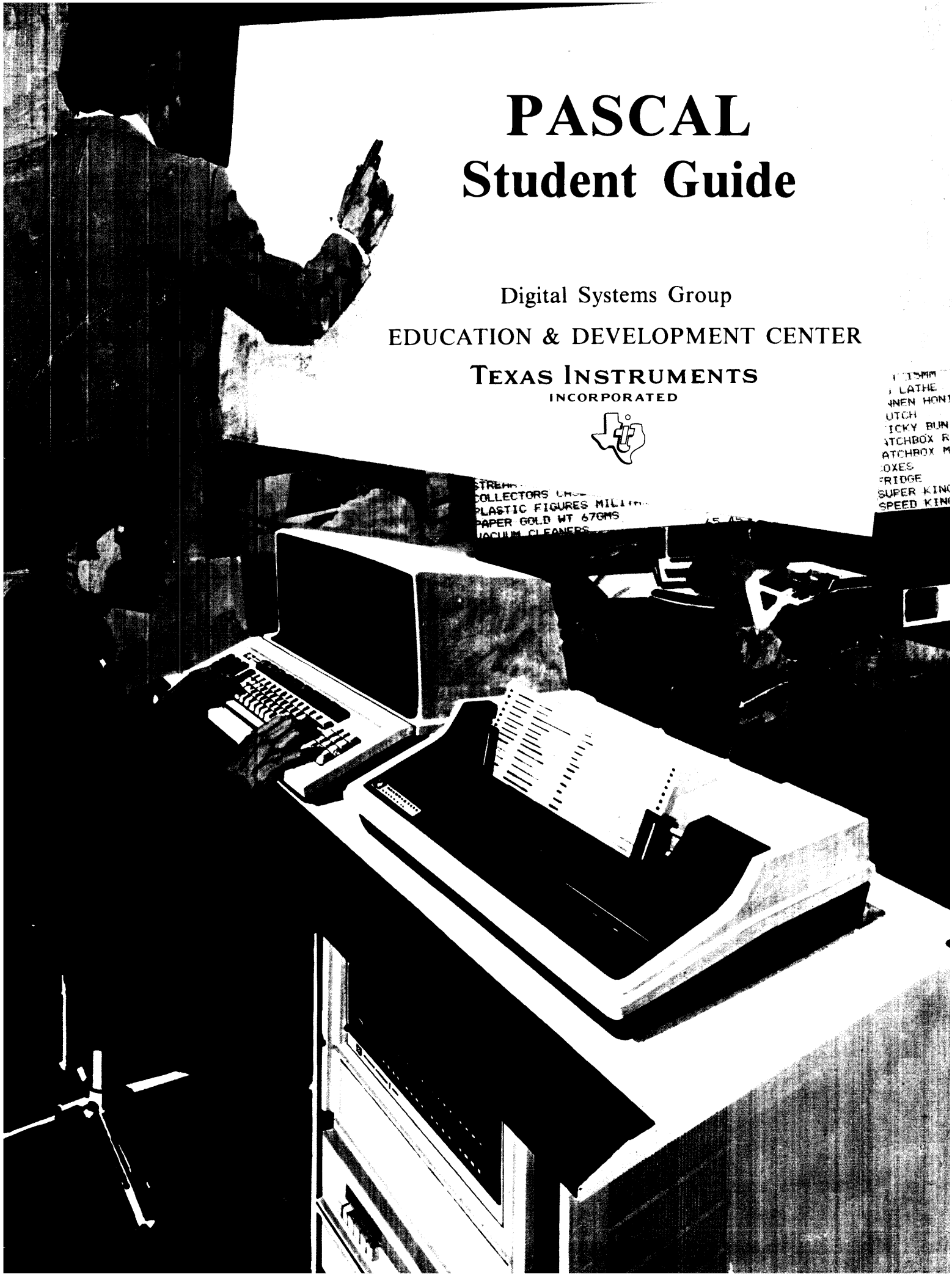
PASCAL
Student Guide



Digital Systems Group

EDUCATION & DEVELOPMENT CENTER

TEXAS INSTRUMENTS
INCORPORATED

PASCAL COURSE AGENDA

MONDAY

1. DX10 OVERVIEW
   - 990 ARCHITECTURE OVERVIEW
   - S/W SYSTEM OVERVIEW
   - SYSTEM COMMAND INTERPRETER (SCI)

2. PASCAL OVERVIEW

3. PASCAL DECLARATIONS
   - VARIABLE DECLARATIONS
   - TYPE DECLARATIONS
   - CONSTANT DECLARATIONS

LUNCH

4. SIMPLE PROGRAM CONTROL STRUCTURES
   - IF-THEN STATEMENT
   - IF-THEN-ELSE STATEMENT
   - COMPOUND STATEMENT
   - WHILE LOOP
   - FOR LOOP

5. SIMPLE I/O
   - TYPE CHAR
   - READ STATEMENT (UNFORMATTED)
   - WRITE STATEMENT(UNFORMATTED)
   - WRITELN STATEMENT

6. SIMPLE PROCEDURES AND FUNCTIONS
   - DEFINITION
   - PARAMETERS

7. ASSIGNMENT 1 (HOMEWORK)
   - COMPILER OPTIONS AND RUNTIME CHECKS
   - READING A MEMORY DUMP

TUESDAY

1. DX10 SYSTEM USAGE
   - BOOTING DX10
   - SOFTWARE MAINTENANCE SCI COMMANDS
   - FILE MANAGEMENT STRUCTURE
   - SYNONYMS

2. PASCAL DEVELOPMENT UNDER DX10
   - PASCAL COMPILER
   - LINK.EDITOR
   - INSTALLATION AND EXECUTION
   - TEXT EDITOR

LUNCH

3. LABORATORY
   - ASSIGNMENT #1

WEDNESDAY

1. PASCAL DATA TYPES
   - BOOLEAN
   - ARRAYS
   - RECORDS

2. PROGRAM CONTROL STATEMENTS
   - REPEAT-UNTIL LOOP
   - CASE STATEMENT
   - WITH STATEMENT

LUNCH

3. FORMATTED I/O

4. NESTED PROCEDURES
   - DEFINITION
   - PARAMETER PASSING
   - SCOPE OF VARIABLES
   - ROUTINE ACCESSIBILITY
   - TOP DOWN DESIGN

5. STACK MEMORY ALLOCATION

6. ASSIGNMENT #2 (HOMEWORK)

THURSDAY

1. LABORATORY
   - ASSIGNMENT #2

LUNCH

2. POINTERS

3. SETS AND TYPE TRANSFER

4. SEQUENTIAL AND RELATIVE RECORD FILES

5. ASSIGNMENT #3 (HOMEWORK)

FRIDAY

1. LABORATORY
   - ASSIGNMENT #3

LUNCH

2. DECLARING EXTERNAL PROCEDURES

3. DIRECT DX10 INTERFACE

4. REENTRANT PASCAL

5. BATCH SCI FOR PASCAL

3

DS990 SYSTEM OVERVIEW

OBJECTIVE

TO AQUAINT THE STUDENT WITH THE OVERALL CHARACTERISTICS OF THE DX10 DEVELOPMENT SYSTEM.

AGENDA

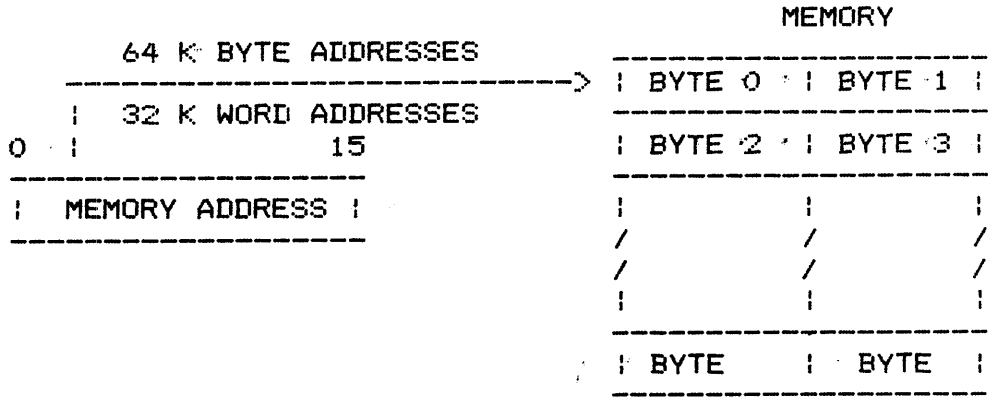
1. HARDWARE CHARACTERISTICS OF THE DS990
   - MEMORY ORGANIZATION
   - CPU CHARACTERISTICS
   - REGISTER ORGANIZATION
   - I/O PORTS
2. PERIPHERALS
   - CRU PERIPHERALS
   - TILINE PERIPHERALS
3. DX10
   - FEATURES
   - MEMORY UTILIZATION
   - MULTI-TASKING PRIORITIES
   - SYSTEM COMMAND INTERPRETER (SCI)
   - PROGRAM DEVELOPMENT UTILITIES
   - HIGH LEVEL LANGUAGES

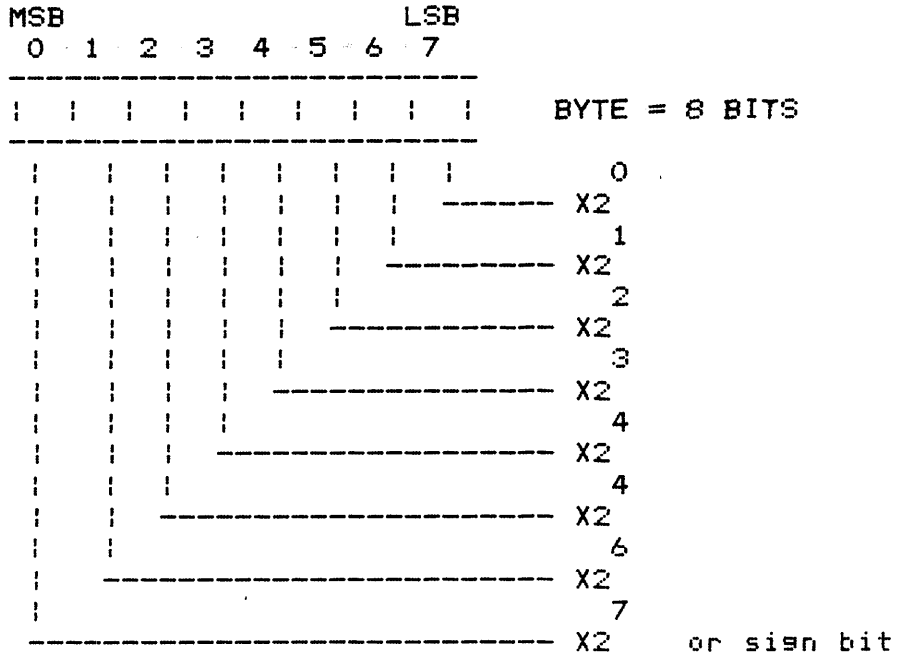
990 SYSTEM OVERVIEW

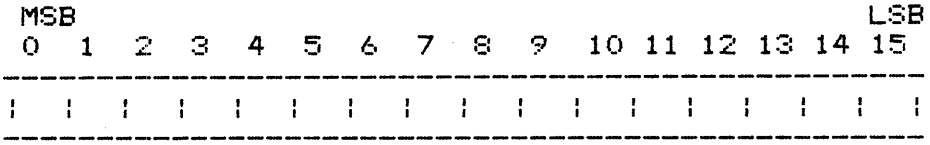
990/10 ARCHITECTURE

MEMORY

- 16 BIT MEMORY WORDS
- FULL BYTE ADDRESSING CAPABILITY
- TWO'S COMPLEMENT REPRESENTATION OF INTEGERS







MINIMUM MEMORY - 32K WORDS
MAXIMUM MEMORY - 1 MILLION WORDS

5

CPU - CENTRAL PROCESSING UNIT

- 2 BOARDS OF TTL LOGIC
- 72 INSTRUCTIONS

INSTRUCTION TYPE	#
ARITHMETIC	13
BRANCH	18
COMPARE	5
CONTROL/CRU	10
LOAD/MOVE	8
LOGICAL	10
SHIFT	4
XOP	1
MAPPING	3
TOTAL	72

REGISTERS

- 0 PROGRAM COUNTER
- 0 STATUS REGISTER
- 0 WORKSPACE

** NOTE: THERE ARE NO H/W ARITHMETIC REGISTERS

HARDWARE REGISTERS

PROGRAM COUNTER

- WORD ADDRESSES
- CONTROL PROGRAM EXECUTION SEQUENCE
- TELLS PROCESSOR WHERE TO FETCH NEXT INSTRUCTION
- AUTOMATICALLY INCREMENTS BY 2 UNLESS MODIFIED
- BY BRANCH-TYPE INSTRUCTION

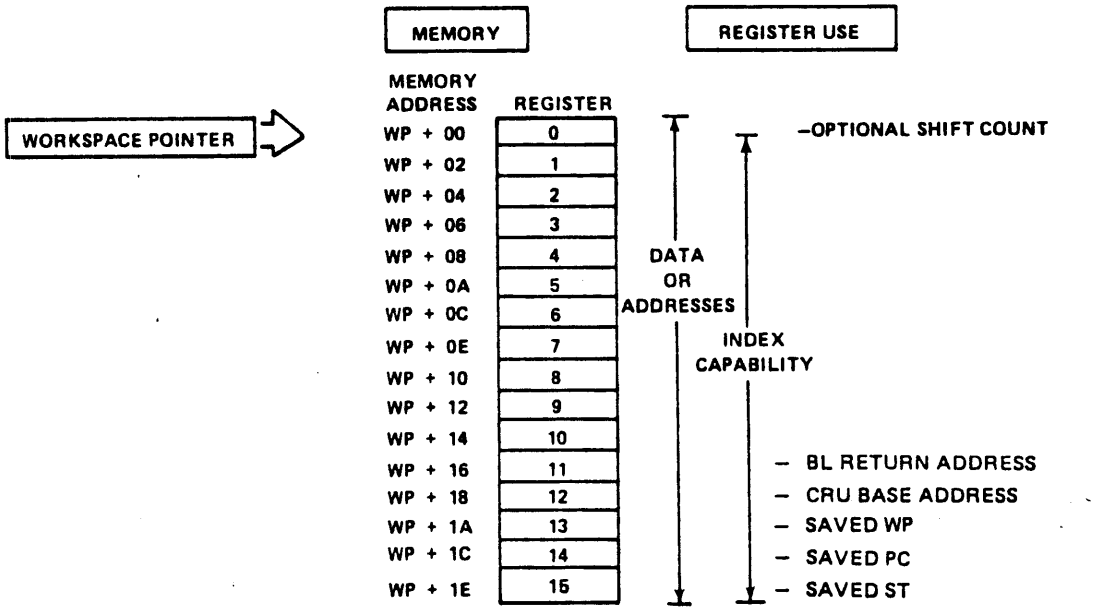
STATUS REGISTER

- CONSTANTLY INDICATES PRESENT STATUS OF CPU
- EX. COMPARE RESULTS
  ARITHMETIC RESULTS
  INTERRUPT MASK
  PARITY, XOP, MAPFILE, ETC.

WORKSPACE POINTER

- WORD ADDRESSES
- INDICATES BEGINNING ADDRESS OF REGISTER FILE
- REGISTER FILE IS ANY 16 SEQUENTIAL WORDS ANYWHERE IN MEMORY

WORKSPACE REGISTERS



INPUT/OUTPUT

COMMUNICATIONS REGISTER UNIT (CRU)

- 1 BIT SERIAL I/O BUS
- SUPPORTS SLOWER 990 PERIPHERAL DEVICES
  - CRT'S
  - LINE PRINTERS
  - FLOPPY DISKS
  - DATA TERMINALS
  - ETC.

TILINE

- 16 BIT PARALLEL I/O BUS
- ASYNCHRONOUS COMMUNICATION FOR TRANSFERS
- SUPPORTS HIGH SPEED PERIPHERAL DEVICES
  - 'HARD' DISKS
  - MAGNETIC TAPE
  - MEMORY
  - ETC.

CRU DEVICES

CARD READER - 804

- 400 CARDS/MINUTE
- READS FRAYED AND DAMAGED CARDS

LINE PRINTER - 810

- IMPACT
- 96 ASCII CHARACTER SET
- 132 COLUMN PAPER - SIX PART MULTICOPY
- 150 CHAR/SEC
- BIDIRECTIONAL PRINTING

LINE PRINTER - 2230

- DRUM LINE PRINTER
- 300 LINES/MINUTE
- 64 ASCII CHARACTER SET
- 132 COLUMN PAPER

LINE PRINTER - 2260

- DRUM LINE PRINTER
- 600 LINES/MINUTE
- 64 ASCII CHARACTER SET
- 132 COLUMN PAPER

773 ASR/KSR

- 95 PRINTABLE CHARACTERS
- 30 CHAR/SECOND
- KEYBOARD, PRINTER, 2 CASSETTES
- CASSETTE SPEED 120 CPS TO XMISS LINE

9

CRU DEVICES (CONTINUED)

911 CRT

- 96 CHARACTER ASCII
- 12 PROGRAMMABLE FUNCTION KEYS
- 960 CHAR (12 X 80) OR 1920 CHAR (24 X 80) FULL SCREEN
- FULL CURSOR CONTROL, INCLUDING PROGRAMMABLE CURSOR INTENSITY

913A CRT

- 57 CHARACTERS + 32 CONTROL (14 FOR USER DEFINITION)
- 960 CHAR FULL SCREEN (12 X 80)
- LOCATED UP TO 2000' FROM CPU

FLOPPY DISK

- IBM COMPATIBLE FLEXIBLE DISKETTE (REMOVABLE)
- 4 DRIVES PER CONTROLLER
- 256K BYTES PER DISKETTE
- TRANSFER RATE 250K BITS PER SECOND (156 WORDS)
- ACCESS TIME 260 MS AVERAGE
- PROCESSOR CONTROLLED I/O

990 COMMUNICATIONS INTERFACE

- RS - 232C INTERFACE
- SELECTABLE BAUD RATES
- SELECTABLE CHARACTER SIZE, 5-9 BIDS WITH PROGRAMMABLE PARITY

16 I/O DATA MODULE - TTL

- 16 INPUT LINES - SINGLE LINE ADDRESSABLE
- 16 OUTPUT LINES - SINGLE LINE ADDRESSABLE
- USER END OF MODULE IS TTL COMPATIBLE

10

CRU DEVICES (CONTINUED)

16 I/O DATA MODULE - EIA

- 16 INPUT LINES - SINGLE LINE ADDRESSABLE
- 16 OUTPUT LINES - SINGLE LINE ADDRESSABLE
- USER END OF MODULE IS EIA COMPATIBLE

PROM PROGRAMMER

- PERMANENTLY CODES PROGRAMS INTO PROM'S ON COMMAND FROM PROM BURNING SOFTWARE
- PROGRAMS TTL PROMS AND EROMS
- USED TO CREATE CUSTOM LOADERS, CONTROL PROGRAMS, ETC., FOR 990/R CPU

TILINE DEVICES

MOVING HEAD DISK - DIABLO 31

- 4 DRIVES PER CONTROLLER
- REMOVABLE CARTRIDGE
- 1,15M WORDS PER CARTRIDGE
- TRANSFER RATE 96K WORDS PER SECOND
- ACCESS TIME 70MS AVERAGE
- AUTOMATIC TRANSFER

MOVING HEAD DISK - DS10

- 2 DRIVES PER CONTROLLER
- DUAL PLATTER, SINGLE ACCESS
- 9.4 MEGABYTES OF FORMATTED STORAGE
- TRANSFER RATE IS 312K BYTES PER SECOND

MOVING HEAD DISK - DS25

- 5 PLATTER REMOVABLE DISK PACK
- 4 DRIVES PER CONTROLLER
- 25 MEGABYTES PER DISK PACK
- AVERAGE TRANSFER RATE IS 403,000 BYTES PER SECOND

MOVING HEAD DISK - DS50

- 5 PLATTER REMOVABLE DISK PACK
- 4 DRIVES PER CONTROLLER
- 50 MEGABYTES PER DISK PACK
- AVERAGE TRANSFER RATE IS 403,000 BYTES PER SECOND

MAGNETIC TAPE DRIVE - MODEL 979A

- SUPPORT UP TO 4 TRANSPORTS PER CONTROLLER
- VACUUM COLUMNS, 37.5 IPS
- 9-TRACK 800 - BPI NRZI VERSION OR 1600 - BPI PE VERSION

DX10 OPERATING SYSTEM

FEATURES

- MULTI-TASKING
  - SUPPORTS UP TO 255 LOGICALLY CONCURRENT TASKS
  - PRIORITY ROLL IN/ROLL OUT DISCIPLINE
  - MULTILEVEL PRIORITY TIME SLICE DISCIPLINE
- MULTI-USER DEVELOPMENT
  - SUPPORTS UP TO 40 CONCURRENT DEVELOPMENT STATIONS
- COMPLETE FILE MANAGEMENT PACKAGE
  - USER DEFINED FILE DIRECTORY STRUCTURE
  - FILE TYPES SUPPORTED
    - SEQUENTIAL
    - RELATIVE RECORD
    - MULTIKEY INDEX
- GENERAL PURPOSE APPLICATIONS O.S.
  - PROCESS CONTROL
  - DATA PROCESSING
  - SCIENTIFIC APPLICATIONS
- GENERAL O.S./USER INTERFACE (SCI)
  - FLEXIBLE COMMAND LANGUAGE
  - NEW COMMANDS MAY BE CREATED
  - OLD COMMANDS MAY BE MODIFIED
- HIGH LEVEL LANGUAGE SUPPORT
  - COBOL
  - PASCAL
  - FORTRAN
  - BASIC
  - RPG II
- DX10 REQUIRED H/W
  - 990/10 WITH MEMORY MAPPING
  - 64K WORDS OF MEMORY
  - MOVING HEAD DISK(S)
  - VIDEO DISPLAY TERMINAL(S)
  - DISK BACKUP CAPABILITY

13

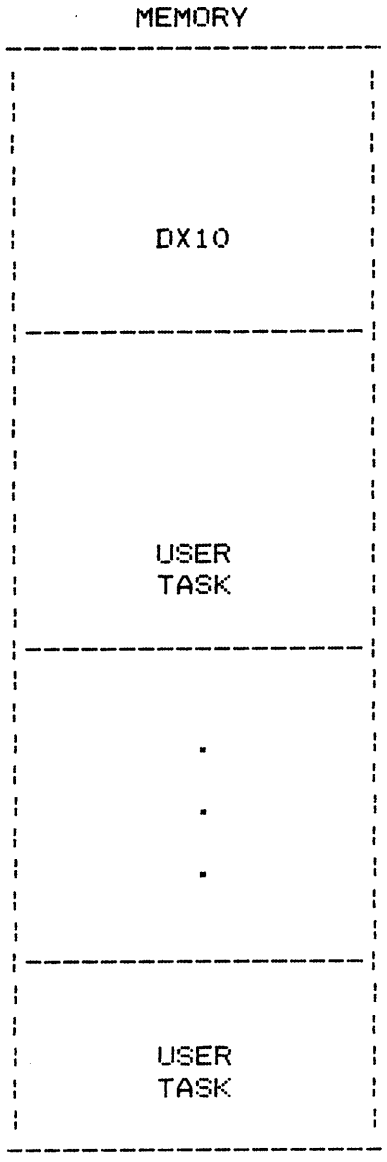
DX10 OPERATING SYSTEM (CONTINUED)

DX10 ORGANIZATION

DX10 RESIDES IN LOW MEMORY AND IS MEMORY RESIDENT

MANY USERS PROGRAMS (TASKS) MAY BE IN MEMORY AT THE SAME TIME

THE USER TASKS 'SHARE' CPU'S TIME AND I/O RESOURCES (MULTI-TASKING)

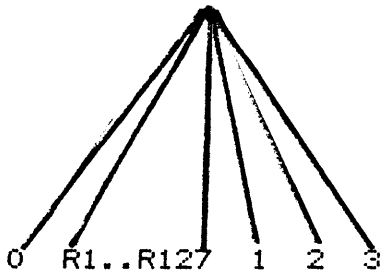


DX10 ALLOCATES 'TIME SLICES' TO ALL OR THE 'ACTIVE' (RUNNING) TASKS IN THE SYSTEM ON A PRIORITY BASIS.

DX10 3.2 TASK SCHEDULER

" THE TASK SCHEDULER ALLOCATES THE CPU TO THE HIGHEST PRIORITY TASK AWAITING EXECUTION. "

Priority Structure:



0 -> System

R1 - R127 -> Real-Time

1 -> Foreground Interactive

2 -> Foreground Compute-Bound

3 -> Background (Batch)

Four SYSGEN parameters determine how the scheduler works:

(1) TIME SLICING - YES/NO

If the time slicing option is selected, then CPU time for a given priority level will be allocated on a round robin basis among all the active tasks on that queue.

If time slicing is not chosen, then the first task on a queue will be allocated CPU time until it terminates, is suspended, or an external event causes a rescheduling.

(2) LENGTH OF TIME SLICE

a multiple of 50 msec intervals

(3) TASK SENTRY - YES/NO

if chosen during the system generation, the Task Sentry will lower the priority of a task by one when the task has had control of the CPU for a specified amount of time.

if the Task Sentry is not enabled, then a CPU-bound task may lock out all tasks of a lower priority level.

(4) SENTRY TIME

A multiple of 50 msec intervals.

INTERACTING FACTORS IN SCHEDULING

TASK SENTRY

TIME SLICE	YES	NO
YES	round robin on queue task bumped when sentry time up	round robin on queue when queue is exhausted next queue is called
NO	first task on queue gets CPU until sentry time up, then bumped	highest task hogs CPU as long as it wants

THE SCHEDULING MAY BE AFFECTED BY THESE EVENTS:

* PREEMPTIVE BIDDING - A higher priority task always gets the CPU when it becomes active

* The executing task suspends

* A time delayed task is due to be activated

* The priority of the executing trask is lowered by the Task Sentry (if task sentry is active in the system)

* A task completes a time slice (if the time slice option was included in the system)

SYSTEM COMMAND INTERPRETER (SCI)

PROVIDES

- USER INTERFACE TO DX10 INLCUDING:
  - LOG IN AND OUT.
  - TIME AND DATE SETUP AND INQUIRY.
  - DISK VOLUME INITIALIZATION, INSTALL, AND UNLOAD.
  - DISK DIRECTORY BACKUP, RESTORE, AND COPY.
  - CREATING AND DELETING DIRECTORIES AND FILES.
  - CHANGING FILE NAMES AND PROTECTION.
  - VIEWING AND LISTING DIRECTORIES AND FILES.
  - COPYING DIRECTORIES AND FILES.
  - LOGICAL UNIT ASSIGNMENT, POSITIONING, AND RELEASE.
  - SYSTEM I/O STATUS DISPLAY.
  - SYSTEM TASK STATUS DISPLAY.
  - PROGRAM ACTIVATION AND CONTROL.
  - BATCH COMMAND INPUT, ACTIVATION, AND STATUS.
  - STATION CONTROL (USER ID, TERMINAL STATUS, ETC.)
  - INSTALLING AND DELETING PROGRAMS.
  - ACTIVATION OF THE SYSTEM LOG.
  - PROGRAM DEBUGGING INCLUDING SUCH ITEMS AS:
    - BREAKPOINTS.
    - MEMORY/DISK DUMP OR DISPLAY.
    - DECIMAL/HEXADECIMAL ARITHMETIC AID.
    - INTERACTIVELY CONTROLLED PROGRAM TRACE.
  - TEXT EDIT CONTROL.
  - LINK EDIT ACTIVATION.
  - ASSEMBLER, COBOL, FORTRAN, PASCAL, RPG II, AND BASIC ACTIVATION.
  - SORT/MERGE ACTIVATION.

TWO MODES OF OPERATION

- CONVERSATIONAL
  INTERACTIVE COMMAND ENTRY AND RESPONSE

- BATCH
  BATCH COMMAND ENTRY FROM A SEQUENTIAL DEVICE OR FILE.

17

PROGRAM DEVELOPMENT

TEXT EDITOR

- CONTROLLED VIA SCI
- CHARACTER ORIENTED EDITING CAPABILITY
- FILES ARE AUTOMATICALLY CREATED WHEN CREATING A NEW PROGRAM
- ANY ASCII TEXT FILE MAY BE EDITED

LINK EDITOR

- CONTROLLED VIA SCI
- LINKS TOGETHER SUBROUTINES AND RUNTIME LIBRARY OBJECT (MACHINE) PROGRAMS FOR INSTALLATION AND EXECUTION UNDER DX10.
- PROVIDES FOR AUTO-INSTALLATION OF PROGRAMS WHEN LINKING IS COMPLETED.
- PROVIDES CAPABILITY TO CREATE AN 'OVERLAY' PROGRAM STRUCTURE.
- PROVIDES CAPABILITY TO CREATE A 'REENTRANT' PROGRAM STRUCTURE.

DX10 SUPPORTED HIGH LEVEL LANGUAGES

TI990 COBOL

- O BUSINESS ORIENTED DATA PROCESSING LANGUAGE
- O CONFORMS TO THE ANSI COBOL SUBSET WITH SELECTED FEATURES FROM LEVEL 2
- O INCLUDES SOME EXTENSIONS BEYOND THE LEVELS DEFINED BY ANSI
- O INCLUDES A STATEMENT LEVEL DEBUG PACKAGE FOR EASE IN DEBUGGING COBOL PROGRAMS
- O UTILIZES A REENTRANT RUNTIME PACKAGE FOR EXECUTION

TI990 FORTRAN

- O SCIENTIFIC PROGRAMMING LANGUAGE
- O ANSI STANDARD FORTRAN IV PLUS ALL RECOMMENDED ISA EXTENSIONS
- O OTHER EXTENTIONS INCLUDE:
  - VARIABLE NAMES OF ANY LENGTH
  - GENERAL INTEGER EXPRESSIONS AS SUBCRIPTS
  - VDT I/O HANDLING STATEMENTS
  - DIRECT CRU I/O CAPABILITY
  - MIXED MODE EXPRESSIONS
  - EXTENDED PRECISION INTEGERS
  - 16 BIT FIXED-POINT ARITHMETIC
  - IMPLICIT VARIABLE TYPING

TI990 RPG II

- O ORIENTED TOWARD BUSINESS PROGRAMMING AND REPORT GENERATION
- O INCLUDES A PRE-FORMATTING FORM EDITOR
- O COMPATIBLE WITH THE IBM SYSTEM/3 RPG II WHERE HARDWARE AND OPERATING SYSTEM CHARACTERISTICS PERMIT
- O ONE PASS COMPILER WILL EXECUTE IN ABOUT 10K WORDS OF MEMORY
- O COMPILER GENERATES ALPHABETIC SUMMARY LISTINGS OF ALL VARIABLES
- O PACKAGE INCLUDES A UNIQUE DEBUG PACKAGE

TI990 PASCAL

- O GENERAL PURPOSE PROGRAMMING LANGUAGE
- O BASED ON THE 'PASCAL USER MANUAL AND REPORT' BY KATHLEEN JENSEN AND NIKLAUS WIRTH
- O CAPABILITIES
  - PROVIDES SCIENTIFIC PROGRAMMING CAPABILITIES SIMILAR TO FORTRAN WITH SOME IMPORTANT IMPROVEMENTS (EXTENDED PRECISION REAL NUMBERS, GENERALIZED ARRAY INDEXING CAPABILITY, ETC)
  - 'BLOCK STRUCTURED' SIMILAR TO ALGOL AND PL/1
  - ACCESS TO ASSEMBLY LANGUAGE SUBROUTINES FOR PRIMITIVE LEVEL PROGRAMMING
  - ACCESS TO ROUTINES WRITTEN IN OTHER LANGUAGES
  - PROGRAM CONTROL STRUCTURES FIT THE CLASSICAL DEFINITIONS FOR 'STRUCTURED' PROBLEM SOLVING

TI990 BASIC

- O GENERAL PURPOSE INTERACTIVE PROBLEM SOLVING LANGUAGE
- O CONFORMS TO DARTMOUTH BASIC WITH CERTAIN EXTENSIONS
- O PACKAGE INCLUDES A MULTI-TERMINAL INTERFACE MODULE TO COORDINATE THE INTERACTIONS OF THE EXECUTIVE, THE BASIC INTERPRETER, AND THE OPERATING SYSTEM

PASCAL OVERVIEW

PASCAL IS A "BLOCK STRUCTURED LANGUAGE". IT WAS WRITTEN WITH THE FOLLOWING ADVANTAGES IN MIND:

* EASE OF DEFINING ALGORITHMS

* GOOD DATA STRUCTURING FACILITIES

* BIT MANIPULATION CAPABILITIES

* TO FACILITATE CONSTRUCTION OF RELIABLE AND TRANSPORTABLE CODE.

  - RESTRICTS FUNCTION SIDE EFFECTS

  - EXPLICIT TYPE CHECKING FOR VARIABLES

  - STRONG TYPE CHECKING FOR PARAMETERS

* EASE OF MODIFICATION

  - READABILITY

  - WELL THOUGHT OUT CONTROL STRUCTURES PROVIDE IMPROVED CAPABILITIES FOR ORGANIZING THE FLOW OF A PROGRAM

* LOOKS FORWARD TO THE DEVELOPMENT OF MORE SOPHISTICATED TECHNIQUES FOR VERIFYING THE CORRECTNESS OF PROGRAMS

USES

* SYSTEM PROGRAMMING

* SCIENTIFIC APPLICATIONS
  DYNAMIC BOUNDS FOR ARRAYS AND SETS
  MULTIPRECISION REAL VARIABLES
  MULTIPRECISION INTEGER VARIABLES
  FIXED DATA TYPE

* GENERAL PURPOSE
  SUPPORTS SEQUENTIAL FILES AND RANDOM ACCESS FILES
  READABLE AND EASILY MODIFIED

COMPARISONS WITH OTHER LANGUAGES

* COMPARISON WITH ALGOL 60

  - ALGOL 60 WAS THE BASIS FOR PASCAL

    PRINCIPLES OF STRUCTURING
    FORM OF EXPRESSIONS

  - ALGOL 60 WAS NOT ADOPTED AS A SUBSET OF PASCAL BECAUSE SOME CONSTRUCTION PRINCIPLES ESPECIALLY THOSE OF DECLARATIONS WOULD NOT HAVE ALLOWED A CONVANIENT REPRESENTATION OF THE ADDITIONAL FEATURES OF PASCAL.

  - EXTENSIONS TO ALGOL 60

    RECORD AND FILE STRUCTURES
    (THE LACK OF DATA STRUCTURING FACILITIES IN ALGOL 60 WAS CONSIDERED THE PRIME CAUSE OF ITS RELATIVELY NARROW RANGE OF APPLICABILITY.)

* COMPARISON WITH PL/1

  - PASCAL CAN BE PROCESSED BY IMPLEMENTATIONS ON SMALL COMPUTERS

  - PASCAL MORE EASILY PRODUCES OPTIMIZED CODE

  - PASCAL PROVIDES BETTER CONTROL STRUCTURES

  - PASCAL REQUIRES EXPLICIT DATA TYPES

  - PASCAL REQUIRES EXPLICIT TYPE TRANSFERS

  - PASCAL AVOIDS UNNECESSARY ALTERNATIVE FUNCTIONS

* COMPARISON WITH FORTRAN

MOST OF THE DIFFERENCES BETWEEN PASCAL AND FORTRAN ARE A RESULT OF PASCAL'S DATA STRUCTURING FACILITIES AND ITS EMPHASIS ON PRODUCING STRUCTURED, RELIABLE CODE THAT IS READABLE AND EASILY MODIFIED.

- PASCAL HAS IMPROVED CONTROL STRUCTURES FOR CONTROLLING THE FLOW OF A PROGRAM

- THE 'GOTO' STATEMENT IS UNNECESSARY

- CERTAIN UNRELIABLE STATEMENTS HAVE BEEN OMITTED FROM PASCAL:

  THE ARITHMETIC 'IF' STATEMENT
  THE COMPUTED 'GOTO' STATEMENT
  THE 'EQUIVALENCE' STATEMENT

- PASCAL IS A STACK IMPLEMENTED LANGUAGE.

  ONLY THE ROUTINES WHICH ARE CURRENTLY INVOKED HAVE STACK SPACE ALLOCATED.

  SUBROUTINE VARIABLE VALUES ARE NOT SAVED ONCE THE SUBROUTINE ENDS

- PASCAL RESTRICTS FUNCTION SIDE EFFECTS

- PASCAL REQUIRES EXPLICIT VARIABLE DECLARATIONS

- PASCAL HAS STRONG TYPE CHECKING

- PASCAL IS NATURALLY REENTRANT

EXTENSIONS TO THE PASCAL LANGUAGE

TI PASCAL CONFORMS TO THE DESCRIPTION OF THE PASCAL LANGUAGE GIVEN IN THE "PASCAL USER MANUAL AND REPORT" BY KATHLEEN JENSEN AND NIKLAUS WIRTH. IN ADDITION, TI PASCAL OFFERS A NUMBER OF ENHANCEMENTS

* COMMON VARIABLES
  - REQUIRE AN 'ACCESS' DECLARATION
  - MAY BE SHARED WITH EXTERNAL ROUTINES
  - EXIST BEYOND THE EXECUTION OF THE ROUTINE IN WHICH THEY ARE DECLARED
* DYNAMIC BOUNDS FOR ARRAYS AND SETS
* MULTIPRECISION INTEGER VARIABLES
* MULTIPRECISION REAL VARIABLES
* 'FIXED' DATA TYPE
  - MAXIMUM PRECISION IS 31 BITS
* 'DECIMAL' DATA TYPE
  - MAXIMUM PRECISION IS 15 DECIMAL DIGITS
* 'ESCAPE' STATEMENT
* EXPLICIT TYPE OVERRIDE OPERATOR
* ASSERT STATEMENT
  - TESTS A CONDITION WHICH SHOULD BE TRUE AT A GIVEN POINT IN THE PROGRAM. IF THE CONDITION IS NOT TRUE, A RUNTIME ERROR OCCURS.
* LINKAGE TO EXTERNAL FORTRAN AND ASSEMBLY LANGUAGE SUBROUTINES USING FORTRAN LINKAGE
* ADDITIONAL TYPE CHECKING FOR PROCEDURE AND FUNCTION PARAMETERS

MODIFICATIONS TO PASCAL

* RESTRICTION OF FUNCTION SIDE EFFECTS

- A FUNCTION MAY NOT CHANGE THE VALUE OF THE FOLLOWING:
  A NONLOCAL VARIABLE
  A VARIABLE PARAMETER OF THE FUNCTION
  A 'COMMON' VARIABLE
  A POINTER VARIABLE FOLLOWED BY @

- A USER DEFINED FUNCTION MAY NOT CONTAIN:
  CALLS TO USER DEFINED PROCEDURES OR THE STANDARD PROCEDURE 'READ'

  CALLS TO EXTERNALLY DEFINED FUNCTIONS

  PROCEDURES OR EXTERNALLY DEFINED FUNCTIONS AS PARAMETERS

  A 'WITH' STATEMENT THAT CONTAINS A RECORD VARIABLE FOLLOWED BY @

  CALLS TO 'NEW' OR 'DISPOSE' THAT HAVE PARAMETERS THAT ARE NOT EITHER LOCAL VARIABLES OR VALUE PARAMETERS

  THE ARRAY INTO WHICH DATA IS PACKED BY 'PACK' MUST BE A LOCAL VARIABLE OR A VALUE PARAMETER

  THE ARRAY INTO WHICH DATA IS UNPACKED BY 'UNPACK' MUST BE A LOCAL VARIABLE OR A VALUE PARAMETER

  THE STRING INTO WHICH DATA IS PLACED BY 'ENCODE' MUST BE A LOCAL VARIABLE OR A VALUE PARAMETER

  THE VARIABLE INTO WHICH DATA IS PLACED BY 'DECODE' MUST BE A LOCAL VARIABLE OR A VALUE PARAMETER

  THE NONFILE PARAMETERS OF A 'READ' PROCEDURE AND THE 'OVAL' PARAMETER OF AN 'IOTERM' PROCEDURE MUST BE LOCAL VARIABLES OR VALUE PARAMETERS

  PROCEDURES 'RESET', 'REWRITE', 'EXTEND', 'WRITEEOF', 'SKIPFILES', 'CLOSE', 'SETNAME', 'SETMEMBER', AND 'IOTERM' MAY BE USED ONLY WITH PARAMETERS OF FILE TYPE THAT ARE LOCAL VARIABLES

25

* LIMITED USE OF THE 'GOTO' STATEMENT
  - A PROCEDURE, FUNCTION, OR PROGRAM WITH LABEL DECLARATIONS WILL NOT HAVE ITS CODE OPTIMIZED BY THE TIP COMPILER
  - YOU MAY NOT JUMP INTO OR OUT OF A PROCEDURE OR FUNCTION
  - YOU MAY NOT JUMP INTO A 'FOR' OR 'WITH' STATEMENT

* EXTENDED FORM OF THE 'WITH' STATEMENT

* LOCAL SCOPE OF 'FOR' STATEMENT INDEX

* 'OTHERWISE' CLAUSE AND SUBRANGE CASE LABELS WITH THE 'CASE' STATEMENT

* MORE FLEXIBLE I/O
  - AUTOMATIC CREATION OF FILES FOR THE DEFAULT FILES 'INPUT' AND 'OUTPUT'
  - CRU I/O

* CONSTANTS AS PROGRAM PARAMETERS

* THE PREDEFINED SYMBOL 'MAXINT' IS NOT SUPPORTED

* ALTERED PRECEDENCE FOR LOGICAL OPERATORS 'NOT', 'AND', AND 'OR'.
  - THESE OPERATORS ARE EVALUATED IN THIS ORDER AND HAVE THE LOWEST OPERATOR PRIORITY

```
PROGRAM SAMPLE;
VAR TIME, MAX, MIN : REAL;

PROCEDURE READ_ECHO (VAR TIME : REAL);
BEGIN
    READ(TIME);
    WRITELN('TIME  = ',TIME );
END;

PROCEDURE DETERMINE_MAX_MIN ( VAR MAX, MIN : REAL; TIME : REAL);
BEGIN
    IF TIME > MAX THEN MAX := TIME
    ELSE IF TIME < MIN THEN MIN := TIME
END;


(* MAIN *)
BEGIN
    RESET ( INPUT );
    IF NOT EOF THEN BEGIN
        READ_ECHO (TIME );
        MAX := TIME; MIN := TIME ;
        IF EOLN THEN READLN;
    END;
    WHILE NOT EOF DO
        IF NOT EOLN THEN BEGIN
            READ_ECHO (TIME );
            DETERMINE_MAX_MIN(MAX, MIN, TIME);
        END
        ELSE READLN;
    WRITELN;
    WRITELN('MAX TIME IS ',MAX,' MIN TIME IS ',MIN);
END.
```

DATA TYPES

OBJECTIVES

STUDENTS SHOULD BE ABLE TO WRITE DECLARATIONS FOR VARIABLES AND CONSTANTS OF THE FOLLOWING TYPES:

- INTEGER
- LONG INTEGER
- REAL

STUDENTS SHOULD BE ABLE TO USE THESE DATA TYPES IN EXPRESSIONS TO SOLVE A SPECIFIED PROBLEM.

AGENDA

1. IDENTIFIERS
2. VARIABLE DECLARATIONS
3. DATA TYPES
   - INTEGER
   - LONG INTEGER (LONGINT)
   - REAL
4. EXPRESSIONS
5. TYPE DECLARATIONS
6. CONSTANT DECLARATIONS

VARIABLES AND DATA TYPES

IDENTIFIERS

DEF: NAMES DENOTING ONE OF THE FOLLOWING:

- CONSTANTS
- TYPES
- VARIABLES
- PROCEDURES
- FUNCTIONS
- ESCAPE LABELS
- PROGRAMS

SYNTAX:

o MUST BEGIN WITH A LETTER OR $

o REST OF CHARACTERS MAY BE

- LETTER
- DIGIT
- $
- UNDERSCORE (_)

o NO LIMIT ON LENGTH

o CANNOT BE THE SAME AS A RESERVED IDENTIFIER

RESERVED IDENTIFIERS:

ACCESS	DOWNTO	INTEGER	RANDOM
AND	ELSE	LABEL	REAL
ARRAY	END	LONGINT	RECORD
ASSERT	ESCAPE	MOD	REPEAT
BEGIN	FALSE	NIL	SET
BOOLEAN	FILE	NOT	TEXT
CASE	FIXED	OF	THEN
CHAR	FOR	OR	TO
COMMON	FUNCTION	OTHERWISE	TRUE
CONST	GOTO	OUTPUT	TYPE
DECIMAL	IF	PACKED	UNTIL
DIV	IN	PROCEDURE	VAR
DO	INPUT	PROGRAM	WHILE
			WITH

EXAMPLES:

LEGAL	ILLEGAL
READ_CHARACTER	_VALUE
READ_CHAR	ARRAY
ARRAY3	3TIMES
ABS	VAR

VARIABLES AND DATA TYPES

VARIABLES

DEF: IDENTIFIER WHICH MAY BE USED TO STORE A DATA VALUE OF A SPECIFIED TYPE.

EVERY VARIABLE USED IN A PASCAL PROGRAM, MUST BE DECLARED IN A 'VAR' DECLARATION.

'VAR' DECLARATIONS

SYNTAX:

VAR < V LIST > : < TYPE >; [< V LIST > : < TYPE >] ...

WHERE,

< V LIST > - LIST OF VARIABLE NAMES WHICH ARE BEING DECLARED

< TYPE > - THE DATA TYPE WHICH MAY BE STORED IN ANY OF THE IDENTIFIERS IN < V LIST >

< TYPE > MAY BE:

- O INTEGER
- O LONGINT
- O BOOLEAN
- O CHAR
- O SCALAR
- O SUBRANGE

ENUMERATION TYPES

- O REAL
- O FIXED
- O DECIMAL
- O POINTER
- O ARRAY
- O RECORD
- O SET
- O FILE

- O USER DEFINED TYPE

EXAMPLE: VAR COUNTER, INDEX, MAX : INTEGER;

SALARY, COST, RATE : REAL;

**NOTE: THE KEY WORD 'VAR' APPEARS ONLY ONCE.

VARIABLE TYPES

INTEGER TYPE

RANGE:

-32768 .. 32767 (16 BIT, 2'S COMPLEMENT)

DECLARATION:

VAR < V LIST > : INTEGER;

EXAMPLE:

VAR COUNT, TEST : INTEGER;

INTEGER OPERATORS

\+ - * DIV MOD

DIV - INTEGER DIVISION (E.G. 5 DIV 2 = 2)

MOD - REMAINDER (MODULO) (E.G. 7 MOD 3 = 1)

EXTENDED INTEGERS (LONGINT)

RANGE:

-2147483648 .. 2147483647 (32 BIT, 2'S COMPLEMENT)

DECLARATION:

VAR < V LIST > : LONGINT;

OPERATORS:

SAME AS FOR INTEGERS

**NOTE: CONSTANTS MAY BE LONGINT BY USING THE SUFFIX 'L'

E.G. 725766L

INTEGER EXPRESSIONS

```
     VAR  VAL, NVAL, IVAL : INTEGER;

            .
            .
            .

          VAL := 27;              (* THIS IS A COMMENT *)
          NVAL := VAL MOD 6;      (* ':=' IS THE ASSIGNMENT OPERATOR *)

(* WHAT VALUE IS ASSIGNED TO IVAL IN THE FOLLOWING 'STATEMENT'?? *)

          IVAL := NVAL + 9 DIV 2; (* ________ *)
          VAL := #FF;             (* '#' MEANS HEXADECIMAL *)
```

OPERATOR PRECEDENCE

*	/	MOD	DIV				HIGHEST PRECEDENCE
+	-						
<	=	>	<=	>=	<>	IN	
NOT							
AND							
OR							LOWEST PRECEDENCE

** NOTE: OPERATORS WITH THE SAME PRECEDENCE ARE EVALUATED LEFT TO RIGHT

COMPILER OPTION CKOVER

- ENABLES OR DISABLES CHECKING OVERFLOW WHEN EVALUATING INTEGER, LONGINT, DECIMAL, AND FIXED EXPRESSIONS
- DEFAULT IS FALSE

REAL TYPE

RANGE:

FRACTION - 6 TO 7 DECIMAL DIGITS OF PRECISION

EXPONENT - 1.0 X 10**-78 .. 1.0 X 10**75

**NOTE: THIS STANDARD PRECISION MAY BE OVERRIDDEN BY SPECIFYING THE DESIRED ACCURACY WHEN A VARIABLE IS DECLARED

E.G. VAR X : REAL (12) --> 12 DECIMAL DIGITS

DECLARATION:

VAR < V LIST > : REAL [(< ACCURACY >)];

EXAMPLE:

```
VAR  TEST, NEW : REAL ( 13 );
     A, B      : REAL;
```

REAL CONSTANTS

SYNTAX:

< DIGITS > . < DIGITS > [E / Q] [+ / -] < DIGITS >

OR

< DIGITS > [E / Q] [+ / -] < DIGITS >

WHERE,

< DIGITS > = 0 .. 9

E = SCIENTIFIC NOTATION

Q = NUMBER IS REPRESENTED INTERNALLY AS AN EXTENDED PRECISION REAL NUMBER

EXAMPLES:

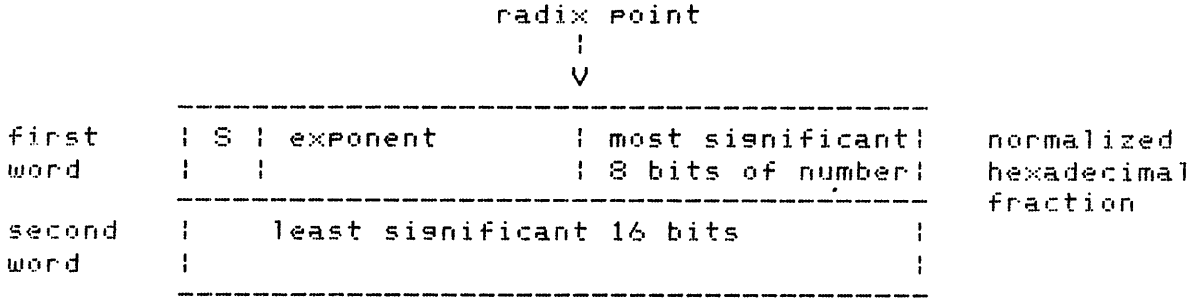
LEGAL: 0.3, 5E-3, 87.36E8, 12.9E-5

ILLEGAL: 3, 450, 26.E5, .001, #120.1

**NOTE: THE DECIMAL POINT MUST BE SURROUNDED BY DIGITS!!!

REAL NUMBER REPRESENTATION

SINGLE PRECISION REAL NUMBERS ARE STORED IN 2 16-BIT WORDS AS FOLLOWS:



S - SIGN OF THE FRACTION

EXPONENT - HEXADECIMAL EXPONENT FOR THE FRACTION. EXPONENT IS BIASED BY >40 (I.E. >41 = 1, >40 = 0, >3F = -1, ETC.)

NUMBER - 24 BIT HEXADECIMAL NORMALIZED FRACTION

EXTENDED PRECISION REAL NUMBERS ARE STORED IN 4 16-BIT WORDS. THE SIZE OF THE EXPONENT IS NOT INCREASED.

EXAMPLES

NUMBER(10)	REAL REP.	HEX REP.
1.0	0 41 100000	4110 0000
100.0	0 42 640000	4264 0000
-1.0	1 41 100000	C110 0000
.03125	0 3F 800000	3F80 0000

REAL OPERATORS

* / + -

**NOTE: THERE IS NO EXPONENTIATION OPERATOR

REAL EXPRESSIONS

```
VAR  A, B : REAL;
       .
       .
       .
   A := 4.7;
   B := 3.6;
   B := ( A + 15.0 ) * ( B - 2.1 ) (* PAREN'S HAVE THE USUAL MEANING *)
```

MIXED MODE

< EXPRESSION > < OPERATOR > < EXPRESSION >

- REAL RESULT IF AT LEAST ONE OPERAND IS REAL.
- INTEGER RESULT IF BOTH OPERANDS ARE INTEGER.

WHERE,

< OPERATOR > IS +, -, *

< EXPRESSION > / < EXPRESSION > IS ALWAYS REAL

- DIV AND MOD MAY ONLY OPERATE ON INTEGER EXPRESSIONS

- INTEGER VALUES MAY BE IMPLICITLY CONVERTED TO REAL BY ASSIGNING IT TO A REAL VARIABLE.

< INTEGER VARIABLE > := < REAL EXPRESSION > -- ILLEGAL

< REAL VARIABLE > := < INTEGER EXPRESSION > -- LEGAL

**NOTE: IN ALL OTHER CASES, THE TYPE OF THE EXPRESSION SHOULD MATCH THAT OF THE VARIABLE ON THE LEFT OF THE EQUAL SIGN.

MIXED MODE EXAMPLES

EXPRESSION		REAL VALUE
5 / 2	=	2.5
5 DIV 2 * 10.0	=	20.0
SQRT (9)	=	3.0
4.0 DIV 2.0	=	* ILLEGAL *
3 / (3 * 10.0)	=	0.1

EXAMPLE:

```
PROGRAM ERRORS;

(* THIS PROGRAM ILLUSTRATES MODE ERRORS *)


VAR     PI, PIOVERTWO : REAL;
        INT           : INTEGER;

BEGIN (* ERRORS *)
     PI := 3.14159;
     PIOVERTWO := PI DIV 2;      (* ILLEGAL *)
     INT := PI / 2;              (* ILLEGAL *)
     INT := ROUND ( PI / 2 )     (*    OK    *)
END. (* ERRORS *)
```

CONSTANT DECLARATIONS

VALUES MAY BE ASSIGNED TO VARIABLE NAMES BY USING CONSTANT DECLARATIONS.

SYNTAX :

CONST < C NAME > = < C EXPRESSION >; [< C NAME > = < C EXPR >] ...

WHERE,

< C NAME > - IDENTIFIER FOR CONSTANT

< C EXPRESSION > - EXPRESSION WHICH MAY BE EVALUATED AT COMPILE TIME.

NOTES:

- THE TYPE OF THE EXPRESSION DETERMINES THE TYPE OF THE CONSTANT

- NEW VALUES MAY ** NOT ** BE ASSIGNED TO A CONSTANT AT RUN TIME!

EXAMPLE:

```
CONST   PI = 3.14159;
        R  = 27.0;

VAR   AREA : REAL;

        .
        .
        .

  AREA := PI * R * R;

  PI := 2.7;    (* ILLEGAL *)
```

TYPE DECLARATIONS

NEW DATA TYPES MAY BE SPECIFIED OR STANDARD DATA TYPES RENAMED BY USING A 'TYPE' DECLARATION.

SYNTAX:

TYPE < T NAME > = < TYPE >; [< T NAME > = < TYPE >;] ...

WHERE,

< T NAME > - NEW TYPE IDENTIFIER

< TYPE > - SAME AS FOR THE 'VAR' DECLARATION

EXAMPLE:

```
TYPE  XREAL = REAL( 15 );
      LI    = LONGINT;

          .
          .
          .

VAR     A, B : REAL;
        C, D : XREAL;
        W, X : INTEGER;
        Y, Z : LI;
```

**NOTE: OTHER USES FOR THE 'TYPE' DECLARATION WILL BE ADDRESSED LATER.

WORKSHEET 1

1. DECLARE A CONSTANT C TO HAVE A VALUE OF 200.

2. DECLARE A VARIABLE I_NUM TO BE OF TYPE INTEGER

3. DECLARE A VARIABLE RNUM1 AND RNUM2 TO BE OF TYPE REAL.

4. WRITE A STATEMENT TO ASSIGN RNUM1 THE VALUE OF THE FOLLOWING EXPRESSION. (C IS A CONSTANT, B IS A REAL VARIABLE.)

$$\frac{RNUM2 \times 2.0}{C - (B + 1)}$$

5. DECLARE A NEW TYPE R8 WHICH WILL BE A REAL NUMBER WITH 8 DECIMAL PLACES OF ACCURACY.

GIVEN THE FOLLOWING DECLARATIONS, INDICATE WHETHER THE EXPRESSIONS ARE LEGAL:

```
CONST RC = 5.236
VAR R1, R2 : REAL;
    I1, I2 : INTEGER;
```

6. R1 := R2 / 2;

7. R2 := I1 + 12;

8. I1 := R2;

9. I2 := 3 / 4;

10. I2 := 3.0 DIV 4.0;

STATEMENTS

O STATEMENTS PERFORM ARITHMETIC AND LOGIC OPERATIONS

E.G.

- CALL PROCEDURES AND FUNCTIONS
- CONTROL SEQUENCE OF PROGRAM EXECUTION
- PERFORM EVALUATION AND ASSIGNMENTS OF VALUES

SIMPLE STATEMENTS

- ASSIGNMENT
- PROCEDURE
- ESCAPE
- GOTO
- ASSERT
- EMPTY

STRUCTURED STATEMENTS

- COMPOUND
- CONDITIONAL
- REPETITIVE
- WITH

PASCAL PROGRAM FORMAT

```
PROGRAM < PROG. NAME >;

    < LABEL DECLARATIONS >;
    < CONSTANT DECLARATIONS >;
    < TYPE DECLARATIONS >;
    < VARIABLE DECLARATIONS >;
    < COMMON DECLARATIONS >;
    < ACCESS DECLARATIONS >;
    < PROCEDURE AND FUNCTION DECLARATIONS >;

    BEGIN

        [ BODY OF PROGRAM  ( BLOCK ) ]

    END.
```

MODULE 4

PROGRAM CONTROL STRUCTURES I

OBJECTIVES

- BE ABLE TO EVALUATE BOOLEAN EXPRESSIONS

- BE ABLE TO USE THE FOLLOWING PROGRAM STRUCTURES
  IF-THEN-ELSE STATEMENT
  COMPOUND STATEMENT
  WHILE STATEMENT
  FOR LOOP

TESTING METHOD: WORKSHEET
PROGRAMMING ASSIGNMENT 1

AGENDA
---

1. BOOLEAN EXPRESSIONS
   - RELATIONAL OPERATORS

2. PROGRAMMING STRUCTURES
   - IF-THEN-STATEMENT
   - COMPOUND STATEMENT
   - IF-THEN-ELSE STATEMENT
   - NESTED IF STATEMENT
   - WHILE LOOP
   - FOR LOOP

WORKSHEET

PROGRAM CONTROL STRUCTURES

BOOLEAN EXPRESSIONS

DEFINITION: A BOOLEAN EXPRESSION IS AN EXPRESSION WHICH EVALUATES TO TRUE OR FALSE

RELATIONAL OPERATORS

A BOOLEAN EXPRESSION CONTAINS ONE OF THE FOLLOWING RELATIONAL OPERATORS:

=	EQUALITY
<	LESS THAN
>	GREATER THAN
<>	INEQUALITY
<=	LESS THAN OR EQUAL TO
>=	GREATER THAN OR EQUAL TO

EXAMPLES:

A = 3	IS TRUE WHEN A IS 3 AND FALSE THEN A IS NOT 3
A <> B	IS TRUE WHEN A IS NOT EQUAL TO B AND FALSE WHEN A AND B ARE EQUAL
5 + 3 < 2 + 4	IS FALSE

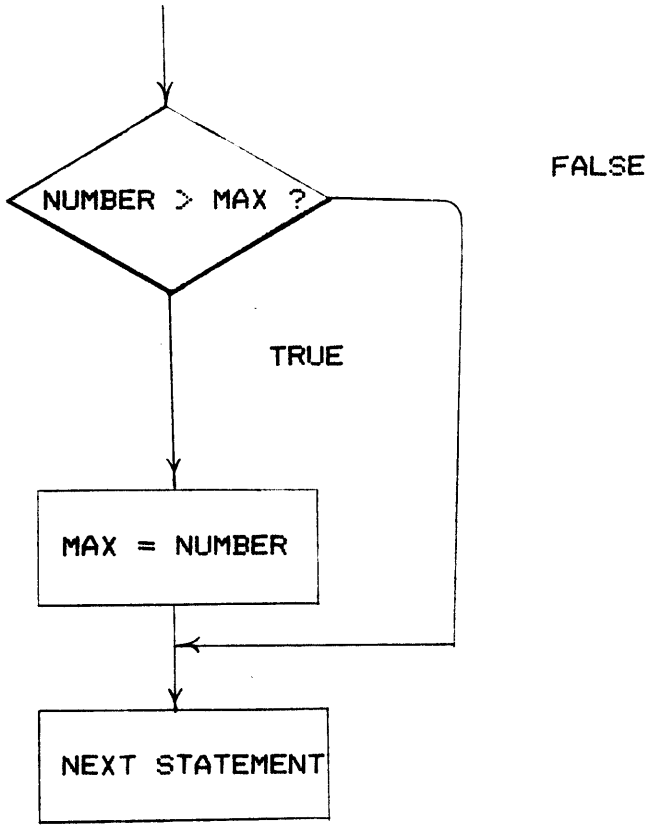
**NOTE: '=' IS A RELATIONAL OPERATOR. ':=' ASSIGNS A VALUE

IF-THEN STATEMENT

SYNTAX: IF < EXPRESSION > THEN < STATEMENT >

THE STATEMENT WILL BE EXECUTED IF THE EXPRESSION IS TRUE.
IT WILL NOT BE EXECUTED IF THE EXPRESSION IS FALSE.

EXAMPLE: IF NUMBER > MAX THEN MAX := NUMBER;



IF-THEN-ELSE STATEMENT

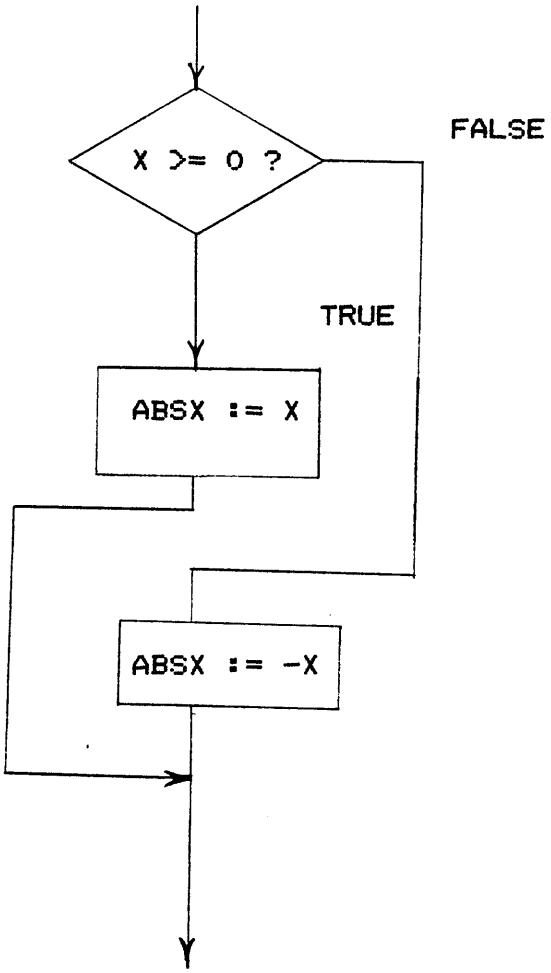
SYNTAX: IF < BOOLEAN EXPRESSION > THE < STATEMENT1 >
ELSE < STATEMENT2 >

NOTE: THERE IS NO ';' BEFORE 'ELSE'

IF THE EXPRESSION IS TRUE THEN ONLY STATEMENT1 WILL EXECUTE.
IF THE EXPRESSION IS FALSE THEN ONLY STATEMENT2 WILL EXECUTE.

EXAMPLE: FIND THE ABSOLUTE VALUE ABSX OF REAL NUMBER X.

```
VAR X, ABSX : REAL;
      .
      .
IF X >= 0 THEN
   ABSX := X
ELSE ABSX := -X;
```



COMPOUND STATEMENT

SOMETIMES YOU MAY WISH TO EXECUTE A GROUP OF STATEMENTS WHEN A CERTAIN CONDITION IS TRUE INSTEAD OF ONLY ONE. PASCAL ALLOWS STATEMENTS TO BE GROUPED TOGETHER BY USING A 'BEGIN-END' BLOCK.

```
SYNTAX:  BEGIN
           <STATEMENT>;
                 .
                 .
           <STATEMENT> (*NO ';' *)
         END
```

** NOTE: THERE IS NO ';' BEFORE 'END'

WHEN ONE OR MORE STATEMENTS ARE ENCLOSED WITHIN A 'BEGIN-END' BLOCK. THE ENTIRE GROUP FROM 'BEGIN' TO 'END' FUNCTIONS AS A SINGLE STATEMENT.

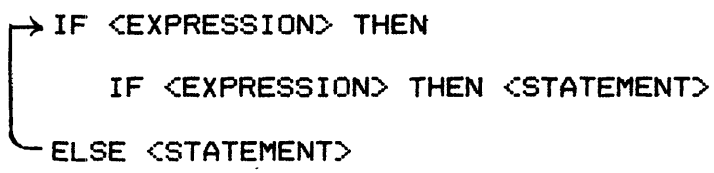
```
EXAMPLE:  IF COUNTER = MAX THEN

            BEGIN

              NUM := NUM + 1;

              AVG := SUM / COUNTER;

              COUNTER := 0                (* NO ';' *)

            END;
```

NESTED 'IF' STATEMENT

```
SYNTAX:   IF <EXPRESSION> THEN
             IF <EXPRESSION> THEN <STATEMENT>
             ELSE <STATEMENT>
          ELSE <STATEMENT>

EXAMPLE:   IF CODE = 0 THEN
              DEDUCTION := ZERODEP        (* NO ';' *)
           ELSE IF CODE = 1 THEN
              DEDUCTION = ONEDEP          (* NO ';' *)
           ELSE IF CODE = 2 THEN
              DEDUCTION = TWODEP;
```

**NOTE: EACH 'ELSE' IS ASSOCIATED WITH THE CLOSEST 'IF'

BUT -- WHAT IF YOU WANT THIS?



IN SPITE OF THE FORMAT. THE 'ELSE' WILL BE ASSOCIATED WITH THE SECOND 'IF' SINCE IT IS THE CLOSEST ONE.

THIS PROBLEM CAN BE SOLVED BY USING A 'BEGIN-END' BLOCK:

```
          IF <EXPRESSION> THEN

            BEGIN

              IF <EXPRESSION> THEN <STATEMENT>

            END          (* NO ";" *)

          ELSE <STATEMENT>;
```

EXAMPLE: A SURVEY IS BEING MADE OF THE NUMBER OF TRAFFIC TICKETS RECEIVED BY DRIVERS. DRIVERS ARE GROUPED BY:

- SEX
- AGE (< 21, > = 21)
- # OF TICKETS (< = 5, 6 TO 10, > 10)

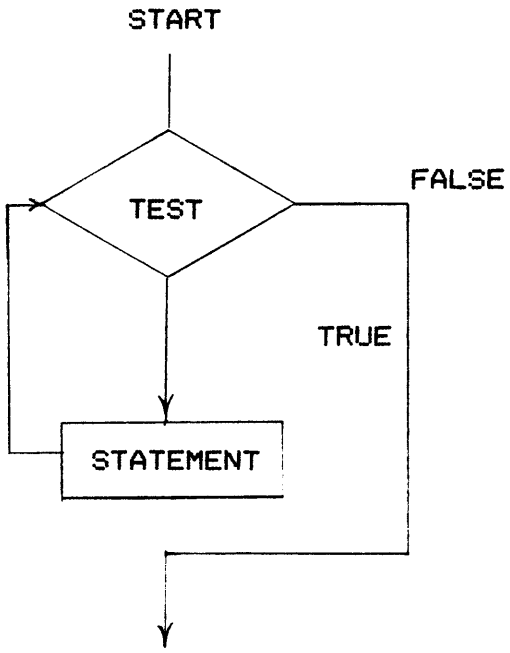
GROUP1 MIGHT BE FEMALES UNDER 21 WITH 5 OR FEWER TICKETS.

WRITE AN 'IF' STATEMENT TO SUM THE GROUPS FOR MEN.
ASSUME SEX CODES ARE 'F' AND 'M'.

WHILE LOOP

SYNTAX: WHILE <BOOLEAN EXPRESSION> DO <STATEMENT>

- THE WHILE STATEMENT SPECIFIES THAT A CERTAIN STATEMENT WILL BE REPEATEDLY EXECUTED WHILE A CERTAIN CONDITION IS TRUE. IF THE STATEMENT IS INITIALLY FALSE THE EXPRESSION WILL NOT BE EXECUTED AT ALL.
- THE STATEMENT MAY BE A COMPOUND STATEMENT.
- THE EXPRESSION IS EVALUATED BEFORE EACH ITERATION OF THE LOOP. SO KEEP IT SIMPLE.



**NOTE: THE VARIABLE CONTROLLING THE LOOP MUST BE MODIFIED IN THE STATEMENT TO AVOID AN INFINITE LOOP.

```
EXAMPLE ;  FIND THE SMALLEST N SUCH THAT
           1 + 1/2 + 1/3 +...+ 1/N IS GREATER THAN SOME
           CONSTANT C

PROGRAM FINDN;

CONST C = 2.5;
VAR N : INTEGER;
    SUM : REAL;

BEGIN
  N := 0;
  SUM := 0;
  WHILE SUM <= C DO
    BEGIN
      N := N + 1;
      SUM := SUM + 1 / N
    END;

(* PRINT OUT N *)

END.    (* FINDN     *)


EXAMPLE:  COMPUTE N FACTORIAL

PROGRAM NFACTORIAL;

VAR I, J, N : INTEGER;

BEGIN (* NFACTORIAL *)
   [ READ A VALUE FOR N ]
   I := 1;
   J := 1;
   WHILE I < N DO
      BEGIN
        I := I + 1;
        J := I * J
      END;
   [ WRITE OUT J ]
END. (* NFACTORIAL *)
```

FOR-LOOP

* THE FOR-LOOP IS LIKE THE 'DO' STATEMENT IN FORTRAN.

SYNTAX: FOR <VARIABLE> := <EXPRESSION> TO <EXPRESSION> DO

<STATEMENT>

OR

FOR <VARIABLE> := <EXPRESSION> DOWNTO <EXPRESSION> DO

<STATEMENT>

EXAMPLE:

```
PROGRAM FORLOOP;

 VAR N : INTEGER;
     SUM1, SUM2 : REAL;

BEGIN
  READ(N);
  SUM1 := 0;
  SUM2 := 0;
  FOR I := 1 TO N DO
     BEGIN
       SUM1 := SUM1 + 1 / I;
       SUM2 := SUM2 + SUM1
     END;
  WRITE(SUM1, SUM2)
END.
```

EXAMPLE: Write a FOR LOOP that will compute N Factorial
for some given N.

**NOTE: THE LIMITS OF THE FOR LOOP ARE CONSTANT AND ARE NOT CHANGED INSIDE THE LOOP.

```
FOR I := 1 TO N DO
    N := N + 1
```

IF N WAS ORIGINALLY 10, THE LOOP WILL BE EXECUTED 10 TIMES.

THE INDEX OF THE FOR LOOP DOES NOT HAVE TO BE DECLARED.

COMPILER OPTION FORINDEX

- ENABLES OR DISABLES THE ISSUING OF WARNING MESSAGES WHEN NAMES OF FOR CONTROL VARIABLES ARE IDENTICAL TO NAMES OF OTHER ACCESSIBLE VARIABLES.
- DEFAULT IS FALSE

SUMMARY

- THE 'IF' STATEMENT SPECIFIES THAT A STATEMENT BE EXECUTED IF A CERTAIN CONDITION IS TRUE. IF IT IS FALSE, EITHER NO STATEMENT IS EXECUTED OR THE STATEMENT FOLLOWING THE 'ELSE' IS EXECUTED.

- 'IF' STATEMENTS MAY BE NESTED. THE 'ELSE' IS ASSOCIATED WITH THE MOST RECENT 'IF'.

- PASCAL ALLOWS US TO FORM A COMPOUND STATEMENT CONSISTING OF ONE OR MORE STATEMENTS ENCLOSED BY A BEGIN-END. A COMPOUND STATEMENT ACTS EXACTLY LIKE A SINGLE STATEMENT.

- A BOOLEAN EXPRESSION EVALUATES TO TRUE OR FALSE. IT CONTAINS ONE OF THE RELATIONAL OPERATORS =, <, >, <=, >=, <>

- THE WHILE STATEMENT ALLOWS A STATEMENT TO BE EXECUTED REPEATEDLY WHILE A CERTAIN CONDITION IS TRUE.

- THE FOR LOOP ALLOWS A STATEMENT TO BE EXECUTED REPEATEDLY A SPECIFIED NUMBER OF TIMES. IT IS SIMILAR TO THE 'DO' LOOP IN FORTRAN.

WORKSHEET 2

EVALUATE THE FOLLOWING BOOLEAN EXPRESSIONS:

1. 4 > 1

   N <= 10

2. WHEN N = 7

3. WHEN N = 10

4. WHEN N = 11

WRITE AN 'IF' STATEMENT FOR EACH OF THE FOLLOWING:

5. ADD 1 TO THE VALUE OF I WHEN I IS LESS THAN N.

6. IF A IS LARGER THAN B THEN DIFF IS EQUAL TO A - B, BUT
   IF B IS LARGER THAN A THEN DIFF IS EQUAL TO B - A.

7. USE A COMPOUND STATEMENT IN WRITING THIS 'IF' STATEMENT:
   IF A IS LESS THAN OR EQUAL TO B THEN A EQUALS A TIMES B AND
   COUNTER EQUALS COUNTER PLUS 1. OTHERWISE COUNTER EQUALS 0.

8. USE A NESTED 'IF' STATEMENT IN WRITING THE FOLLOWING:
   IF A = B AND IF C = D THEN SUM = SUM + D
   BUT IF C = E THEN SUM = SUM + E
   BUT IF C IS ANYTHING ELSE, SUM = 0.

9. WRITE A 'WHILE' STATEMENT THAT WILL ADD 1 TO COUNTER AND
   ADD GRADE TO SUM AS LONG AS COUNTER IS LESS THAN OR EQUAL TO
   NUMGRADES.

10. USING A FOR LOOP CALCULATE THE VALUE OF THE FOLLOWING
    SUM WHEN I GOES FROM 1 TO 20: 1 + 1/2 + 1/3 + ... + 1/I

INPUT/OUTPUT

OBJECTIVES:

- BE ABLE TO DECLARE A VARIABLE TO BE OF TYPE CHAR AND ASSIGN IT A VALUE.
- BE ABLE TO READ AND WRITE CHARACTERS OR NUMBERS USING A TEXTFILE OR THE DEFAULT FILES 'INPUT' AND 'OUTPUT'.
- BE ABLE TO SHOW ON A DIAGRAM HOW THE INPUT MUST LOOK FOR A GIVEN READ STATEMENT
- BE ABLE TO SHOW HOW THE OUTPUT WILL LOOK AFTER A GIVEN WRITE STATEMENT IS EXECUTED.
- BE ABLE TO USE THE BOOLEAN FUNCTIONS EOF AND EOLN TO TEST FOR THE END OF FILE AND THE END OF A LINE.

AGENDA

1. CHAR DATA TYPE

2. TEXTFILES

3. INPUT
   - READ STATEMENT

WORKSHEET

   - RESET
   - READLN
   - EOF
   - EOLN

4. OUTPUT

WORKSHEET

TYPE CHAR

A VALUE OF TYPE CHAR IS AN ELEMENT OF THE ORDERED SET OF CHARACTERS REPRESENTED THE MACHINE'S CHARACTER SET.

THE ORDERING IS IMPLEMENTATION DEPENDENT.

SYNTAX: VAR CH : CHAR;

WRITING CHARACTERS:

- CHARACTER VALUES ARE WRITTEN AS A SINGLE CHARACTER SURROUNDED BY APOSTROPHES
  'A' '3' '+'

- THE APOSTROPHE ' AND THE NUMBER SIGN # ARE REPRESENTED BY 2 CONSECUTIVE CHARACTERS
  '''' '##'

- A CHARACTER MAY BE WRITTEN AS A # FOLLOWED BY ITS 2 DIGIT HEXIDECIMAL CHARACTER CODE

A	CAN BE WRITTEN	'A'	OR	#41
Z	CAN BE WRITTEN	'Z'	OR	#5A
1	CAN BE WRITTEN	'1'	OR	#31

- NOTICE THAT THE CHARACTER REPRESENTATION OF 1 IS NOT THE SAME AS THE NUMBER 1. YOU CANNOT PERFORM ARITHMETIC OPERATIONS ON CHARACTERS.

Determine Which of the followine are legal:

```
CH := 'A' ;

CH := 65 ;

CH := #65 ;

CH := '#41';

CH := 'B' + CH2 ;
```

TEXTFILES

DEFINITION: A FILE WHOSE COMPONENTS ARE CHARACTERS. TEXTFILES ARE THE MOST COMMONLY USED FILES.

- THE TYPES WHICH MAY BE READ TO OR FROM A TEXTFILE ARE

CHAR	
INTEGER	
LONGINT	
REAL	
DECIMAL	
FIXED	
BOOLEAN	TRUE, FALSE, T, OR F
STRINGS	

- FOR TYPES OTHER THAN CHAR AN IMPLICIT DATA CONVERSION IS MADE.

SYNTAX: VAR FILENAME : TEXT;

DEFINES A FILE OF TYPE TEXT

EXAMPLE: VAR TFILE : TEXT;

DEFINES TFILE AS A TEXTFILE

FILES 'INPUT' AND 'OUTPUT'

PASCAL PROVIDES TWO DEFAULT FILES, 'INPUT' AND 'OUTPUT' WHICH THE PROGRAMMER CAN USE WITHOUT SPECIFYING THE FILE NAME.

READ STATMENT - UNFORMATTED

SYNTAX: READ (FILENAME, V1, ..., VN) READS FROM FILE FILENAME

READ (VL, ..., VN) READS FROM DEFAULT FILE INPUT

EXAMPLES:

READ (TFILE, CH) READS A CHARACTER FROM FILE TFILE AND ASSIGNS ITS VALUE TO CH.

READ (DATA, NUM1, NUM2) READS 2 VALUES FROM FILE DATA AND ASSIGNS THEM TO NUM1 AND NUM2.

READ (CH) READ FROM THE DEFAULT FILE 'INPUT'

NUMBERS -- THE ENTIRE FIELD IS READ

GIVEN THE DECLARATIONS
VAR I : INTEGER;

```
: 2 : 3 : 1 :   : 4 :
--------------------
^
```

EXECUTING READ (I) RESULTS IN

```
: 2 : 3 : 1 :   : 4 :
--------------------
          ^
```

I = 231

** NOTE: FIELDS MUST BE SEPARATED BY A BLANK

CHARACTERS -- MUST BE READ ONE AT A TIME

GIVEN THE DECLARATIONS
VAR CH : CHAR;

```
: T : H : I : S :   : I : S :
-----------------------------
^
```

EXECUTING READ (CH) RESULTS IN

```
: T : H : I : S :   : I : S :
-----------------------------
    ^
```

CH = 'T'

WORKSHEET 3

1. GIVEN CH OF TYPE CHAR, WRITE A STATEMENT ASSIGNING CH THE VALUE 'A'.

2. ASSIGN CH THE VALUE OF THE CHARACTER WHOSE HEX CHARACTER CODE IS 41

3. WRITE A DECLARATION STATEMENT TO DECLARE A FILE 'NAMES' AS A TEXTFILE

WRITE A READ STATEMENT FOR EACH OF THE FOLLOWING:

4. READ ONE INTEGER N FROM A FILE CALLED 'DATA'

5. READ TWO NUMBERS R1 AND R2 FROM THE DEFAULT FILE 'INPUT'

6. SHOW HOW THE DATA FOR THE PREVIOUS READ WOULD LOOK ON A CARD IF R1 = 12345 AND R2 = 4762

:	:	:	:	:	:	:	:	:	:	:	:

7. READ A CHARACTER CH FROM THE DEFAULT FILE 'INPUT'

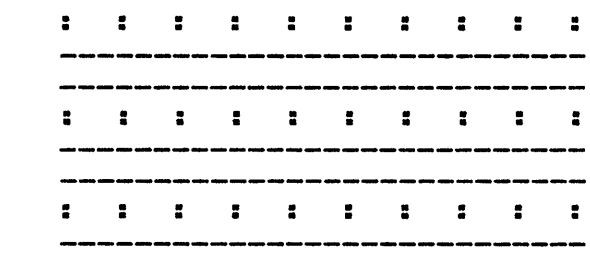
RESET STATEMENT

DEFINITION: A FILE MUST BE 'RESET' BEFORE IT MAY BE READ. RESET OPENS THE FILE AND POSITIONS THE FILE POINTER TO THE BEGINNING OF THE FILE.

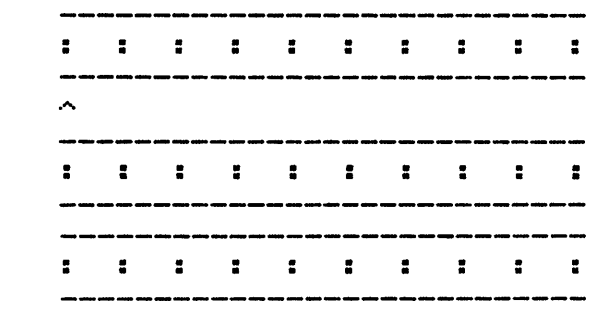
SYNTAX: RESET (FILENAME)

EXAMPLE:
```
RESET (INPUT)          RESETS THE FILE 'INPUT'
RESET (DATA)           RESETS THE FILE 'DATA'
```

SUPPOSE THE INPUT FILE CONSIST OF THE THREE LINES



THEN RESET (INPUT) YIELDS



EXAMPLE: READ A CHARACTER FROM DEFAULT FILE 'INPUT'

```
RESET ( INPUT );
WHILE CH <> ' ' DO
  BEGIN
    READ (CH);
    (*PROCESS CH*)
  END;
```

NOTE: 'INPUT' MUST BE SPECIFIED IN THE RESET STATEMENT.

EOF FUNCTION -- HAS VALUES TRUE OR FALSE

DEFINITION: INDICATES IF THE END OF THE FILE HAS BEEN REACHED.
EOF WILL HAVE A VALUE OF TRUE IF THE LAST RECORD
IN THE FILE HAS BEEN READ AND FALSE OTHERWISE.

SYNTAX: EOF (FILENAME)

EOF - DEFAULT FILE 'INPUT' IS ASSUMED

```
EXAMPLE:  IF NOT EOF THEN
              BEGIN

              (*  PROGRAM   *)

              END;
```

EOLN FUNCTION -- HAS VALUES TRUE OR FALSE

DEFINITION: INDICATES WHEN THE END OF LINE HAS BEEN REACHED. EOLN WILL
HAVE A VALUE OF TRUE IF THE LAST CHARACTER ON THE LINE HAS
BEEN READ.

SYNTAX: EOLN (FILENAME)

EOLN - DEFAULT FILE 'INPUT' IS ASSUMED

```
EXAMPLE: WHILE NOT EOF DO
           IF NOT EOLN THEN
              BEGIN
                 READ (CH);

                    (*  PROCESS (CH)   *)

              END;
```

**NOTE: WHEN READING CHARACTERS, IF THE EOLN MARKER
IS ENCOUNTERED, A BLANK IS READ

:	:	:	:	:	EOLN :	EOF :
					^	

AT THIS POINT EOLN = TRUE, EOF = FALSE

READ (CH) => CH = , EOLN = FALSE, EOF = TRUE

READLN STATEMENT

- SKIPS TO THE BEGINNING OF THE NEXT LINE.
- READLN DOES NOT READ A CHARACTER.
- THE NEXT CHARACTER READ AFTER EXECUTING A A 'READLN' WILL BE THE FIRST CHARACTER OF THE NEXT LINE.

SYNTAX: READLN (FILENAME)

READLN ASSUMES DEFAULT FILE INPUT

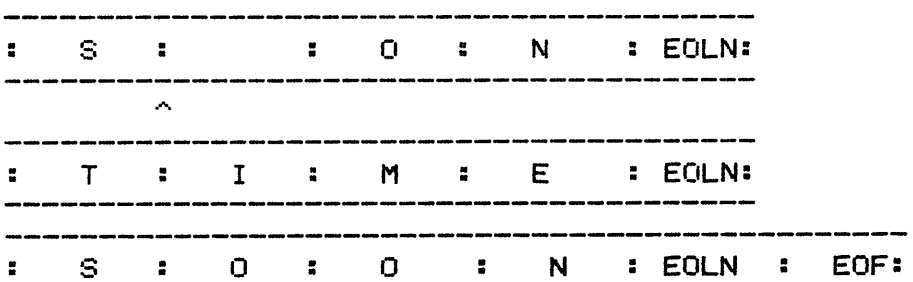
EXAMPLE:

```
VAR  CH:CHAR;
     .
     .
WHILE NOT EOF DO
   IF NOT EOLN DO
      BEGIN
        READ ( CH );
        WRITE ( CH );
      END
   ELSE READLN;
        .
        .
        .
```

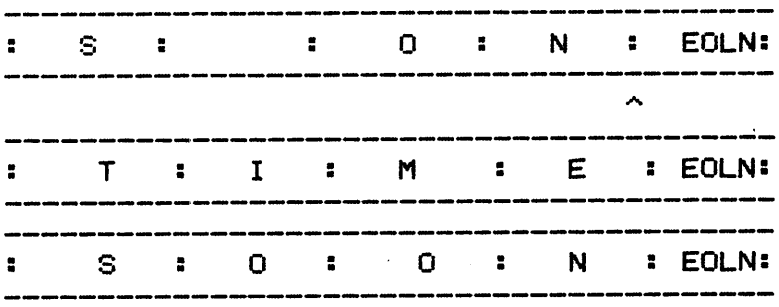
BEFORE A READ IS EXECUTED

EOLN = FALSE
EOF = FALSE

READ (CH) YIELDS



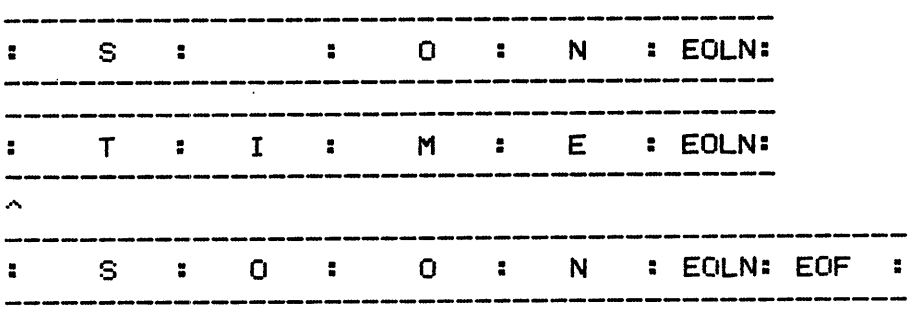
THE FIRST CHARACTER HAS BEEN ASSIGNED TO CH. SUCCESSIVE READ STATEMEN
EVENTUALLY RESULT IN THE FILE POSITION BEING AT THE END OF THE LINE.



EOLN = TRUE

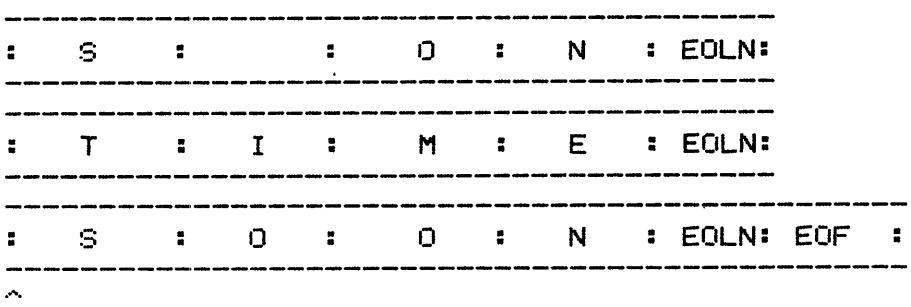
EOF = FALSE

EOLN IS TRUE AND EOF IS STILL FALSE. READ (CH) RESULTS
IN CH = ' ', EOLN (INPUT) = FALSE, EOF (INPUT) = FALSE AND THE FILE
POSITION IS:



EOLN = FALSE
EOF = FALSE

FROM THIS POSITION IF A READLN (INPUT) IS EXECUTED, THE POSITION WOULD
AT THE BEGINNING OF THE NEXT LINE.



SUCCESSIVE READS WILL YIELD:

```
-----------------------------------------
:   S   :       :   O   :   N   : EOLN  :
-----------------------------------------
-----------------------------------------
:   T   :   I   :   M   :   E   : EOLN  :
-----------------------------------------
-------------------------------------------------
:   S   :   O   :   O   :   N   : EOLN  :  EOF :
-------------------------------------------------
                          ^
```

AT WHICH POINT EOLN (INPUT) IS TRUE AND EOF (INPUT) IS FALSE.

ANOTHER READ (INPUT, CH) RESULTS IN

```
-----------------------------------------
:   S   :       :   O   :   N   : EOLN  :
-----------------------------------------
-----------------------------------------
:   T   :   I   :   M   :   E   : EOLN  :
-----------------------------------------
-------------------------------------------------
:   S   :   O   :   O   :   N   : EOLN  :  EOF  :
-------------------------------------------------
                                  ^
```

NOW CH = ' ', EOLN (INPUT) = FALSE, AND EOF (INPUT) = TRUE.

**NOTE: WHEN THE END OF A TEXTFILE IS REACHED, FIRST EOLN IS TRUE ANDFALSE. EOF IS FALSE. THEN THE NEXT READ MAKES EOLN FALSE AND EOF TRUE.

YOU DO NOT HAVE TO DO A 'READLN' TO GET TO THE START OF THE NEXT LINE.

WHEN READING NUMBERS, YOU MUST CHECK FOR EOLN IF YOU CHECK FOR EOF. THIS IS BECAUSE EOF IS NOT SET TO TRUE UNLESS EOLN IS READ FIRST. IF YOU ARE READING NUMBERS, EOLN WILL NEVER BE READ SINCE IT IS NOT A NUMBER.

WORKSHEET 4

1. WRITE A STATEMENT TO RESET THE DEFAULT FILE 'INPUT'

INDICATE BY DRAWING AN ARROW ON THE CHART WHEN THE FILE POINTER WILL BE AFTER THE EXECTUTION OF EACH STATEMENT & FILL IN THE VALUE FOR CH, NUM, EOLN, AND EOF.
ASSUME THE FOLLOWING DECLARATION HAS BEEN MADE:
VAR CH : CHAR; NUM : INTEGER;

2.RESET (INPUT)

```
-------------------------
: J : A : N : E : 3 : EOLN:        EOLN =
-------------------------
-------------------------
: T : I :   : 1 : 2 : EOLN:        EOF =
-------------------------
---------------------------------
: 4 : 3 :   : N : O : EOLN: EOF:
---------------------------------
```

3. READ(CH);

```
-------------------------
: J : A : N : E : 3 : EOLN:        CH =
-------------------------
^
-------------------------
: T : I :   : 1 : 2 : EOLN:        EOLN =
-------------------------
---------------------------------
: 4 : 3 :   : N : O : EOLN: EOF:   EOF =
---------------------------------
```

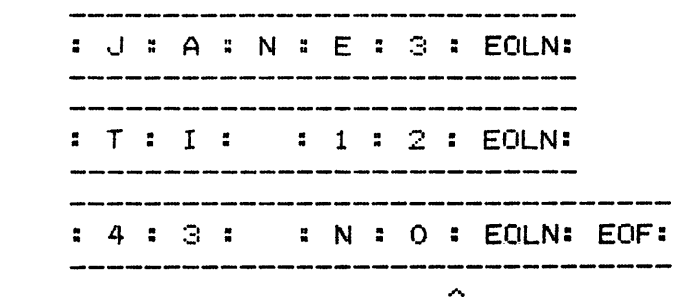
4.READ (NUM)

```
-------------------------
: J : A : N : E : 3 : EOLN:        NUM =
-------------------------
              ^
-------------------------
: T : I :   : 1 : 2 : EOLN:        EOLN =
-------------------------
---------------------------------
: 4 : 3 :   : N : O : EOLN: EOF:   EOF =
---------------------------------
```

5. READLN

```
-------------------------
: J : A : N : E : 3 : EOLN:        EOLN =
-------------------------
   ^
-------------------------
: T : I :   : 1 : 2 : EOLN:        EOF =
-------------------------
---------------------------------
: 4 : 3 :   : N : O : EOLN: EOF:
---------------------------------
```

6. READ (CH)



CH =

EOLN =

EOF =

7. GIVEN VAR CH : CHAR;

WRITE AN IF-THEN-ELSE TO READ CH IF THE END OF LINE HASN'T BEEN REACHED, AND IF IT HAS, DO A READLN TO POSITION THE FILE POINTER TO THE FIRST OF THE NEXT LINE.

REWRITE STATEMENT

DEFINITION: MAKES THE FILE EMPTY AND OPENS IT FOR OUTPUT. REWRITE DOES NOT HAVE TO BE USED WITH THE DEFAULT FILE 'OUTPUT'.

SYNTAX: REWRITE (F)

WRITE STATEMENT

SYNTAX: WRITE (FILENAME, X1, ..., XN) WRITES TO FILE NAME GIVEN

WRITE (X1, ..., XN) WRITES TO DEFAULT 'OUTPUT'

EXAMPLE: PRINT OUT VALUES OF SUM1 AND SUM2 ON ONE LINE TO THE DEFAULT FILE 'OUTPUT'

```
WRITE (SUM1, SUM2);

        OR

WRITE (SUM1);
WRITE (SUM2);
```

- CONSECUTIVE WRITES CONTINUE TO WRITE ON THE SAME LINE

DEFAULT FIELD WIDTH: INTEGERS - 10
REALS - 15 (Reals are printed in E format)

WRITING COMMENTS: COMMENTS MAY BE WRITTEN BY ENCLOSING IN APOSTROPHES

EXAMPLE: WRITE ('WEEKLY SALES REPORT')

```
RESULTS IN
WEEKLY SALES REPORT
WRITE (N1, '      TOTAL 1S', N2)

FOR N1 = 1, N2 = 52
RESULTS IN

_________1      TOTAL 1S________52
```

**NOTE: REMEMBER THE APOSTROPHE IS REPRESENTED BY ''

WRITELN STATEMENT

- TERMINATES A LINE

SYNTAX: WRITELN (F, X1, . . ., XN)

WRITES X1, . . ., XN AND TERMINATES THE LINE

WRITELN WRITES FROM FILE 'OUTPUT'
TERMINATES THE LINE WITHOUT WRITING ANYTHING

EXAMPLES: WRITELN (SUM1, SUM2);

SUM1 AND SUM2 WILL BE WRITTEN ON ONE LINE TO THE DEFAULT FILE 'OUTPUT' AND THE LINE WILL BE TERMINATED

```
WRITELN (SUM1)
WRITELN (SUM2)
```

WRITES SUM1 AND SUM2 ON DIFFERENT LINES

```
WHILE CH <> ' ' DO
   BEGIN
     READ (CH);
     WRITE (CH)
   END;
WRITELN;
```

CONTINUES WRITING ON SAME LINE UNTIL CH IS A BLANK
THEN GOES TO THE NEXT LINE.

SUMMARY

- A VALUE OF TYPE CHAR IS A CHARACTER. THE CHARACTER SET IS DEPENDENT ON THE MACHINE. A CHARACTER IS REPRESENTED AS A NUMBER INSIDE THE MACHINE WHICH RECOGNIZES THAT NUMBER AS A CODE FOR THAT PARTICULAR CHARACTER. A NUMBER SUCH AS 5 IS NOT REPRESENTED IN THE SAME WAY AS THE CHARACTER 5.

- CHARACTERS MAY BE WRITTEN AS A SINGLE CHARACTER SURROUNDED BY APOSTROPHES OR AS A # SIGN FOLLOWED BY ITS 2 DIGIT HEXADECIMAL CHARACTER CODE.

- A TEXTFILE IS A FILE OF CHARACTERS DIVIDED LOGICALLY INTO LINES. SEVERAL TYPES MAY BE WRITTEN TO OR READ FROM A TEXTFILE.

- PASCAL PROVIDES DEFAULT FILES 'INPUT' AND 'OUTPUT'. ON THE 990 THE FILES TO BE ASSIGNED TO 'INPUT' AND 'OUTPUT' MUST BE SPECIFIED WHEN GIVING THE SCI COMMAND 'XPT' TO EXECUTE THE PASCAL PROGRAM.

- THE READ STATEMENT 'READ (FILENAME, X1, . . , XN) WILL READ ONE OR MORE VALUES FROM FILE 'FILENAME'. IF NO FILE IS SPECIFIED, 'READ (X1, . . , XN) THE FILE ASSIGNED TO 'INPUT' IS ASSUMED.

- THE RESET STATEMENT OPENS A FILE AND POSITIONS IT TO THE BEGINNING. A RESET STATEMENT MUST BE EXECUTED BEFORE READING FROM A FILE. THE DEFAULT FILE 'INPUT' MUST BE SPECIFIED IN THE RESET STATEMENT.

- THE WRITE STATEMENT WRITE (FILENAME, X1, . . , XN) WRITE ONE OR MORE VALUES TO THE FILE 'FILENAME'. CONSECUTIVE WRITE STATEMENTS WRITE ON THE SAME LINE. COMMENTS MAY BE WRITTEN OUT BY ENCLOSING THEM IN APOSTROPHES.

- THE WRITELN STATEMENT FINISHES WRITING A LINE. THE NEXT WRITE WILL BEGIN ON A NEW LINE.

- THE REWRITE STATEMENT OPENS A FILE FOR OUTPUT. FILES OTHER THAN THE DEFAULT FILE 'OUTPUT' MUST BE OPENED BEFORE BEING WRITTEN TO.

WORKSHEET 5

1. WRITE NUM1, NUM2, NUM3 ON ONE LINE

   USING 1 STATEMENT:

2. USING 3 STATEMENT:

3. WRITE 'QUARTERLY REPORT'

4. WRITE A STATEMENT WHICH WILL WRITE OUT THE WORD 'NAME' AT THE END A LINE AND THEN BEGIN A NEW LINE.

5. GIVEN THE FOLLOWING CODE SHOW THE OUTPUT

```
VAR, X, Y:  INTEGER
X := 5;
Y := 10;
WRITE (X);
WRITE (Y);
WRITELN;
WRITE (X, Y);
        .
        .
        .
```

SIMPLE PROCEDURES

OBJECTIVES

STUDENTS SHOULD BE ABLE TO DECLARE AND UTILIZE PROCEDURES TO PERFORM SPECIFIED OPERATIONS ON A SET OF PARAMETERS.

STUDENTS SHOULD BE ABLE TO DECLARE A FUNCTION.

STUDENTS SHOULD BE ABLE TO SPECIFY WHETHER A VARIABLE IS GLOBAL OR LOCAL TO A ROUTINE.

STUDENTS SHOULD BE ABLE TO PASS PRARMETERS BY 'VALUE' AND 'REFERENCE'.

AGENDA

1. SIMPLE PROCEDURES
   - SYNTAX
   - PARAMETER PASSING
2. SCOPE OF VARIABLES
   - GLOBAL
   - LOCAL
3. PROCEDURE ACCESSING RULES
4. FUNCTIONS

SIMPLE PROCEDURES

PASCAL PROCEDURES ARE FUNCTIONALLY VERY SIMILAR TO FORTRAN SUBROUTINES.

PURPOSE: TO PERFORM A SET OF PREDEFINED OPERATIONS ON A SET OF PARAMETERS.

EXAMPLE:

```
PROGRAM TESTPROC;

   VAR  VAL1, VAL2 : INTEGER;
        RESULT     : LONGINT;

   PROCEDURE ADDUP ( X, Y : INTEGER; VAR Z : LONGINT );

      VAR  TEMP : LONGINT;

      BEGIN (* ADDUP *)
         TEMP := X+Y;
         IF TEMP < 0 THEN
            Z := 0
         ELSE
            Z := TEMP
      END; (* ADDUP *)

   BEGIN (* TESTPROC *)
      RESET ( INPUT );
      WHILE NOT EOF DO
         BEGIN
            READ ( VAL1, VAL2 );
            ADDUP ( VAL1, VAL2, RESULT );
            IF RESULT <> 0 THEN
               WRITELN ( ' RESULT IS : ', RESULT )
            ELSE
               WRITELN( ' RESULT IS < 0 ' );
            READLN
         END
   END. (* TESTPROC *)
```

PASCAL PROGRAM SYNTAX

```
PROGRAM < PROG. NAME >;

    < LABEL DECLARATIONS >;
    < CONSTANT DECLARATIONS >;
    < TYPE DECLARATIONS >;
    < VARIABLE DECLARATIONS >;
    < COMMON DECLARATIONS >;
    < ACCESS DECLARATIONS >;
    < PROCEDURE AND FUNCTION DECLARATIONS >;

    BEGIN

        [ BODY OF PROGRAM  ( BLOCK ) ]

    END.
```

PASCAL PROCEDURE DECLARATION SYNTAX

```
PROCEDURE < PROCEDURE NAME > [ ( < PARAMETER LIST > ) ];

    [ DECLARATIONS ]

    BEGIN

        [ BODY OF PROCEDURE ( BLOCK ) ]

    END;
```

PROCEDURE PARAMETERS

FORMAL PARAMETERS - THOSE SPECIFIED IN THE PROCEDURE DEFINITION. THE 'FORMAL' (DUMMY) PARAMETERS TAKE ON THE VALUE OF THE 'ACTUAL' PARAMETERS WHICH ARE SPECIFIED IN A PROCEDURE CALL.

ACTUAL PARAMETERS - THOSE SPECIFIED IN A PROCEDURE CALL. THESE ARE VARIABLE NAMES, EXPRESSIONS, ETC., WHOSE VALUES THE 'FORMAL' PARAMETERS WILL TAKE ON FOR PERFORMING THE OPERATION OF THE PROCEDURE.

PROCEDURE PARAMETER LIST

PROCEDURE < P. NAME > [(< PARAM. LIST >)];

< PARAM. LIST > - LIST OF 'FORMAL PARAMETER SPECIFICATIONS' SEPARATED BY SEMICOLONS.

FORMAL PARAMETER SPECIFICATION

[VAR] < FORMAL PARAM NAME > [, < FORMAL PARAM NAME >] ... : TYPE;

EXAMPLE:

```
PROCEDURE TEST (       X                : INTEGER;
                       Y                : REAL;
                       YES, NO          : BOOLEAN;
                VAR    Z                : REAL;
                VAR    MAYBE, COULDBE   : BOOLEAN );
```

*NOTE: The TYPE description must be a one word name.

ACTUAL VS FORMAL PARAMETERS

ACTUAL (CALLING) PARAMETERS - THOSE APPEARING IN A PROCEDURE CALL

FORMAL (DUMMY) PARAMETERS - THOSE APPEARING IN THE PROCEDURE DEFINITION

**RULE: CORRESPONDING ACTUAL AND FORMAL PARAMETERS MUST BE OF EXACTLY THE SAME TYPE. (THERE IS ONE EXCEPTION TO THIS RULE TO ALLOW FOR VARIABLE SIZE ARRAYS)

EXAMPLE:

```
VAR X : ARRAY [ 1..10 ] OF INTEGER;

VAR Y : ARRAY [ 0..9 ] OF INTEGER;
```

HERE, X AND Y ARE NOT COMPATIBLE AS MATCHING ACTUAL AND FORMAL PARAMETERS SINCE THEY DO NOT START AND END WITH THE SAME INDICES.

EXAMPLE:

```
PROGRAM TEST;
   VAR I, J : REAL;
   PROCEDURE ONE ( X : REAL;  VAR Y : REAL );      FORMAL PARMS
     BEGIN (* ONE *)
     END; (* ONE *)
  BEGIN (* TEST *)
        ONE ( I, J );                              ACTUAL PARMS
  END. (* TEST *)
```

'VAR' FORMAL PARAMETERS (CALLED BY REFERENCE)

WHEN A VALUE IS ASSIGNED TO A 'VAR' PARAMETER, THE CORRESPONDING ACTUAL PARAMETERS VALUE IS CHANGED ALSO.

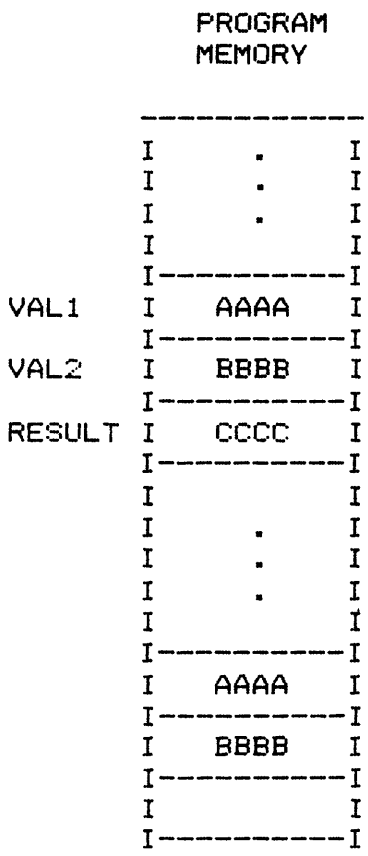
THIS ALLOWS COMPUTED VALUES TO BE RETURNED TO THE CALLING ROUTINE.

FORMAL PARAMETERS WITH NO 'VAR' (CALLED BY VALUE)

A COPY OF THE ACTUAL PARAMETER IS MADE FOR USE IN THE PROCEDURE.

IF A 'VALUE' PARAMETER IS CHANGED, ONLY THE COPY IS CHANGED AND THE CORRESPONDING ACTUAL PARAMETER DOES NOT CHANGE.

EXAMPLE:




```
PROGRAM PARMS;

   VAR VAL1, VAL2, RESULT : INTEGER;

   PROCEDURE TEST (      X, Y : INTEGER;
                     VAR Z    : INTEGER );
         .
         .
         .

      END; (* TEST *)

   BEGIN (* PARMS *)

      TEST ( VAL1, VAL2, RESULT );

   END. (* PARMS *)
```

WORKSHEET 6

FILL IN THE SPACES IN THE COMMENTS. RESPOND WITH THE VALUE OF THE VARIABLE AT THE POINT WHERE THE COMMENT APPEARS IN THE PROGRAM.

```
PROGRAM TEST;

   VAR  A, B : INTEGER;

   PROCEDURE AD( X : INTEGER; VAR Y : INTEGER );
      BEGIN (* AD *)
         Y := X + Y;
      END; (* AD *)

   BEGIN (* TEST *)
      A := 14;
      B := 2;
      AD( A, B );
1.  (*
                     ???  A =           , B =
                                                          *)
      AD( B, A );
2.  (*
                     ???  A =           , B =
                                                          *)
   END. (* TEST *)
```

2. GIVEN THE FOLLOWING PROGRAM, SPECIFY WHICH OF THE CALLS TO THE PROCEDURE 'HELP' ARE LEGAL.

** DON'T WORRY ABOUT WHAT THE PROCEDURE DOES!!!!! **

```
PROGRAM TEST;

   TYPE NEW = ARRAY [ 11..20 ] OF INTEGER;

   VAR A : ARRAY [ 1..10 ] OF INTEGER;
       B : INTEGER;
       C : LONGINT;
       D : NEW;

   PROCEDURE HELP (       X : NEW;
                    VAR   Y : LONGINT   );

      VAR MAX : INTEGER;

      BEGIN (* HELP *)
         MAX := X [ 11 ];
         FOR I := 12 TO 20 DO
            IF MAX < X [ I ] THEN
               MAX := X [ I ];
         X := MAX
      END; (* HELP *)

   BEGIN (* TEST *)
      FOR I := 1 TO 10 DO
         READ ( A [ I ] );
      FOR I := 11 TO 20 DO
         READ ( A [ I ] );

   (* A *)  HELP ( B, C, A ); (* Y     , N     : WHY                *)
                                 ____    ____     _______________

   (* B *)  HELP ( B, A );    (* Y     , N     : WHY                *)
                                 ____    ____     _______________

   (* C *)  HELP ( A, B );    (* Y     , N     : WHY                *)
                                 ____    ____     _______________

   (* D *)  HELP ( D, C );    (* Y     , N     : WHY                *)
                                 ____    ____     _______________

   (* E *)  HELP ( A, C );    (* Y     , N     : WHY                *)
                                 ____    ____     _______________

   END. (* TEST *)
```

SCOPE OF VARIABLES

GLOBAL VARIABLES - ACCESSIBLE FROM ANYWHERE IN THE PROGRAM.
- GLOBAL VARIABLES MUST BE DECLARED AT THE 'PROGRAM' LEVEL.

LOCAL VARIABLES - ACCESSIBLE INSIDE THE PROCEDURE IN WHICH THEY ARE DECLARED.

EXAMPLE

```
PROGRAM SCOPE;

  VAR    X, Y : INTEGER;

  PROCEDURE ONE ( Q : INTEGER );

    VAR    A : INTEGER;

    BEGIN (* ONE *)
          .
          .          (*  X, Y, A ARE ACCESSIBLE *)
          .
    END; (* ONE *)

  BEGIN (* SCOPE *)
        .
        .            (*   X, Y ARE ACCESSIBLE *)
        .
  END. (* SCOPE *)
```

**NOTES:
- A IS SAID TO BE LOCAL TO THE PROCEDURE 'ONE'
- X AND Y ARE GLOBAL VARIABLES

EXAMPLE

```
PROGRAM SCOPE1;

  VAR    X, Y : INTEGER;

  PROCEDURE ONE ( Q : INTEGER );

    VAR   A : INTEGER;

  PROCEDURE TWO ( R : INTEGER );

      VAR   B : INTEGER;

      BEGIN (* TWO *)
            .
            .              (*  X, Y, B  ARE ACCESSIBLE *)
            .
      END; (* TWO *)

    BEGIN (* ONE *)
          .
          .                (*  X, Y, A  ARE ACCESSIBLE *)
          .
    END; (* ONE *)

  PROCEDURE THREE ( S : INTEGER );

    VAR   C : INTEGER;

    BEGIN (* THREE *)
          .
          .                (*  X, Y, C  ARE ACCESSIBLE *)
          .
    END; (* THREE *)

  BEGIN (* SCOPE1 *)
        .
        .                  (*  X, Y  ARE ACCESSIBLE *)
        .
  END. (* SCOPE1 *)
```

NOTE: SPACE FOR LOCAL VARIABLES IS ALLOCATED WHEN THE PROCEDURE IS CALLED, AND THE SPACE IS RELEASED WHEN THE PROCEDURE IS EXITED.

DUPLICATING VARIABLE NAMES

**RULE: IF A VARIABLE IS DECLARED IN A PROCEDURE WHICH HAS THE SAME NAME AS A VARIABLE WHICH IS GLOBAL TO THAT PROCEDURE, THE NEWLY DECLARED VARIABLE WILL BE 'ACTIVE' UNTIL THE PROCEDURE IS EXITED.

EXAMPLE:

```
PROGRAM TEST;

   VAR I : INTEGER;

   PROCEDURE A ( .. );

      VAR I : REAL;

      * HERE I IS REAL

   PROCEDURE B ( .. );

      VAR I : BOOLEAN;

      * HERE I IS BOOLEAN



    * HERE I IS INTEGER
```

**RULE: IF A FORMAL PARAMETER IS DEFINED WHICH HAS THE SAME NAME AS A VARIABLE WHICH IS GLOBAL TO THE PROCEDURE, THEN THE FORMAL PARAMETER IS 'ACTIVE' AND THE GLOBAL VARIABLE IS NOT 'ACTIVE'

EXAMPLE:

```
PROGRAM TEST;

     VAR I : INTEGER;
         J : REAL;

   -- PROCEDURE ONE ( I : REAL );
   I
   I
   I        * I TAKES ON THE VALUE OF J AT THE TIME
   I          OF THE CALL.  THE VALUE OF THE GLOBAL
   I          VARIABLE 'I' IS NOT AFFECTED.
   I
   ------   END; (* ONE *)

      BEGIN (* TEST *)


           ONE ( J );


      END. (* TEST *)
```

ACCESSIBILITY OF PROCEDURES

THE ORDER IN WHICH PROCEDURES ARE DEFINED AND THE 'LEVEL' AT WHICH THEY ARE DEFINED WILL DETERMINE WHICH OTHER PROCEDURES MAY CALL THEM.

** A PROGRAM OR PROCEDURE MAY CALL ITSELF

** A PROCEDURE MAY CALL ANOTHER PROCEDURE WHICH IS DEFINED WITHIN IT

```
PROGRAM T

    PROCEDURE A

    PROCEDURE B
```

PROGRAM T MAY CALL ITSELF, PROCEDURE A, AND PROCEDURE B.

** A PROCEDURE MAY CALL ANOTHER PROCEDURE AT THE SAME LEVEL IF IT HAS ALREADY BEEN DEFINED.

```
PROGRAM T

    PROCEDURE A

    PROCEDURE B

    PROCEDURE C
```

PROCEDURE C MAY CALL ITSELF, PROCEDURE A, PROCEDURE B, AND PROGRAM T

PROCEDURE B MAY CALL ITSELF, AND PROGRAM T, PROCEDURE A

FUNCTIONS

PURPOSE:

TO RETURN A SINGLE VALUE AS THE RESULT OF OPERATING ON A SET OF PARAMETERS.

FUNCTION CALLS MAY BE USED IN AN EXPRESSION SINCE THE FUNCTION NAME TAKES ON THE VALUE WHICH IS RETURNED BY THE CALL.

SYNTAX:

```
,FUNCTION < FUNC. NAME > [ ( < PARAMETER LIST > ) ] : < TYPE >;

     [ DECLARATIONS ]

     BEGIN

          [ BODY OF FUNCTION ( BLOCK ) ]

     END;
```

RULES:

- < TYPE > MAY BE

INTEGER	SUBRANGE
LONGINT	REAL
BOOLEAN	FIXED
CHAR	DECIMAL
SCALAR	POINTER

- THE NAME OF THE FUNCTION MUST APPEAR ON THE LEFT OF AN ASSIGNMENT AT LEAST ONCE IN THE BODY OF THE FUNCTION.

- THE LAST VALUE ASSIGNED TO THE FUNCTION NAME IS THE VALUE RETURNED BY THE FUNCTION.

**NOTES:

- ALL VARIABLE SCOPE RULES ARE THE SAME FOR FUNCTIONS AS THEY ARE FOR PROCEDURES

- THE FUNCTION ACCESS RULES ARE THE SAME FOR FUNCTIONS AS THEY ARE FOR PROCEDURES

EXAMPLE:

```
(*****************************************************************
*                                                                *
*     FUNCTION TITLE: POWER                                      *
*                                                                *
*    FUNCTION AUTHOR: JOSEPH PRO GRAMMER                         *
*                                                                *
*            PURPOSE: TO RAISE A REAL NUMBER TO AN INTEGER       *
*                     POWER AND RETURN THAT REAL VALUE AS        *
*                     THE VALUE OF THE FUNCTION                  *
*                                                                *
* FORMAL PARAMETERS:                                             *
*                                                                *
*     RVAL - REAL NUMBER TO BE RAISED TO AN INTEGER POWER        *
*                                                                *
*    EXPON - INTEGER POWER                                       *
*                                                                *
*                                                                *
*        CONSTRAINTS: IT IS ASSUMED THAT THE INTEGER POWER       *
*                     IS A POSITIVE INTEGER.                     *
*                                                                *
*                                                                *
*****************************************************************)

FUNCTION POWER (  RVAL : REAL;
                  EXPON : INTEGER ) : REAL;

     VAR TEMP : REAL;

     BEGIN (* POWER *)
          TEMP := 1;
          FOR I := 1 TO EXPON DO
               TEMP := TEMP * RVAL;
          POWER := TEMP
     END; (* POWER *)
```

SAMPLE CALL:

```
     NEWVAL := 37.6 + POWER( X, NUM ) * 3;
```

SIDE EFFECTS IN FUNCTIONS

DEFINITION: A SIDE EFFECT IS THE CHANGING OF A VALUE OF ANY VARIABLE WHICH IS NOT LOCAL TO THE FUNCTION.

TI PASCAL HANDLING OF SIDE EFFECTS

THE TIP COMPILER WILL NOT ALLOW THE USER TO DO THINGS WHICH COULD CAUSE SIDE EFFECTS TO OCCUR. TO PREVENT THEM THE FOLLOWING RULES APPLY:

- THE LEFT-HAND SIDE OF AN ASSIGNMENT STATEMENT OCCURRING IN A USER DEFINED FUNCTION MAY NOT CONTAIN:

  - O A VARIABLE DECLARED EXTERNAL TO THE FUNCTION (GLOBAL)
  - O A VARIABLE PARAMETER OF THE FUNCTION (MAKING 'VAR' USELESS)
  - O A COMMON VARIABLE ACCESSED WITHIN THE FUNCTION
  - O A POINTER VARIABLE FOLLOWED BY @

- IN ADDITION, USER DEFINED FUNCTIONS MAY NOT CONTAIN

  - O PROCEDURE STATEMENTS INVOLVING USER DEFINED PROCEDURES OR THE STANDARD PROCEDURES 'READ', 'NEW', OR 'DISPOSE'
  - O CALLS TO EXTERNALLY DEFINED FUNCTIONS
  - O @ IN A RECORD VARIABLE USED IN THE HEADING OF A WITH STATEMENT
  - O PROCEDURES OR EXTERNALLY DEFINED FUNCTIONS AS PARAMETERS

- THE FOLLOWING MAY ONLY BE USED WITH ARGUMENTS WHICH ARE LOCAL TO THE FUNCTION.

ENCODE	DECODE
RESET	REWRITE
PACK	UNPACK

SIDE EFFECTS EXAMPLE

```
PROGRAM SIDEEF;

     .
     .
     .

     FUNCTION SQ ( VAR A : REAL ) : REAL;

          BEGIN (* SQ *)
               A := A*A;
               SQ := A
          END; (* SQ *)

     BEGIN (* SIDEEF *)
          X := 5;
          Y := X + SQ( X );          (* Y = 30 *)
          X := 5;
          Y := SQ( X ) + X           (* Y = 50 *)
     END. (* SIDEEF *)
```

ASSIGNMENT 1

OBJECTIVES:

- TO BE ABLE TO CREATE AND RUN A PASCAL PROGRAM ON THE DX SYSTEM
- TO USE A WHILE-LOOP, IF STATEMENT AND INPUT/OUTPUT STATEMENTS AND ASSIGNMENT STATEMENTS IN WRITING A PASCAL PROGRAM.
- TO CREATE AND USE A DATA FILE.
- TO BE ABLE TO WRITE AND CALL PROCEDURES.

A series of sonar tests were made in the Pacific ocean at various locations. Sound was emitted from a sonar, bounced off the ocean floor, and was 'heard' by a receiver. the time the sound took to travel from the generator to the receiver was recorded on the file PASCAL.DATA.DATA1.

1A Read in each travel time.
Echo print each number as it is being read in.
Output one number per line along with a message.

Write program 1A without using procedures. The purpose is to give you some experience in using PASCAL and in running a PASCAL program using the DX10 operating system.

1B Read in each travel time and echo print with a message about which test was taking place.
Find the maximum and minimum travel time and print them out with a message. It is not necessary to use arrays to do this.

IMPLEMENT THIS PROGRAM BY WRITING TWO PROCEDURES:

READ_ECHO : Reads in a value for the depth and echo prints with a message

DETERMINE_MAX_AND_MIN : Compares the number just read in to MAX.
If the number is larger than MAX, reset MAX.
If not larger than MAX, compare to MIN. If smaller than MIN, reset MIN.

PARAMETERS : MAX, MIN, and NUM.

YOUR MAIN PROGRAM SHOULD DO THE FOLLOWING:

```
Reset the data file
Call READ_ECHO to read in first travel time
Initialize variables for max and min to first travel time
Do the following until you reach the end-of-file
   call READ_ECHO
   call SET_MAX_AND_MIN
Print out the maximum and minimum travel times
```

1C Add a function to your program 1B to calculate the ocean depth at the location of a test.

use the following formula :

$$DEPTH = \frac{(\text{ TRAVEL TIME }) * (\text{ 5000 FT/SEC })}{2}$$

YOUR PROGRAM SHOULD DO THE FOLLOWING :

Read in each travel time and echo print.
Calculate the ocean depth for each test. Output this with a message.
Find the maximum and minimum depths. Write these out with a message.

COMPILER OPTIONS

THERE IS A SET OF OPTIONS FOR THE PASCAL COMPILER WHICH MAY BE USED TO:

- SPECIFY CONTENT OF COMPILED LISTING
- CONTROL CONTENT OF COMPILED OBJECT
- CONTROL RUNTIME CHECKS

SPECIFYING OPTIONS

OPTIONS ARE SPECIFIED IN SPECIAL COMMENTS

SYNTAX:

```
(*$ <OPTION>, <OPTION> . . . *)
```

WHERE,

```
<OPTION> - [NO] <OPTION NAME>
        OR [RESUME] <OPTIONNAME>
```

EXAMPLE:

```
(*$NO LIST, MAP, RESUME CKOVER *)
```

SEMANTICS:

```
          <OPTION> - OPTION BECOMES TRUE

       NO <OPTION> - OPTION BECOMES FALSE

   RESUME <OPTION> - OPTION IS SET TO THE VALUE IT HAD WHEN
                     THE ROUTINE (PROGRAM) WAS ENTERED
```

SCOPE OPTIONS

OPTION COMMENTS ARE EFFECTIVE WITHIN THE SCOPE OF THE ROUTINE OR PROGRAM IN WHICH THEY OCCUR

IF AN OPTION IS CHANGED IN A ROUTINE, IT WILL TAKE ON THE VALUE BEFORE IT WAS CHANGED WHEN THE ROUTINE IS EXITED.

(I.E. YOU GET A NEW COPY OF THE OPTIONS WHEN A ROUTINE IS ENTERED)

CHANGING OPTION VALUES

PROGRAM LEVEL OPTIONS

- MAY ONLY BE CHANGED BEFORE THE 'PROGRAM' STATEMENT

ROUTINE LEVEL OPTIONS

- MAY BE CHANGED
  - BEFORE THE 'PROGRAM' STATEMENT
  - RIGHT BEFORE THE 'BEGIN' STATEMENT FOR THE BODY OF THE PROGRAM OR ROUTINE.
  - RIGHT AFTER THE 'BEGIN' STATEMENT FOR THE BODY OF A ROUTINE.

STATEMENT LEVEL OPTIONS

- MAY BE CHANGED ANYWHERE IN THE PROGRAM

LIST CONTROL OPTIONS

LIST (STATEMENT - TRUE)

- ENABLES OR DISABLES PROGRAM SOURCE LISTING
- WHEN SET TO FALSE, ONLY LINES WITH ERRORS AND THE ERROR MESSAGES ARE PRINTED

WIDELIST (PROGRAM - FALSE)

- ENABLES OR DISABLES SOURCE LINE NUMBER AND COMPOUND STATEMENT NUMBERS

MAP (ROUTINE - FALSE)

- ENABLES OR DISABLES A 'MAP' OF THE VARIABLES DEFINED IN THE ROUTINE

**NOTE: THIS IS USEFUL WHEN ATTEMPTING TO READ AN ERROR DUMP OR STACK AND HEAP MEMORY

MAP INFORMATION

IDENTIFIER NAME

KIND--VARIABLE, PARAMETER, ETC.

SIZE--IN BYTES AND BITS

STACK DISPLACEMENT

PICTURE--WHERE DATA IS STORED, FOR PACKED FIELDS ONLY

ASSERTS (STATEMENT - TRUE)

- ENABLES OR DISABLES RECOGNITION OF ASSERT STATEMENTS IN A PROGRAM

ASSERT STATEMENT

PURPOSE: TO GENERATE A RUNTIME ERROR IF A SPECIFIED BOOLEAN CONDITION IS FALSE.

SYNTAX:

ASSERT <BOOLEAN EXPRESSION>

SYMANTICS:

- IF <BOOLEAN EXPRESSION> IS TRUE, THEN CONTINUE EXECUTION OF PROGRAM
- IF <BOOLEAN EXPRESSION> IS FALSE, THEN GENERATE A RUNTIME ERROR.

MEMORY DUMPS

THE CONTENTS OF THE STACK WILL BE PRINTED OUT WHEN

- A RUNTIME ERROR OCCURS
- AN 'ASSERT' STATEMENT IS FALSE

THE COMPILER OPTION (*$ MAP *) MUST BE USED

```
DXPSCL        1.5.0  78.317   TI 990 PASCAL COMPILER   01/05/79 10:53:1

PROGRAM DUMP_IT;
(*$ MAP, ASSERTS *)

VAR COUNTER, NUM, SUM : INTEGER;

PROCEDURE SUM_IT ( VAR SUM : INTEGER;
                       NUM : INTEGER );

BEGIN
     SUM := SUM + NUM
END; (* SUM_IT *)

   MAP OF IDENTIFIERS FOR  SUM_IT

IDENTIFIER NAME   KIND        SIZE         STACK                PICTURE
                            (BYTES,BITS) DISPLACEMENT     (PACKED FIELDS ONLY)
                            LEVEL(DISPL)  (BYTE,BIT)

SUM              PARAMETER  (2,0)        #0028    INDIRECT
NUM              PARAMETER  (2,0)        #002A    DIRECT

PROCEDURE READ_ECHO ( VAR NUM : INTEGER ) ;
BEGIN
   READ ( NUM );
   WRITELN ( 'THE NUMBER READ IS ', NUM);
END; (* READ_ECHO *)

   MAP OF IDENTIFIERS FOR  READ_ECH

IDENTIFIER NAME   KIND        SIZE         STACK                PICTURE
                            (BYTES,BITS) DISPLACEMENT     (PACKED FIELDS ONLY)
                            LEVEL(DISPL)  (BYTE,BIT)

NUM              PARAMETER  (2,0)        #0028    INDIRECT

(* MAIN *)
BEGIN
   RESET ( INPUT );
   SUM := 0;
   COUNTER := 0;
   WHILE NOT EOF DO
       IF NOT EOLN THEN
           BEGIN
             READ_ECHO ( NUM );
             COUNTER := COUNTER + 1;
             SUM_IT ( SUM, NUM );
           END
       ELSE READLN;
   ASSERT COUNTER = 100;
   WRITELN('THE NUMBER OF INTEGERS READ WAS ', COUNTER);
   WRITELN('THE FINAL SUM WAS ', SUM);
END.

   MAP OF IDENTIFIERS FOR  DUMP_IT

IDENTIFIER NAME   KIND        SIZE         STACK                PICTURE
                            (BYTES,BITS) DISPLACEMENT     (PACKED FIELDS ONLY)
                            LEVEL(DISPL)  (BYTE,BIT)

COUNTER          VARIABLE   (2,0)        #0080    DIRECT
```

```
NUM                VARIABLE    (2,0)        #0082     DIRECT
SUM                VARIABLE    (2,0)        #0084     DIRECT

MAXIMUM NUMBER OF IDENTIFIERS USED = 11

INSTRUCTIONS =      4 (LESS    0 WORDS OF DEAD CODE REMOVED)
  SUM_IT    LITERALS =      14  CODE =      18  DATA =      44
INSTRUCTIONS =     23 (LESS    0 WORDS OF DEAD CODE REMOVED)
  READ_ECH  LITERALS =      40  CODE =     104  DATA =      42
INSTRUCTIONS =     73 (LESS    0 WORDS OF DEAD CODE REMOVED)
  DUMP_IT   LITERALS =      94  CODE =     312  DATA =     134
DXPSCL         1.5.0  78.317   TI 990 PASCAL COMPILER   01/05/79 10:53:1
```

```
THE NUMBER READ IS           1
THE NUMBER READ IS           2
THE NUMBER READ IS           3
THE NUMBER READ IS           4
THE NUMBER READ IS           5
THE NUMBER READ IS           6
THE NUMBER READ IS           7
THE NUMBER READ IS           8
THE NUMBER READ IS           9
THE NUMRER READ IS           0

"ASSERT" FAILED AT STATEMENT 13

      *** DUMP OF PROCESS ***

PROCESS RECORD FOR  DUMP_IT
5FEC (0000) 4E4A 5B6A 5C1A FF00 FF00 FF00 FF00 FF00  (NJ[.\..........)
5FFC (0010) FF00 FF00 FF00 FF00 FF00 FF00 FF00 FF00  (................)
600C (0020) FF00 5C1A 1594 C18F 4E4A 8000 0000 5B68  (..\.....NJ....[.)
601C (0030) 5DD0 5FEA 0000 0000 5710 56F8 0300       (].._.....W.V...  )

TOP OF STACK
5C42 (0000) 0300 0000 000D 0000 0020 001E 0000 52A0  (......... ....R.)
5C52 (0010) 56F8 5C42 5C6C 00B2 5FEC 52A0 2772 C18F  (V.\B\...._.R.'...)
5C62 (0020) 36DE 318F 0000 368C 0000 258F 5B40 5AD2  (6.1...6...%.[@Z.)
5C72 (0030) 0020 5C44 0000 5C4A 5B40 001F 0000 001F  (. \D..\J[@......)
5C82 (0040) 0000 0000 0033 0031 5C78 5CA2 2A86 5FEC  (.....3.1\.\.*._.)
5C92 (0050) 5C46 26BE D18F 5C86 001D 0000 5FEC 5B40  (\F&...\....._.[@)
5CA2 (0060) 21BA C18F 0050 0001 0000 5420 5C9C 5CC8  (!....P....T \.\.)
5CB2 (0070) 260C 5FEC 5C1A 3C24 318F 0000 0000 FFFF  (&._.\.<$1.......)

DATA AREA FOR  HALT$        LEVEL=2
5C1A (0000) FFFE FFFF 5420 2020 0001 0050 0001 5720  (....T   ...P..W )
5C2A (0010) 0001 5C1A 5C42 00CC 5FEC 5BF0 4B28 D18F  (..\.\B.._.[.K(..)
5C3A (0020) 4EBA 5BF0 4B28 5C5E                      (N.[.K(\^        )

DATA AREA FOR  ASSER$       LEVEL=2
5BF0 (0000) 5F6A 0000 0000 0000 0000 5BCA FFFE 5C20  (_.........[...\ )
5C00 (0010) 5BE0 5BF0 5C1A 1584 5FEC 5B6A 4706 5B6A  ([.[.\...._.[.G.[.)
5C10 (0020) 0000 0000 FFFF 0000 000D                 (.........       )

DATA AREA FOR  DUMP_IT      LEVEL=1
5B6A (0000) 5F6A 0000 0000 0000 0000 0000 0000 39B6  (_.............9.)
5B7A (0010) 5C20 5B6A 5BF0 4A9E 5BCA 5B52 03C8 018F  (\ [.[.J.[.[R....)
5B8A (0020) 0000 5B52 03C8 0000 0000 0000 0000 0000  (..[R............)
5B9A (0030) 0000 0000 0000 0000 0000 0000 0000 0000  (................)
5BAA (0040) 5BAA 0011 004F 5AD2 0141 0001 0050 5B40  ([..4.OZ..A...P[@)
5BBA (0050) 4F55 5450 5554 2020 0001 0100 0000 0000  (OUTPUT  ........)
5BCA (0060) 5BCA 0000 FFFE 5A44 0341 0002 0050 5AAA  ([.....ZD.A...PZ.)
5BDA (0070) 494E 5055 5420 2020 0001 0100 0000 0000  (INPUT   ........)
5BEA (0080) 000A 0000 002D                           (.....-          )

      *** DUMP OF PROCESS ***

PROCESS RECORD FOR  GO$
4E4A (0000) 5FEC 5B6A 4F50 FF00 FF00 FF00 FF00 FF00  (_.[.OP..........)
4E5A (0010) FF00 FF00 FF00 FF00 FF00 FF00 FF00 FF00  (................)
4E6A (0020) FF00 4EBA 3016 FFFF 5FEC 8000 0000 4E8C  (..N.0..._.....N.)
4E7A (0030) 516A 5296 0000 0000 5710 56F8 FFFF       (Q.R.....W.V...  )

DATA AREA FOR  TERM$        LEVEL=2
4EBA (0000) 5216 0000 0000 0000 0000 0000 0001 0000  (R...............)
```

```
4ECA (0010) 2E68 4EBA 4F16 062E 4E4A 5FEC 1560 5B6A  (..N.O...NJ_...[.)
4EDA (0020) FF00 4E8E 1560 0000 5FEC 39A4 4455 4D50  (..N.....__.9.DUMP)
4EEA (0030) 5F49 5420 2020 4558.4543 5554 494F 4E20  (_IT   EXECUTION )
4EFA (0040) 4245 4749 4E53 2EDD 0603 0000 0000 0400  (BEGINS.........)
4F0A (0050) 4F00 4F3C 4E4A 0003 0000 4F06            (O.O<NJ....O.    )

DATA AREA FOR  GO$          LEVEL=2
4E8E (0000) 5216 0000 0000 0000 0000 0000 0000 4E4A  (R.............NJ)
4E9E (0010) 0000 4E8E 4EBA 2C92 4E4A 4E76 03C8 018F  (..N.N.,.NJN.....)
4EAE (0020) 0000 0000 FFFF 0000 5FEC 5FEC            (..........__    )
*** END OF PROCESS DUMP

          *** DUMP OF HEAP ***

4DC4 (0000) 4841 4C54 2043 414C 4C45 4420 4C45 4420  (HALT CALLED LED )
4DD4 (0010) 494F 4E20 4245 4749 4E53 0000 0000 0000  (ION BEGINS......)
4DE4 (0020) 0000 0000 0000 0000 0000 0000 0000 0000  (................)
4DF4-4E03  SAME AS LAST LINE
4E04 (0040) 0000 0000 0000 0000 0000 0000 0000 0000  (................)

4E16 (0000) 092E 5359 534D 5347 3033                 (..SYSMSG03      )

4E22 (0000) 0000 0B04 0009 4DC4 0050 000C 0000 0000  (......M..P......)
4E32 (0010) 068D 0000 0000 4E16 0000 0000 0000 0000  (......N.........)
4E42 (0020) 0000 0000 01FF                           (......          )

5A44 (0000) 3120 3220 3320 3420 3520 3620 3720 3820  (1 2 3 4 5 6 7 8 )
5A54 (0010) 3920 3020 2020 2020 2020 2020 2020 2020  (9 0             )
5A64 (0020) 2020 2020 2020 2020 2020 2020 2020 2020  (                )
5A74-5A83  SAME AS LAST LINE
5A84 (0040) 2020 2020 2020 2020 2020 2020 2020 2020  (                )

5A96 (0000) 112E 4A41 4E45 2E44 4154 412E 4455 4D50  (..JANE.DATA.DUMP)
5AA6 (0010) 4954                                     (IT              )

5AAA (0000) 0000 0906 2018 5A44 0050 0000 0000 0000  (.... .ZD.P......)
5ABA (0010) 040D 0000 0000 5A96 0000 0000 0000 0000  (......Z.........)
5ACA (0020) 0000 0000 01FF                           (......          )

5AD2 (0000) 2035 4144 3220 2830 3030 3029 2032 3033  ( 5AD2 (0000) 203)
5AE2 (0010) 3520 3431 3331 2033 3333 3120 3230 3333  (0 3431 3331 2033)
5AF2 (0020) 2033 3333 3320 3331 3230 2033 3233 3320  ( 3230 2033 3233 )
5B02 (0030) 3333 3230 2020 2820 3332 3330 2032 3033  (3033  ( 3230 203)
5B12 (0040) 3233 3020 3230 3329 0000 0000 0000 0000  (230 203).........

5B24 (0000) 192E 4A41 4E45 2E50 524F 4753 2E4F 5554  (..JANE.PROGS.OUT)
5B34 (0010) 5055 542E 4455 4D50 4954                 (PUT.DUMPIT      )

5B40 (0000) 0000 0B05 0009 4FA8 0050 0002 0000 0000  (......O..P......)
5B50 (0010) 868D 0000 0000 5B24 0000 0000 8000 0001  (......[$........)
5B60 (0020) 0000 0000 01FF                           (......          )
```

DX10 USAGE

OBJECTIVES

1. STUDENTS SHOULD BE ABLE TO:

   - BID SCI
   - ASSIGN A USER ID
   - MODIFY THE TERMINAL STATUS TO REQUIRE LOG ON
   - REBID SCI USING THE ASSIGNED USER ID

2. STUDENTS SHOULD UTILIZE THE DIRECTORY STRUCTURE OF THE TRAINING DISK TO DEVELOP PROGRAMS FOR THE CLASS.

3. STUDENTS SHOULD UTILIZE SYNONYMS TO DECREASE THE NUMBER OF KEYSTROKES REQUIRED TO ENTER FILE PATHNAMES.

4. STUDENTS SHOULD BE ABLE TO USE THE TEXT EDITOR.

AGENDA

1. LOADING DX10

2. SCI

   - MODES OF OPERATION
   - SYSTEM INITIALIZATION
   - USER ID'S

3. DISK FILE MANAGEMENT

   - FILE TYPES AND ACCESS METHODS
   - DISK VOLUMES
   - DIRECTORY STRUCTURE
   - SUPPORTED DISK CHARACTERISTICS

4. DISK RELATED SCI COMMANDS

5. SUMMARY

LOADING (BOOTING) DX10

DX10 MAY BE LOADED (BOOTED) FROM:

* CARDS
* CASSETTE
* ROM'S
* MAG TAPE

DEPENDING ON YOUR SYSTEM CONFIGURATION

LOADING DX10 USING A ROM LOADER

1) PRESS THE 'HALT/SIE' SWITCH

2) PRESS THE 'RESET' SWITCH

3) PRESS THE 'LOAD' SWITCH

4) THE BOOT/LOADER EXECUTES AND DX10 IS LOADED INTO MEMORY

LOADING DX10 WITH A LOADER ON PUNCHED CARDS

1) LOAD BOOT CARD DECK INTO CARD READER HOPPER
   AND RESET THE CARD READER

2) PRESS 'HALT/SIE' SWITCH

3) PRESS THE 'RESET' SWITCH

4) PRESS THE 'CLR' SWITCH

5) SET THE DATA SWITCHES TO HEX 0080

6) PRESS 'MA ENTER' SWITCH

7) PRESS 'MDE' SWITCH

8) PRESS THE 'LOAD' SWITCH

9) THE LOADER PROGRAM IS READ FROM CARDS AND
   DX10 IS LOADED

LOADING DX10 WITH A LOADER ON CASSETTE

1) LOAD THE CASSETTE CONTAINING THE LOADER PROGRAM INTO EITHER CASSETTE UNIT ON THE MODEL 733 ASR AND PLACE THAT CASSETTE UNIT IN 'PLAYBACK' MODE

2) PRESS 'HALT/SIE' SWITCH

3) PRESS 'RESET' SWITCH

4) PRESS THE 'LOAD' SWITCH

5) THE LOADER PROGRAM IS READ FROM CASSETTE AND DX10 IS LOADED

*** NOTE: ONCE THE SYSTEM IS BOOTED, YOU MAY ACTIVATE 'SCI'

SYSTEM COMMAND INTERPRETER

* SINGLE UNIFORM INTERFACE BETWEEN USERS AND:
  - DX10
  - SOFTWARE DEVELOPMENT UTILITIES
  - SYSTEM UTILITIES
  - APPLICATION PROGRAMS

SCI MODES OF OPERATION

TTY MODE - TELETYPE MODE

- USED FOR SCI EXECUTION ON A HARD COPY DATA TERMINAL LIKE A 733 ASR
- PRINTS EACH PROMPT AND WAITS FOR USER TO RESPOND BEFORE PRINTING THE NEXT ONE
- SQUARE BRACKET ([]) IS SCI PROMPT

VDT MODE - VIDEO DISPLAY TERMINAL MODE

- USE FOR CRT TYPE DEVICES (911, 913)
- DISPLAYS ALL PROMPTS AND POSITIONS CURSOR TO FIRST FIELD
- RETURN (NEW LINE - 913) POSITIONS CURSOR TO FIRST FIELD CHARACTER OF NEXT FIELD

VDT MODE SCI COMMAND PROMPT

T E X A S I N S T R U M E N T S

D X 1 0 S Y S T E M 3 . 1 . 0

SELECT ONE OF THE FOLLOWING COMMAND GROUPS

/DEV - DEVICE OPERATIONS

/FILE - FILE OPERATIONS

/PDEV - PROGRAM DEVELOPMENT

/SMAIN - DX10 MAINTENANCE

/SOP - DX10 OPERATION

[]

BATCH MODE - SCI COMMAND STREAM IS READ FROM
A SEQUENTIAL FILE OR A DEVICE.

- MAY BE USED FOR BATCH COMPILATIONS
AND EXECUTIONS OF HIGH LEVEL LANGUAGE
PROGRAMS

SCI ACTIVATION

FOLLOWING IS THE KEY STROKE SEQUENCE TO ACTIVATE SCI AT THE DIFFERENT STATION TYPES.

	733/743	911 VDT	913 VDT
KEY 1 -	ESC	RESET (BLANK KEY)	RESET
KEY 2 -	!	!	!

SYSTEM INITIALIZATON

AFTER BOOTING DX10, YOU SHOULD INITIALIZE THE SYSTEM.

SYSTEM LOG

THE FOLLOWING INFORMATION MAY BE AUTOMATICALLY RECORDED ON A FILE OR DEVICE:

- DEVICE HARDWARE ERRORS
- INPUT/OUTPUT ERRORS
- PROGRAM (TASK) ERRORS
- MESSAGES GENERATED BY A USER PROGRAM

INITIALIZE SYSTEM (IS) COMMAND

PURPOSE: DX10 INITIALIZATION INCLUDING:

- INITIALIZE DATE AND TIME
- INITIALIZE SYSTEM LOG
- ASSIGN NEEDED GLOBAL LUNOS

FORMAT:

[] IS

INITIALIZE SYSTEM

INITIALIZE SYSTEM LOG: YES/NO

YEAR:

MONTH:

DAY:

HOUR: (24 HOUR CLOCK)

MINUTE:

ATTENTION DEVICE:

LOGGING DEVICE: MESSAGE LOGGING DEVICE

FILES: YES/NO

THESE PROMPTS ONLY PRINTED IF 'YES' WAS ENTERED FOR 'ISL' PROMPT

DX10 MAINTENANCE

USER ID'S

A TERMINAL MAY BE 'SET UP' SO THAT A USER MUST LOG ON BEFORE SCI IS ACTIVATED.

USER ID'S ARE QUITE USEFUL BECAUSE THEY MAINTAIN THE USER'S ENVIRONMENT BETWEEN DEVELOPMENT SESSIONS (SYNONYMS)

ASSIGN USER ID (AUI) COMMAND

PURPOSE: TO ESTABLISH A NEW USER ID WHICH DX10 WILL RECOGNIZE

FORMAT:

```
[] AUI

ASSIGN USER ID

        USER DESCRIPTION: 1 - 16 CHARACTER STRING

             NEW USER ID: AAANNN - A=LETTER, N=(0-9)

            NEW PASSCODE: 1-8 CHARACTER PASSCODE

USER PRIVILEGE CODE ( 0-7 ): PRIVILEGE CODE
```

EXAMPLE:

```
[] AUI

ASSIGN USER ID

        USER DESCRIPTION: JOHN, PASCAL

             NEW USER ID: JON001

            NEW PASSCODE: CHECK1

USER PRIVILEGE CODE ( 0-7 ): 7
```

OTHER USER ID RELATED SCI COMMANDS

* MUI - MODIFY USER ID

  USED TO CHANGE THE PASSCODE AND/OR PRIVILEGE CODE
  ASSOCIATED WITH A USER ID

* DUI - DELETE USER ID

  USED TO DELETE AN EXISTING USER ID

* LUI - LIST USER ID'S

  USED TO LIST ALL EXISTING USER ID'S
  PASSCODES AND PRIVILEGE CODES ARE NOT LISTED

* MTS - MODIFY TERMINAL STATUS

  USED TO MODIFY THE 'STATUS' OF A TERMINAL
  TO REQUIRE LOGIN

MODIFY TERMINAL STATUS (MTS) COMMAND

PURPOSE: TO CHANGE THE CURRENT 'STATUS' OF A TERMINAL
(I.E. ENABLE SCI OPERATION, REQUIRE LOGIN, ETC)

FORMAT:

[] MTS

MODIFY TERMINAL STATUS

```
        TERMINAL NAME: ST05     ( STATION ID )
     TERMINAL STATUS: ON        ( ON/OFF SCI OPERATION ENABLED )
  NEW MODE (TTY/VDT): VDT       ( SCI MODE )
      LOGIN REQUIRED: YES       ( YES/NO - LOGIN FOR THIS STATION )
 USER PRIVILEGE CODE: 7         ( 0-7, PRIVILEGE CODE FOR STATION )
        DEFAULT MODE:           ( STANDARD MODE FOR STATION )
```

** NOTE: STATION STATUS CHANGES DO NOT TAKE EFFECT UNTIL A
'QUIT' (Q) COMMAND IS ENTERED.

DX10 DEVICE NAMES

** ALL DX10 SUPPORTED DEVICES HAVE 4 CHARACTER NAMES AS FOLLOWS

'HARD' DISKS

DS01 - PRIMARY DISK DRIVE
DS02 - SECONDARY DISK DRIVE
ETC.

FLOPPY DISKS

DK01
DK02
ETC.

LINE PRINTERS

LP01
LP02
ETC.

MAG TAPE DRIVES

MT01
MT02
ETC.

KEYBOARD DEVICES

ST01
ST02
ETC.

CASSETTE TRANSPORTS

CS01
CS02
ETC.

CARD READERS

CR01
CR02
ETC.

SPECIAL DEVICES

USER DEFINED
1-4 CHARACTERS

DX10 DISK FILE MANAGEMENT

FILE ORGANIZATIONS

EXPANDABLE - FILE SIZE CAN GROW AS MORE SPACE IS NEEDED

NON-EXPANDABLE - FILE SIZE IS FIXED WHEN FILE IS CREATED

FILE TYPES (ACCESS)

SEQUENTIAL - 'STANDARD' SEQUENTIAL LOGICAL DEVICE

RELATIVE RECORD - FIXED LENGTH RECORDS ARE ACCESSED BY RECORD NUMBER

MULTI-KEY INDEX - RECORDS ARE ACCESSED BY KEY

ACCESS PRIVILEGES (SPECIFIED WHEN OPENED)

READ ONLY - ONLY READ OPERATIONS ARE LEGAL

READ/WRITE - ALL OPERATIONS ARE LEGAL

EXCLUSIVE WRITE - ONLY OPENING TASK MAY WRITE TO FILE
OTHERS MAY READ

EXCLUSIVE ALL - ONLY OPENING TASK MAY PERFORM I/O TO FILE

RECORD LOCKING

A RECORD IN A FILE MAY BE 'LOCKED' WHEN AN UPDATE OPERATION IS PERFORMED TO PREVENT OTHER PROGRAMS FROM ACCESSING IT DURING THE UPDATE OPERATION.

FILE HIERARCHY

DISK VOLUMES

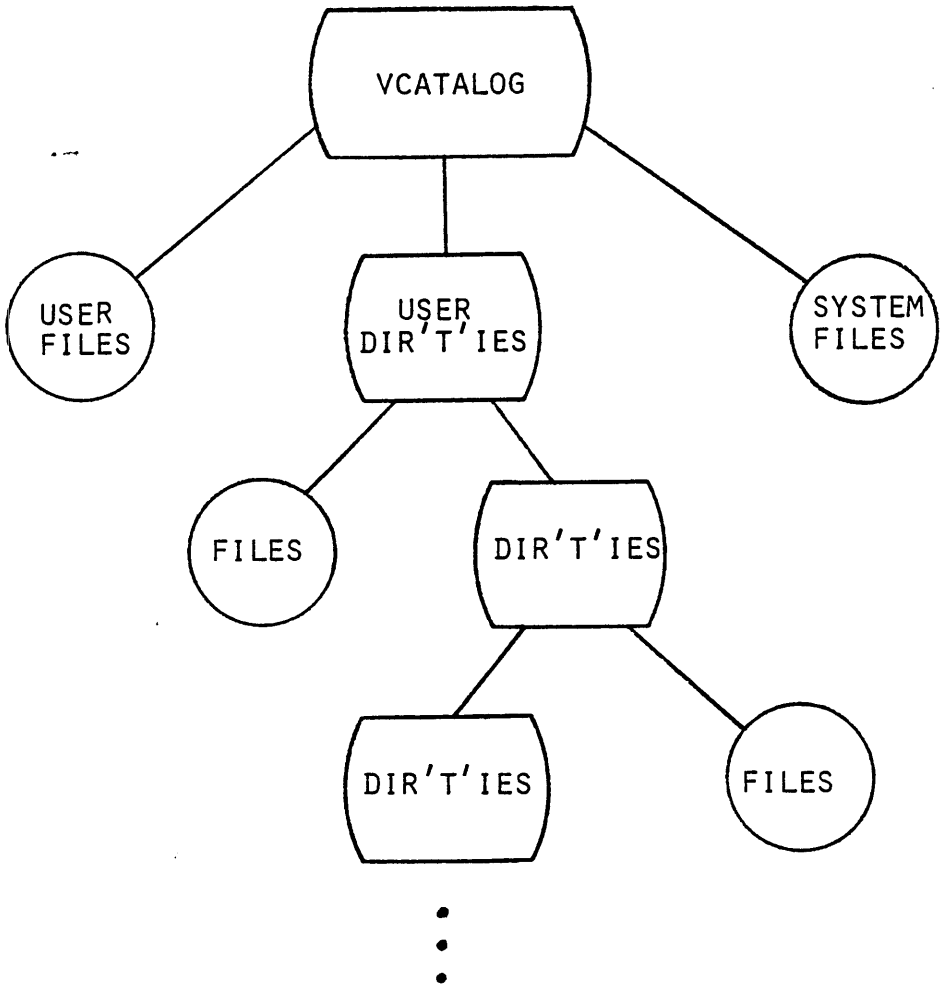
EACH DISK HAS A VOLUME NAME WHICH IS ESTABLISHED WHEN THE DISK IS INITIALIZED (SEE INSTALL NEW VOLUME COMMAND (INV) VOL. II)

EACH VOLUME HAS A CENTRAL DIRECTORY CALLED 'VCATALOG'

VCATALOG MAINTAINS INFORMATION ON:

- SYSTEM FILES
- USER DIRECTORIES
- USER FILES

DIRECTORIES AND FILES ARE MAINTAINED IN A 'TREE' STRUCTURE



FILE ACCESS PATHNAMES

* FILES ARE ACCESSED BY PATHNAME

PATHNAME FORMAT

[VOLUME NAME]
[DRIVE NAME] . <DIR NAME> ... <DIR NAME> . <FILE NAME>

** NOTE: THE DIRECTORY NAME 'VCATALOG' NEED NOT BE INCLUDED.

EXAMPLES:

DS02.VCATALOG.CLASS.SRC.MYPROG

VOL1.COBOL.LST.NEWPROG

.PASCAL.SRC.SORT

DISK ALLOCATION

ALLOCATION UNITS (ADU'S)

DISK SPACE IS ALLOCATED IN 'CHUNKS' CALLED ALLOCATION UNITS. THE SIZE OF AN ADU AND THE TOTAL NUMBER OF ADU'S ON A DISK IS DEPENDENT ON THE TYPE OF DISK BEING USED.

DISK STATISTICS

	DS31	DS10	T25	T50	T200
HEADS/DISK	2	4	5	5	19
TRACKS/DISK	406	1632	2040	4075	15485
SECTORS/TRACK	24	20	38	38	38
WORDS/SECTOR	144	144	144	144	144
SECTORS/ADU	1	1	2	3	9

DISK RELATED SCI COMMANDS

CREATE FILE DIRECTORY (CFDIR) COMMAND

PURPOSE: TO CREATE A USER FILE DIRECTORY

FORMAT:

[] CFDIR

CREATE DIRECTORY FILE

PATHNAME: COMPLETE DIRECTORY PATHNAME

MAX ENTRIES: MAXIMUM NUMBER OF FILES OR DIRECTORIES WHICH MAY BE KEPT UNDER THIS DIRECTORY

EXAMPLE:

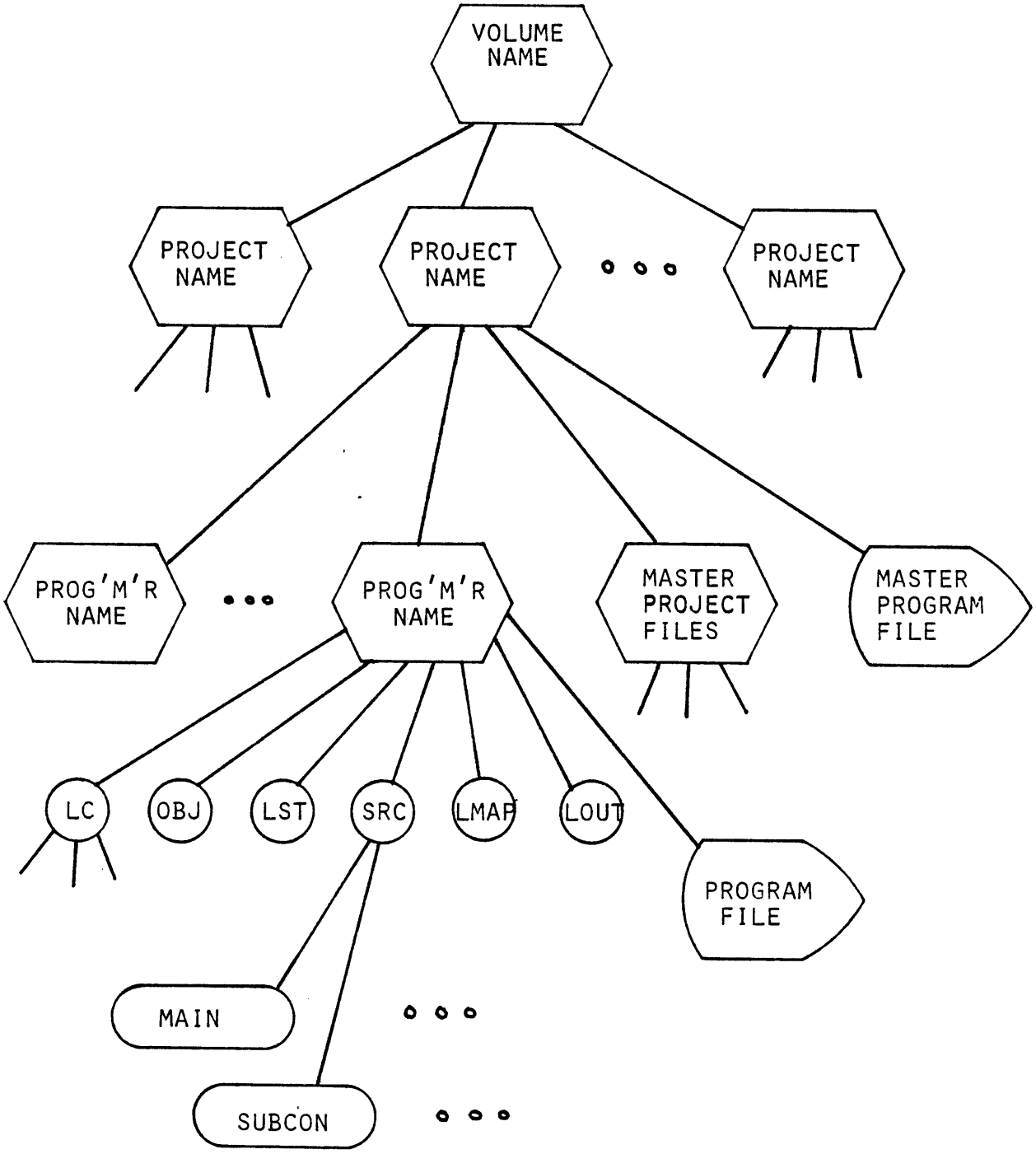
[] CFDIR

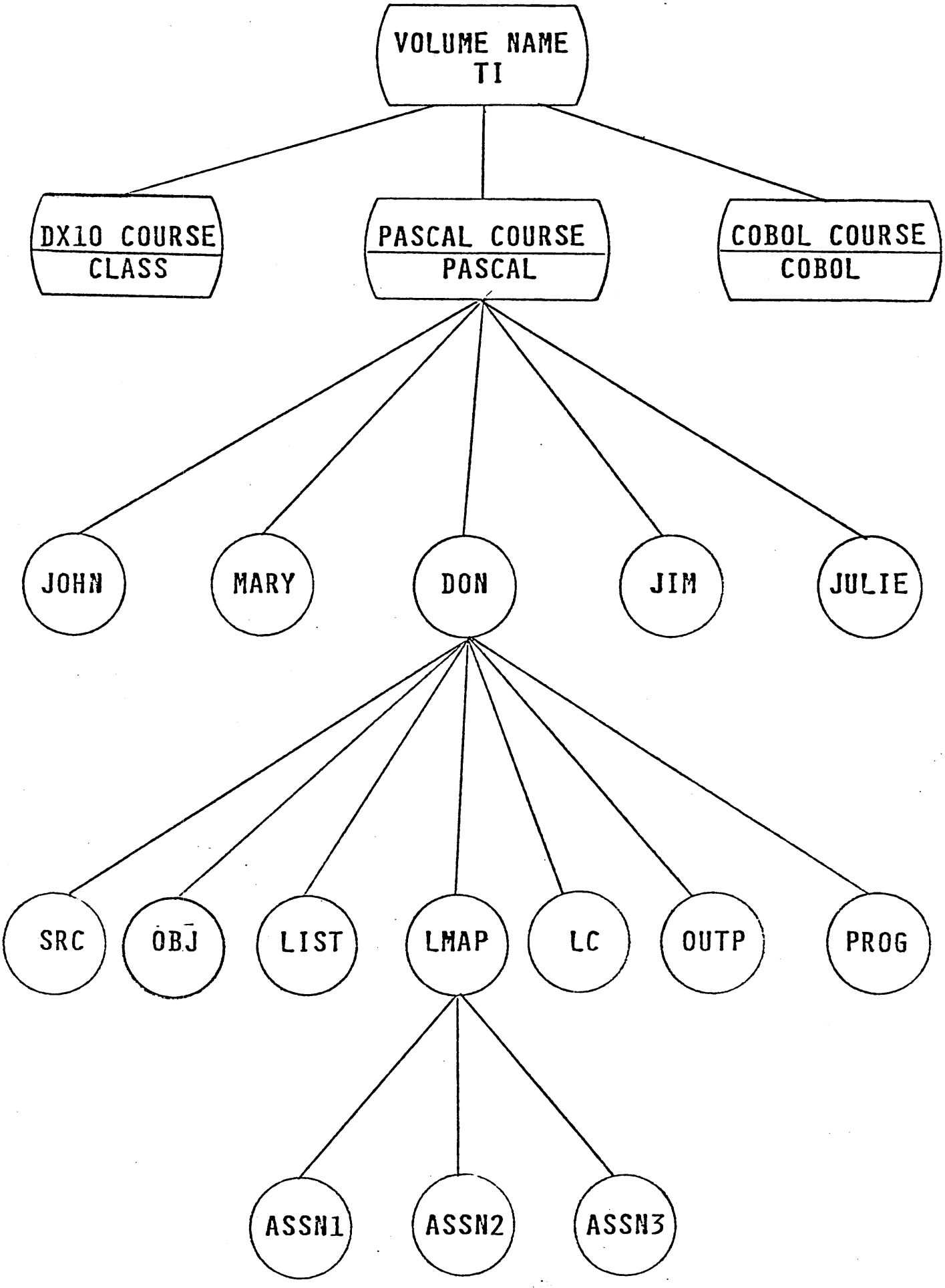
CREATE DIRECTORY FILE

PATHNAME: VOL1.PROJ1.SRC

MAX ENTRIES: 100

DIRECTORY AND FILE STRUCTURE OF A
VOLUME USED FOR DEVELOPMENT




VOLUME NAME
TI
DX10 COURSE
CLASS
PASCAL COURSE
PASCAL
COBOL COURSE
COBOL
JOHN
MARY
DON
JIM
JULIE
SRC
OBJ
LIST
LMAP
LC
OUTP
PROG
ASSN1
ASSN2
ASSN3

DISK ALLOCATION

ALLOCATION UNITS (ADU'S)

DISK SPACE IS ALLOCATED IN 'CHUNKS' CALLED ALLOCATION UNITS. THE SIZE OF AN ADU AND THE TOTAL NUMBER OF ADU'S ON A DISK IS DEPENDENT ON THE TYPE OF DISK BEING USED.

DISK STATISTICS

	DS31	DS10	T25	T50	T200
HEADS/DISK	2	4	5	5	19
TRACKS/DISK	406	1632	2040	4075	15485
SECTORS/TRACK	24	20	38	38	38
WORDS/SECTOR	144	144	144	144	144
SECTORS/ADU	1	1	2	3	9

DISK RELATED SCI COMMANDS

CREATE FILE DIRECTORY (CFDIR) COMMAND

PURPOSE: TO CREATE A USER FILE DIRECTORY

FORMAT:

[] CFDIR

CREATE DIRECTORY FILE

PATHNAME: COMPLETE DIRECTORY PATHNAME

MAX ENTRIES: MAXIMUM NUMBER OF FILES OR DIRECTORIES WHICH MAY BE KEPT UNDER THIS DIRECTORY

EXAMPLE:

[] CFDIR

CREATE DIRECTORY FILE

PATHNAME: VOL1.PROJ1.SRC

MAX ENTRIES: 100

LIST FILE DIRECTORY (LD) COMMAND

PURPOSE: TO LIST NAMES OF FILES AND SUBDIRECTORIES
IN A DIRECTORY

FORMAT:

[] LD

LIST DIRECTORY

PATHNAME: PATHNAME OF DIRECTORY TO BE LISTED

LISTING ACCESS NAME: DEVICE WHERE LISTING WILL BE DISPLAYED

EXAMPLE:

[] LD

LIST DIRECTORY

PATHNAME: LES.DX10

LISTING ACCESS NAME: LP01

```
DIRECTORY LISTING OF:  LES.DX10
  MAX # OF ENTRIES: 23    # OF ENTRIES AVAILABLE: 13

 DIRECTORY  ALIAS OF  ENTRIES    LAST UPDATE          CREATION
  DATA          *        23   02/02/78  13:14:24   02/02/78  13:13:54
  GEN           *        31   02/02/78  13:57:08   02/02/78  13:22:03
  LST           *        31   02/02/78  13:08:47   02/02/78  13:08:45
  OBJ           *        31   02/02/78  13:09:44   02/02/78  13:09:43
  SRC           *        31   02/02/78  13:57:49   02/02/78  13:09:28


 FILE        ALIAS OF  RECORDS    LAST UPDATE        FMT  TYPE   BLK PROTECT
  PKG1          *        47   02/05/78  20:11:17   BS   N SEQ  YES
  PKG2          *       106   02/14/78  08:20:52   BS   N SEQ  YES
  PKG3          *        40   02/05/78  20:33:57   BS   N SEQ  YES
  PKG4          *        41   02/06/78  16:17:20   BS   N SEQ  YES
  PKG5          *       108   03/02/78  15:50:19   BS   N SEQ  YES
10:00:20 TUESDAY, MAR 07, 1978.
```

MAP DISK (MD) COMMAND

PURPOSE: TO LIST INFORMATION ABOUT THE CONTENTS OF A DISK, VOLUME, OR DIRECTORY

FORMAT:

```
[] MD

MAP DISC

                    PATHNAME: PATHNAME OF DISK OR DIRECTORY TO MAP

       LISTING ACCESS NAME: DEVICE WHERE LISTING WILL BE PRINTED

                SHORT FORM?: YES/NO - ABBREVIATED MAP

            TOP LEVEL ONLY?: YES/NO - LIST ONLY DIRECT SONS ( DAUGHTERS )

     DIRECTORY NODES ONLY?: YES/NO - ONLY LIST DIRECTORY NODES
```

EXAMPLE:

```
[] MD

MAP DISC

                    PATHNAME: JANE

       LISTING ACCESS NAME: LP01

                SHORT FORM: YES

            TOP LEVEL ONLY?: NO

     DIRECTORY NODES ONLY?: NO
```

DISC MAP OF JANE
TODAY IS 17:13:40 TUESDAY, NOV 14, 1978.

LV	NAME	FILE TYPE	NUMBER OF RECORDS	CURRENT EOM ADU	TOTAL ALLOC ADU	LAST UPDATE
0	VCATALOG:			FILES=10	AVAILABLE=337	
	FOILS	D	54	54	54	11/14/78 17:10:36
	LISTING	D	12	12	12	11/14/78 17:10:47
	OBJ	D	4	4	4	11/14/78 17: 3: 6
	OBJECT	D	12	12	12	11/14/78 17: 2:13
	SOURCE	D	54	54	54	11/14/78 17: 3: 0
	VCATALOG	D	348	0	348	11/14/78 17:10:42
	LC	S	7	3	3	11/14/78 17: 2:29
	LMAP	S	28	3	3	11/14/78 17:10:40
	PROG	P	166	166	166	11/14/78 17: 2:27
1	FOILS:			FILES=6	AVAILABLE=47	
	REVIEW2	S	242	15	15	11/14/78 17:10:28
	REVIEW4	S	283	21	21	11/14/78 17:10:34
	REVIEW5	S	246	18	18	11/14/78 17:10:22
	REVIEW7	S	161	12	12	11/14/78 17:10:25
	SETS	S	339	21	21	11/14/78 17:10:37
	TRANS	S	88	6	6	11/14/78 17:10:31
**JANE.FOILS				TOTAL SIZE = 147 ADUS		
1	LISTING:			FILES=2	AVAILABLE=9	
	TEST	S	33	6	6	11/14/78 17:10:45
	TESTING	S	30	3	3	11/14/78 17:10:47
**JANE.LISTING				TOTAL SIZE = 21 ADUS		
1	OBJ:			FILES=1	AVAILABLE=2	
	TEST	S	17	6	6	11/14/78 17: 3: 6
**JANE.OBJ				TOTAL SIZE = 10 ADUS		
1	OBJECT:			FILES=1	AVAILABLE=10	
	TESTINT	S	43	15	15	11/14/78 17: 2:14
**JANE.OBJECT				TOTAL SIZE = 27 ADUS		
1	SOURCE:			FILES=10	AVAILABLE=43	
	ASSN1A	S	27	3	3	11/14/78 17: 2:55
	ASSN1B	S	43	6	6	11/14/78 17: 2:52
	ASSN1C	S	51	6	6	11/14/78 17: 2:50
	ASSN2A	S	64	9	9	11/14/78 17: 3: 1
	ASSN2B	S	82	12	12	11/14/78 17: 2:47
	ASSN2C	S	143	18	18	11/14/78 17: 2:58
	ASSN2D	S	188	24	24	11/14/78 17: 2:45
	ASSN3A	S	111	15	15	11/14/78 17: 2:35
	ASSN3B	S	118	15	15	11/14/78 17: 2:41
	ASSN3C	S	121	15	15	11/14/78 17: 2:38
**JANE.SOURCE				TOTAL SIZE = 177 ADUS		
**JANE				TOTAL SIZE = 2594 ADUS		

INSTALL VOLUME (IV) COMMAND

PURPOSE: TO INSTALL AN 'OLD' (ONE WHICH HAS GOOD DATA ON IT) VOLUME FOR USAGE UNDER DX10

FORMAT:

[] IV

INSTALL VOLUME

UNIT NAME: DS03

VOLUME NAME: JANE

**NOTE - IF YOU WISH TO INSTALL A 'NEW' (ONE WHICH HAS NO GOOD DATA ON IT) VOLUME, YOU SHOULD USE THE INSTALL NEW VOLUME (INV) COMMAND.

**NOTE - IF A VOLUME IS INSTALLED AND YOU WISH TO INSTALL ANOTHER VOLUME IN THE SAME DRIVE, YOU MUST UNLOAD THE CURRENT VOLUME USING THE UNLOAD VOLUME (UV) COMMAND.

UNLOAD VOLUME (UV) COMMAND

PURPOSE: TO UNLOAD A PREVIOUSLY INSTALLED VOLUME

FORMAT:

[] UV

UNLOAD VOLUME

VOLUME NAME: JANE

SYNONYMS

** USED TO SAVE AN OPERATOR KEYSTROKES

** ALLOWS ONE STRING TO REPLACE ANOTHER STRING

** SYNONYM RELATED SCI COMMANDS

- ASSIGN SYNONYM (AS)
- LIST SYNONYMS (LS)
- MODIFY SYNONYMS (MS)

**NOTE - SYNONYMS ARE LOCAL TO THE TERMINAL AT WHICH THEY ARE BEING USED

**NOTE - WHEN A USER ID IS IN USE AND SYNONYMS ARE ASSIGNED, THOSE SYNONYMS GET SAVED SO THAT THE NEXT TIME THAT USER LOGS ON, HIS SYNONYMS WILL STILL BE ASSIGNED

ASSIGN SYNONYM (AS) COMMAND

PURPOSE: TO DEFINE A STRING TO SUBSTITUTE FOR ANOTHER STRING

FORMAT:

```
[] AS

ASSIGN SYNONYM VALUE

          SYNONYM: REPLACEMENT STRING

            VALUE: REPLACED STRING
```

EXAMPLE:

```
[] AS

ASSIGN SYNONYM VALUE

          SYNONYM: L

            VALUE: LES.PASCAL.SRC
```

*** NOW

```
L.PROG1 = LES.PASCAL.SRC.PROG1
```

PROGRAMS UNDER DX10 (TASKS)

PROGRAMS WHICH ARE GOING TO RUN UNDER DX10 MUST BE INSTALLED ON A PROGRAM FILE BEFORE THEY MAY BE EXECUTED.

ANY PROGRAM WHICH IS INSTALLED ON A PROGRAM FILE WILL HAVE A 'TASK ID'. TASKS MAY BE INSTALLED ON A PROGRAM FILE AND ASSIGNED A 'TASK ID' BY THE LINKAGE EDITOR AUTOMATICALLY AT LINK TIME.

EXECUTING TASKS UNDER DX10

TASKS MAY BE EXECUTED IN MANY DIFFERENT WAYS UNDER DX10.

(XT) EXECUTE TASK
(XHT) EXECUTE AND HALT TASK
(XTS) EXECUTE AND TERMINATE SCI
(XCTF) EXECUTE COBOL TASK FOREGROUND
(XPT) EXECUTE PASCAL TASK

ETC.

MODES OF EXECUTION

ANY TASK WHICH IS EXECUTED FROM A STATION MAY EXECUTE IN ONE OF TWO MODES:

FOREGROUND - THIS MODE IS FOR AN INTERACTIVE TASK WHICH WILL PERFORM I/O TO THE STATION FROM WHICH IT WAS EXECUTED.

- SCI IS SUSPENDED UNTIL THE TASK COMPLETES EXECUTION.

BACKGROUND - THIS MODE IS FOR A TASK WHICH REQUIRES NO ON INTERACTION WITH THE STATION FROM WHICH IT WAS EXECUTED.

- SCI IS NOT SUSPENDED

- CONTROL RETURNS TO SCI AS SOON AS THE TASK SPECIFIED IS PLACED IN EXECUTION.

** NOTE: ONLY ONE FOREGROUND AND ONE BACKGROUND TASK MAY BE PLACED IN EXECUTION FROM A STATION AT ANY GIVEN TIME.

SUMMARY

SYSTEM COMMAND INTERPRETER (SCI)

- 3 MODES OF OPERATION
  - TTY MODE
  - VDT MODE
  - BATCH MODE

DX10 MAINTENANCE

- USER ID'S
  - MAINTAIN DEVELOPMENT ENVIRONMENT (SYNONYMS)
  - MAY RESTRICT SCI COMMANDS WHICH MAY BE USED
  - PROVIDE SOME SYSTEM SECURITY (LOG ON)

DX10 FILE MANAGEMENT

- 3 TYPES OF FILES SUPPORTED
  - SEQUENTIAL
  - RELATIVE RECORD (RANDOM ACCESS)
  - MULTI-KEY INDEX (KEY ACCESS)
- HIERARCHY DIRECTORY AND FILE STRUCTURE
- SCI COMMANDS SUPPORT DIRECTORY STRUCTURE
  - CREATE DIRECTORY
  - LIST DIRECTORY
  - MAP DISK
    ETC.

USER PROGRAMS (TASKS)

- TWO MODES OF OPERATION
  - FOREGROUND (FOR INTERACTIVE PROGRAMS)
  - BACKGROUND (FOR NON-INTERACTIVE PROGRAMS)
  - A USER MAY EXECUTE ONE OF EACH AT A STATION AT A TIME

PASCAL PROGRAM DEVELOPMENT

OBJECTIVES

STUDENTS SHOULD BE ABLE TO USE THE TEXT EDITOR TO CREATE AND EDIT PASCAL SOURCE PROGRAMS.

STUDENTS SHOULD BE ABLE TO COMPILE, LINK, INSTALL, AND EXECUTE PASCAL PROGRAMS.

AGENDA

1. PASCAL COMPILER
   - XTIP COMMAND
   - STACK AND HEAP MEMORY SPECIFICATIONS
   - ERROR MESSAGES
2. PROGRAM LINKING
   - XLE COMMAND
   - LINK CONTROL FILES
   - EXAMPLE
3. PROGRAM EXECUTION
   - XPT COMMAND
   - TASK MANAGEMENT SCI COMMANDS
4. TEXT EDITOR

PASCAL - SOURCE TO EXECUTION

3 MAJOR STEPS

1. COMPILATION (XTIP)
2. LINKING (XLE)
3. EXECUTION (XPT)

PASCAL COMPILER

The TEXAS INSTRUMENTS PASCAL (TIP) Compiler has 3 phases of execution

PHASE 1 (SILT1) - Initial SYNTACTIC scan of source program

PHASE 2 (SILT2) - Translation from source to intermediate language

PHASE 3 (CODEGEN) - Generation of 990 object from intermediate LANGUAGE

NOTE: No phase will execute if errors were found in the previous phase phase.

EXECUTION OF TIP COMPILER

[] XTIP

EXECUTE TI PASCAL COMPILER

SOURCE: Pathname of PASCAL source

OBJECT: Pathname of PASCAL compiled object

LISTING: Pathname for compiled listing

MESSAGES: Pathname where Compiler messages are written

MEM1: S,H Stack, Heap memory for SILT1

MEM2: S,H Stack, Heap memory for SILT2

MEM3: S,H Stack, Heap memory for CODEGEN

MODE: FOREGROUND or BACKGROUND - Compiler execution mode

EXAMPLE

```
[] XTIP

EXECUTE TI PASCAL COMPILER

             SOURCE: TI.PASCAL.SRC.MYPROG

             OBJECT: TI.PASCAL.OBJ.MYPROG

            LISTING: TI.PASCAL.LST.MYPROG

           MESSAGES: ME

               MEM1:

               MEM2:

               MEM3:

               MODE: FOREGROUND

== FOREGROUND COMMAND EXECUTING ==



COMPILER MESSAGES
-----------------


SILT1      EXECUTION BEGINS
MYPROG
NORMAL TERMINATION
STACK USED =   3710  HEAP USED =  1960


SILT2      EXECUTION BEGINS
MYPROG
NO ERRORS IN PROGRAM
NORMAL TERMINATION
STACK USED =  10298  HEAP USED =  1232


CODEGEN    EXECUTION BEGINS
MYPROG
NORMAL TERMINATION
STACK USED =   8430  HEAP USED =  2684
```

STACK AND HEAP REQUIREMENTS FOR TIP

THE TIP COMPILER IS WRITTEN IN TIP. MEMORY FOR VARIABLES USED IN A TIP PROGRAM IS ALLOCATED IN ONE OF TWO WAYS: STACK OR HEAP.

STACK MEMORY

THIS MEMORY IS ALLOCATED FOR 'STATIC' VARIABLES IN A LAST-IN - FIRST-OUT FASHION (THE STACK MEMORY REQUIRED BY A PASCAL PROGRAM IS DEFINED AT COMPILE TIME)

HEAP MEMORY

THIS MEMORY IS ALLOCATED FOR 'DYNAMIC' VARIABLES WHICH ARE CREATED AND DELETED USING THE 'NEW' AND 'DISPOSE' STANDARD FUNCTION CALLS (THE HEAP MEMORY REQUIRED FOR 'DYNAMIC' VARIABLES IS ONLY DEFINED AT RUN TIME)

** NOTE - BOTH OF THESE MEMORY ALLOCATION STRATEGIES WILL BE DISCUSSED LATER IN THE COURSE.

TIP REQUIREMENTS

THE USER MAY SPECIFY THE AMOUNT OF MEMORY TO BE USED FOR STACK AND HEAP MEMORY BY EACH PHASE OF TIP COMPILER WHEN IT IS EXECUTED. IF THE SPECIFICATION IS NOT MADE, A DEFAULT VALUE IS USED.

THESE SPECIFICATIONS ARE MADE IN " K'S OF BYTES ".

** SEE NEXT PAGE FOR TABLE OF STACK AND HEAP VALUES FOR TIP **

123

STACK AND HEAP VALUES FOR TIP COMPILER

PHASE	STACK			HEAP		
	MIN	MAX	DEFAULT	MIN	MAX	DEFAULT
MEM1 (SILT1)	6K	6K	6K	4K	30K	10K
MEM2 (SILT2)	12K	13K	13K	2K	7K	4K
MEM3 (CODEGEN)	10K	10K	10K	4K	10K	8K

** NOTES **

1) THE DEFAULT STACK AND HEAP VALUES ARE ADEQUATE FOR MOST 'AVERAGE SIZE' PROGRAM.

2) IF ROUTINES ARE VERY DEEPLY 'NESTED' WITHIN OTHER ROUTINES, THEN MORE STACK SPACE MAY BE REQUIRED.

3) ROUTINES CONTAINING VERY LONG EXPRESSIONS MAY REQUIRE MORE STACK SPACE.

4) IF A PROGRAM USES A LARGE NUMBER OF IDENTIFIERS, MORE HEAP SPACE MAY BE REQUIRED.

5) VERY LONG ROUTINES OR ONES THAT USE A LARGE NUMBER OF CONSTANTS MAY REQUIRE MORE HEAP SPACE.

124

EXAMPLES OF TIP COMPILER ERROR MESSAGES

```
DXPSCL        1.5.0  78.317   TI 990 PASCAL COMPILER   01/05/79 13:15:36

   PROGRAM ERRORS;

   VAR B , INPUT : INTEGER;
                !2
       C          : ARRAY [ 1..20 ] OF REAL;
       TEST       : BOOLEAN;

   BEGIN (* ERRORS *)
       A := 47;
          !104
  **** ERROR # 105 ****
  **** ERROR # 145 ****
       B := 3.48;
  **** ERROR # 145 ****
       FOR J = 3 TO 20 DO
             !51
           BEGIN
               C [ J ] := 0.0;
               WRITELN(' PRINT A AMESSAGE ')
           END;
       INPUT := 21;
           !43   !5
  **** ERROR # 167 ****
       TEST := TRUE;
       WHILE TEST DO
           BEGIN
               B := B + 1;
               IF B > 17 THEN
                   TEST := FALSE;
           END
       WRITELN( ' END OF A BAD PROGRAM!!!')
             !14
   END.


  MAXIMUM NUMBER OF IDENTIFIERS USED = 6

  NUMBER OF ERRORS = 10
     2  E  IDENTIFIER EXPECTED
     5  E  ':' EXPECTED
    14  E  ';' EXPECTED
    43  E  STATEMENT EXPECTED
    51  E  ':=' EXPECTED
   104  E  UNDECLARED IDENTIFIER
   105  F  CLASS OF IDENTIFER IS NOT VARIABLE
   145  F  TYPE CONFLICT IN ASSIGNMENT
   167  F  UNDECLARED LABEL
```

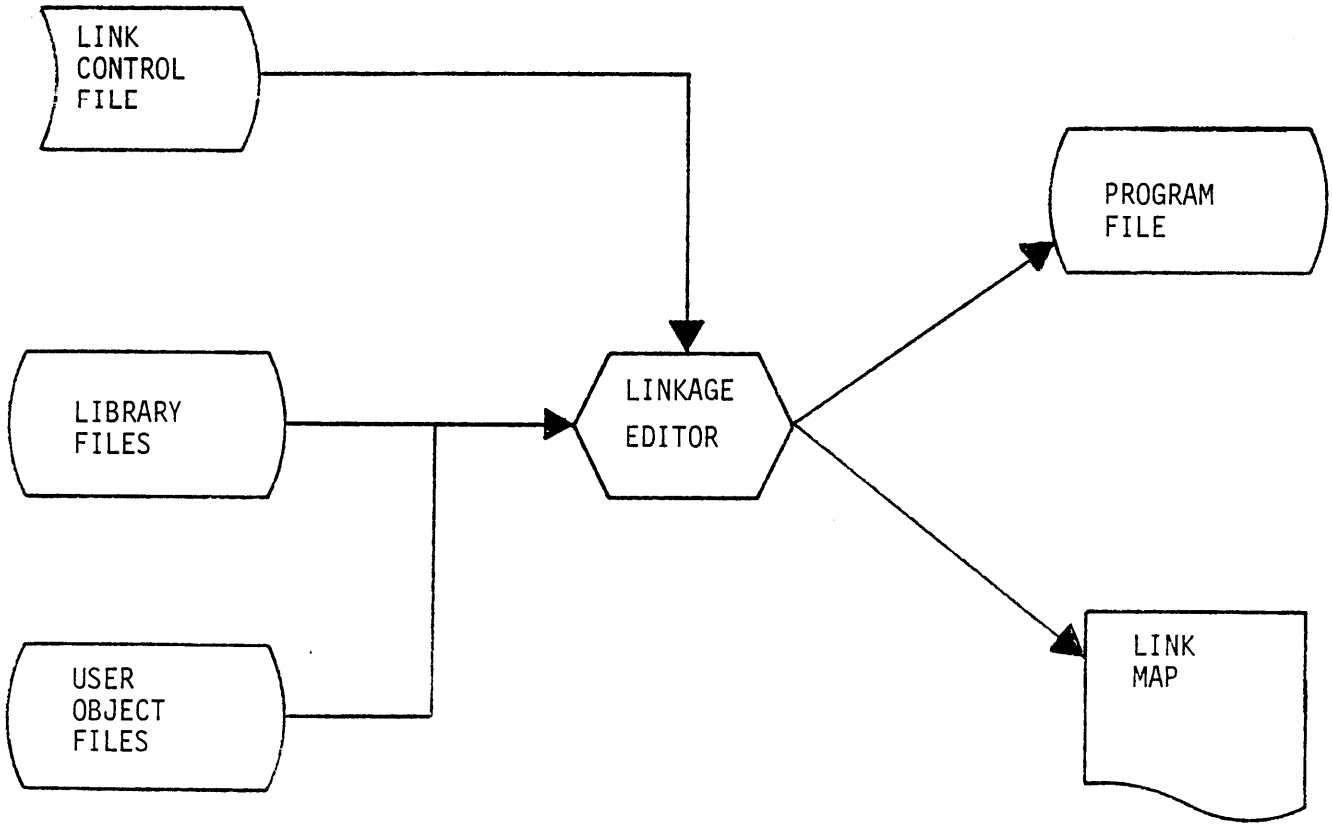
LINKING A PROGRAM

BEFORE AN ERROR FREE PROGRAM MAY BE EXECUTED, IT MUST BE LINKED TO THE NEEDED RUNTIME LIBRARIES AND INSTALLED ON A PROGRAM FILE AS A TASK.

LINK EDITOR

THE LINK EDITOR IS A UTILITY WHICH RESOLVES REFERENCES BETWEEN INDEPENDENTLY COMPILED OR ASSEMBLED MODULES.

LINK EDITOR OPERATION



126

LINK EDITOR

LINK CONTROL FILE (INPUT)
- SPECIFIES TO LINK EDITOR THE LIBRARY FILES AND USER OBJECT FILES WHICH ARE TO BE LINKED INTO A COMPLETE PROGRAM

LIBRARIES (INPUT)
- FILE DIRECTORIES WHICH CONTAIN THE RUNTIME ROUTINES WHICH MAY BE UTILIZED BY A PROGRAM. (ONLY THOSE USED BY A PROGRAM ARE LINKED IN)

USER OBJECT (INPUT)
- OBJECT FILE(S) WHICH CONTAIN THE COMPILED OR ASSEMBLED OBJECT FOR A USER PROGRAM. (CONSTRUCTED BY A COMPILER OR ASSEMBLER)

PROGRAM FILE (OUTPUT)
- ANY PROGRAM (TASK) WHICH IS GOING TO RUN UNDER DX10, MUST BE INSTALLED ON A SPECIAL TYPE OF FILE CALLED A 'PROGRAM FILE' BEFORE IT CAN BE EXECUTED.
  WHEN EXECUTING THE LINK EDITOR, YOU MAY HAVE THE LINKED OBJECT AUTOMATICALLY INSTALLED ON A PROGRAM FILE FOR EXECUTION.

LINK MAP (OUTPUT)
- LISTING PRODUCED BY THE LINKAGE EDITOR WHICH LISTS ALL THE LINKING INFORMATION NEEDED BY THE USER.

EXECUTION OF THE LINK EDITOR

```
[] XLE

EXECUTE LINKAGE EDITOR

      CONTROL ACCESS NAME: <PATHNAME OF LINK CONTROL FILE>

LINKED OUTPUT ACCESS NAME: <PROGRAM FILE PATHNAME>

      LISTING ACCESS NAME: <PATHNAME TO WHICH LINK MAP IS WRITTEN>

              PRINT WIDTH: 80
```

LINK CONTROL FILES (COMMAND STREAM FOR LINK EDITOR)

PURPOSE:

* SPECIFIES OBJECT FILES AND LIBRARIES WHICH ARE TO BE LINKED
* ASSIGNS A TASK NAME TO THE PROGRAM BEING LINKED
* SPECIFIES PRIORITY OF THE PROGRAM BEING LINKED
* CREATED BY USING THE TEXT EDITOR

LINK CONTROL COMMANDS

TASK COMMAND

PURPOSE: ASSIGNS A NAME TO A TASK BEING LINKED

FORMAT:

```
TASK  < NAME >
```

WHERE, < NAME > - 1 TO 8 CHARACTER NAME FOR PROGRAM

EXAMPLE:

```
TASK  MYPROG
```

INCLUDE COMMAND

PURPOSE: SPECIFIES FILE NAME OF OBJECT MODULE TO BE INCLUDED IN THE LINKED OUTPUT MODULE.

FORMAT:

```
INCLUDE  < ASSEMBLED OR COMPILED OBJECT FILE PATHNAME >
```

EXAMPLE:

```
INCLUDE  TI.OBJECT.MYPROG
```

**NOTE: IF EXTENDED PRECISION REAL NUMBERS ARE DESIRED, TWO ADDITIONAL 'INCLUDE' COMMANDS MUST FOLLOW THE 'INCLUDE (MAIN)' COMMAND IN THE LINK CONTROL FILE. THESE TWO COMMANDS ARE:

```
INCLUDE (FL$ITD)
INCLUDE (TEN$D)
```

LIBRARY COMMAND

PURPOSE: SPECIFIES DIRECTORIES WHICH ARE TO BE SEARCHED WHEN AN ABBREVIATED INCLUDE COMMAND IS USED.

FORMAT:

```
LIBRARY  < DIRECTORY NAME > [ < DIRECTORY NAME > ... ]
```

EXAMPLE:

```
     LIBRARY   TI.OBJECT  --I
               .            I
               .            I < == >  INCLUDE TI.OBJECT.MYPROG
               .            I
     INCLUDE   ( MYPROG ) --I
```

**NOTE: LIBRARIES ARE SEARCHED IN THE ORDER IN WHICH THEY ARE SPECIFIED.

FORMAT COMMAND

PURPOSE: SPECIFIES THE FORMAT THAT THE LINKED OBJECT MODULE SHOULD BE IN.

FORMAT:

```
                 IMAGE
FORMAT           ASCII          [ ,REPLACE ]      ,PRIORITY
                 COMPRESSED
```

IMAGE - OUTPUT WILL GO TO A PROGRAM FILE

ASCII - OUTPUT WILL GO TO A STANDARD OBJECT FILE IN ASCII FORMAT

COMPRESSED - OUTPUT WILL GO TO A STANDARD OBJECT FILE IN A SPECIAL 'COMPRESSED' FORMAT

EXAMPLE:

```
FORMAT   IMAGE,REPLACE,2
```

NOSYMT COMMAND

PURPOSE: SPECIFIES THAT NO SYMBOL TABLE IS TO BE INCLUDED IN THE LINKED OUTPUT. (THIS COMMAND SHOULD BE USED WHEN THE 'FORMAT IMAGE' COMMAND IS USED)

FORMAT:

NOSYMT

END COMMAND

PURPOSE: TERMINATES THE LINK CONTROL FILE INPUT STREAM TO THE LINK EDITOR (EVERY LINK CONTROL FILE MUST CONTAIN AN END COMMAND)

FORMAT:

END

SAMPLE PASCAL LINK CONTROL FILE

```
NOSYMT

LIBRARY  .TIP.OBJ

FORMAT  IMAGE,REPLACE,3

TASK  MYPROG

      INCLUDE  (MAIN)

      INCLUDE  TI.PASCAL.OBJ.MYPROG

END
```

NOTES:

* COLUMN POSITION IS NOT IMPORTANT

* ' (MAIN) ' IS THE PASCAL RUNTIME PACKAGE WHICH MUST BE INCLUDED IN EVERY PASCAL LINK.

* THE 'INCLUDE' COMMAND FOR ' (MAIN) ' MUST BE THE FIRST 'INCLUDE' COMMAND IN THE LINK STREAM

PASCAL LINK EXAMPLE

```
[] XLE

EXECUTE LINKAGE EDITOR
        CONTROL ACCESS NAME: TI.PASCAL.LC.MYPROG
 LINKED OUTPUT ACCESS NAME: TI.PASCAL.PROG
        LISTING ACCESS NAME: TI.PASCAL.LMAP.MYPROG
                PRINT WIDTH: 80
```

SAMPLE LINK MAP

```
TI 990/10 SDSLNK 939187 *A    11/15/78  08:31:22                    PAGE    1
COMMAND LIST
NOSYMT
LIBRARY .TIP.OBJ
FORMAT IMAGE,REPLACE,3
TASK MYPROG
INCLUDE (MAIN)
INCLUDE TI.PASCAL.OBJ.MYPROG
END

TI 990/10 SDSLNK 939187 *A    11/15/78  08:31:22                    PAGE    2
LINK MAP
CONTROL FILE = TI.PASCAL.LC.MYPROG

LINKED OUTPUT FILE = TI.PASCAL.PROG

LIST FILE = TI.PASCAL.LMAP.MYPROG

NUMBER OF OUTPUT RECORDS = 84

OUTPUT FORMAT = IMAGE
```

PHASE 0, MYPROG ORIGIN = 0000 LENGTH = 5810 (TASK ID = 1)

MODULE	NO	ORIGIN	LENGTH	TYPE	DATE	TIME	CREAEAT
MAIN	1	0000	39DA	INCLUDE	03/09/78	21:38:24	SDSLSLN
$DATA	1	4E62	0034				
SUM_IT	2	39DA	0020	INCLUDE	11/15/78	08:27:45	DXPSPSC
READ_ECH	3	39FA	0090	INCLUDE	11/15/78	08:27:50	DXPSPSC
DUMP_IT	4	3A8A	01F4	INCLUDE	11/15/78	08:28:07	DXPSCL
MSG$	5	3C7E	00DE	LIBRARY	03/08/78	16:57:56	DXPSCL
SCIRTNS	6	3D5C	0726	LIBRARY	03/09/78	21:44:36	SDSLNK
$DATA	6	4E96	041E				
P$TERM	7	4482	0058	LIBRARY	03/08/78	19:14:34	SDSMAC
INIT$1	8	44DA	0060	LIBRARY	03/08/78	16:56:46	DXPSCL
P$INIT	9	453A	001C	LIBRARY	03/08/78	19:13:56	SDSMAC
PB$INIT	10	4556	0002	LIBRARY	03/08/78	19:14:48	SDSMAC
RDI$T	11	4558	0158	LIBRARY	03/08/78	18:23:30	DXPSCL
REST$T	12	46B0	0050	LIBRARY	03/08/78	17:41:13	DXPSCL
EOF$	13	4700	0009	LIBRARY	03/08/78	19:04:44	SDSMAC
RDLN$	14	470A	00E4	LIBRARY	03/08/78	17:39:22	DXPSCL
ASSRT$	15	47EE	000A	LIBRARY	03/08/78	18:54:24	SDSMAC
GET$CH	16	47F8	002A	LIBRARY	03/08/78	19:11:17	SDSMAC
DEI$T	17	4822	0268	LIBRARY	03/08/78	18:12:08	DXPSCL
GET$RCOR	18	4A8A	00CA	LIBRARY	03/08/78	17:33:56	DXPSCL
ASSER$	19	4B54	00C4	LIBRARY	03/08/78	16:46:33	DXPSCL
DEX$T	20	4C18	024A	LIBRARY	03/08/78	18:15:03	DXPSCL

COMMON	NO	ORIGIN	LENGTH
CUR$	1	52B4	0002
PARM$	7	52B6	000A
HEAP$	1	52C0	000E
SYS$MS	5	52CE	0020
MEM$	1	52EE	0522

D E F I N I T I O N S

NAME	VALUE	NO	NAME	VALUE	NO	NAME	VALUE	NO	NAME	VALUE	NO
*ABEND$	005E	1	*ABND$1	0064	1	ASSER$	4B8A	19	ASSRT$	47EE	15
*CLOSE$	014E	1	CLS$	01A8	1	*CLS$FI	0210	1	*CMP$ST	0242	1
*CREAT$	0262	1	DEI$T	484A	17	DEX$T	4C38	20	*DIV$	03A8	1
*DSTR$$	0408	1	*DSTRY$	03F8	1	*DUMP$H	044E	1	*DUMP$P	065E	1
*DUMP$S	08FA	1	*ENC$T	0C20	1	*ENI$T	0CE0	1	*ENS$T	0E40	1
*ENT$	008C	1	ENT$1	0070	1	ENT$2	007C	1	ENT$M	0096	1
ENT$S	0114	1	*ENX$T	0F40	1	EOF$	4700	13	*EOF$WR	10EE	1
EOLN$	110E	1	*FIND$S	3254	1	FL$INI	12B6	1	*FREE$	116C	1
GET$CH	47F8	16	*GET$ME	1380	1	*GET$PA	13DA	1	GET$RC	4AA2	1
*GET$TC	321E	1	*GO$	148A	1	HALT$	14F6	1	*HEAP$T	153E	1
*INIT$	1598	1	INIT$1	44F6	8	*INIT$D	16FE	1	IO$ERR	186C	1
*MAP$	1BC6	1	MOV$4	1D9A	1	*MOV$5	1D98	1	*MOV$6	1D96	1
*MOV$7	1D94	1	*MOV$8	1D92	1	*MOV$N	1D8C	1	MSG$	3C92	5

NAME	VALUE	NO	NAME	VALUE	NO	NAME	VALUE	NO	NAME	VALUE	NO
*NEW$	1DBA	1	OPEN$	209A	1	*OPN$FI	2202	1	P$INIT	454A	9
*P$MAIN	0008	1	P$TERM	449E	7	*PATCH$	00AA	1	PB$INI	4556	1C
PB$TER	4556	10	*PRT$ME	2410	1	PSCL$$	3AFC	4	*PUT$RC	2740	1
*PUTCH$	26FC	1	RDI$T	4574	11	RDLN$	4724	14	READ_E	3A22	3
*RESM$$	0048	1	REST$T	46BE	12	*RESUM$	28A4	1	RET$1	013C	1
RET$2	0128	1	RET$M	0124	1	RET$S	0126	1	REWND$	2810	1
REWRT$	284A	1	*RSUMR$	28F6	1	*RWIND$	291A	1	S$GTCA	43B4	6
*S$IADD	3D5C	6	*S$IASC	3DD0	6	*S$IDIV	3E72	6	*S$IMUL	3EB2	6
S$INT	3F6C	6	*S$ISUB	3D84	6	S$MAPS	4104	6	*S$NAME	2966	1
*S$NEW	4196	6	S$PARM	4222	6	S$PTCA	43FC	6	*S$RTCA	443A	6
*S$SCPY	424C	6	S$SETS	42CA	6	*SCB$FR	29D0	1	*SCB$IN	2A3C	1
SET$AC	2AC2	1	*SET$NA	2B84	1	*STACK$	2BAE	1	*STORE$	322C	1
SUM_IT	39E8	2	SVC$	2BC8	1	*T$VEC	0000	1	*TERM$	2E02	1
TX$ERR	335A	1	*WRC$T	3514	1	*WREOF$	360A	1	WRI$T	366C	1
WRLN$	3768	1	WRS$T	37EC	1	*WRX$T	38EE	1			

**** LINKING COMPLETED

EXECUTION OF A PASCAL PROGRAM

```
[]XPT

EXECUTE TI PASCAL TASK

        PROGRAM FILE: PROGRAM FILE PATHNAME

     TASK NAME OR ID: PROGRAM NAME ( SEE 'TASK' COMMAND )

               INPUT: PATHNAME FOR PASCAL 'INPUT' FILE

              OUTPUT: PATHNAME FOR PASCAL 'OUTPUT' FILE

            MESSAGES: LIST FILE PATHNAME FOR EXECUTION MESSAGES

                MODE: FOREGROUND OR BACKGROUND

              MEMORY: STACK AND HEAP MEMORY FOR EXECUTING PROGRAM
```

EXAMPLE

```
[]XPT

EXECUTE TI PASCAL TASK
        PROGRAM FILE: TI.PASCAL.PROG
     TASK NAME OR ID: MYPROG
               INPUT: .TESTDATA
              OUTPUT: LP01
            MESSAGES: ME
                MODE: FOREGROUND
              MEMORY:

== FOREGROUND COMMAND EXECUTING ==


MYPROG    EXECUTION BEGINS
NORMAL TERMINATION
STACK USED =   618  HEAP USED =   468
EXECUTION OF PASCAL TASK IS COMPLETE.:
```

TASK RELATED SCI COMMANDS

SHOW TASK STATUS (STS) COMMAND

PURPOSE: TO DISPLAY THE STATUS OF ONE OR ALL TASKS CURRENTLY RUNNING UNDER DX10.

FORMAT:

[] STS

SHOW TASK STATUS

INSTALLED ID: < PROGRAM FILE TASK ID. >

OUTPUT ACCESS NAME: < DEVICE OR FILE FOR STATUS LISTING >

EXAMPLE:

[] STS

SHOW TASK STATUS

INSTALLED ID: (NO ENTRY ==> ALL TASKS)

OUTPUT ACCESS NAME: (NO ENTRY ==> STATION YOU ARE AT)

BID ID	RUN ID	STATION	STATE	PRIORITY	FLAG1	FLAG2	WP	PC
20	65	2	09	01	5401	0120	48B8	2EF6
20	2C	3	17	01	5000	0100	48B8	6078
23	23		05	01	6000	0000	0616	06B4

KILL TASK (KT) COMMAND

PURPOSE: TO KILL AN ACTIVE TASK

FORMAT:

```
[] KT

KILL TASK

            RUN ID: < RUNTIME ID OF TASK - SEE STS PRINTOUT >

   STATION NUMBER: < STATION WHERE TASK WAS EXECUTED - SEE STS >
```

DELETE TASK (DT) COMMAND

PURPOSE: TO DELETE A TASK FROM A PROGRAM FILE

FORMAT:

```
[] DT

DELETE TASK

  PROGRAM FILE OR LUNO: < NAME OF PROGRAM FILE >

       TASK NAME OR ID: < INSTALLED ID OR NAME >
```

EXAMPLE:

```
[] DT

DELETE TASK

   PROGRAM FILE OR LUNO: TI.PASCAL.PROG

        TASK NAME OR ID: MYPROG
```

**NOTE: A TASK MUST BE DELETED BEFORE A NEW TASK WITH THE SAME NAME OR ID MAY BE INSTALLED.

IF THE 'FORMAT IMAGE,REPLACE' LINK CONTROL COMMAND IS USED, THE OLD TASK IS AUTOMATICALLY DELETED AND THE NEW ONE INSTALLED IN IT'S PLACE.

TEXT EDITOR

(REF - DX10 REL. 3.0 MANUAL - SEC 2 OF VOL IV)

INTRODUCTION

THE TEXT EDITOR ALLOWS THE USER TO INTERACTIVELY CREATE AND MODIFY SOURCE PROGRAMS ON DISK FILES. THE TEXT EDITOR MAY BE USED TO CREATE AND MODIFY OTHER TYPES OF FILES AS WELL.

THE TEXT EDITOR MAY BE EXECUTED FROM THE FOLLOWING PERIPHERALS:

- 911 VDT
- 913 VDT
- 733 ASR OR KSR
- 743 KSR

STEPS IN EXECUTING THE TEXT EDITOR ON A VDT

1. BID SCI (HIT RESET THEN !)

   IF SYSTEM RESPONDS WITH THE FOLLOWING SCREEN FORMAT, THEN GO TO STEP 4.

   SELECT ONE OF THE FOLLOWING COMMAND GROUPS

   /DEV - DEVICE OPERATIONS

   /FILE - FILE OPERATIONS

   /PSDEV - PROGRAM DEVELOPMENT

   /SMAIN - DX10 MAINTENANCE

   /SOP - DX10 OPERATION

   []

   IF THE SYSTEM RESPONDS ONLY WITH THE BRACKETS PROMPT ([]), THEN GO TO STEP 2.

2. PUT TERMINAL IN VDT MODE.

IF ONLY THE BRACKETS PROMPT ([]) WAS PRINTED, THEN THE VDT IS IN TTY MODE AND SHOULD BE CHANGED TO VDT MODE. THIS MAY BE DONE USING THE MODIFY TERMINAL STATUS (MTS) COMMAND. (SECTION 1, VOLUM II, DX10 3.0).

USER DOES THE FOLLOWING (UNDERLINED INFORMATION IS ENTERED BY THE USER).

```
[] MTS
MODIFY TERMINAL STATUS
            TERMINAL NAME:  ME
      NEW STATUS (ON/OFF):  ON
       NEW MODE (TTY/VDT):  VDT
          LOGIN REQUIRED?:  NO
      USER PRIVILEGE CODE:  7
   DEFAULT MODE (TTY/VDT):  <CR>


[]Q
QUIT
```

3. REBID SCI (RESET - !)
(COMPUTER SHOULD RESPOND)

SELECT ONE OF THE FOLLOWING COMMAND GROUPS

```
    /DEV        -  DEVICE OPERATIONS
    /FILE       -  FILE OPERATIONS
    /PDEV       -  PROGRAM DEVELOPMENT
    /SMAIN      -  DX10 MAINTENANCE
    /SOP        -  DX10 OPERATION
```

[]

(TERMINAL IS NOW IN VDT MODE)

4. EXECUTE THE TEXT EDITOR (XE)

```
[] XE
INITIATE TEXT EDITOR
    FILE ACCESS NAME:  PATANAME OF FILE TO BE EDITED
```

CREATING A NEW PROGRAM

1) CLEAR 'FILE ACCESS' FIELD (USE CLEAR KEY AS SHOWN BELOW)
2) HIT RETURN (NEW LINE - 913)

CLEAR KEY INFORMATION		
TERMINAL -	911	913
CLEAR KEY -	ERASE INPUT	CLEAR

NOTE: YOU WILL INDICATE THE FILE TO WHICH THE PROGRAM IS WRITTEN WHEN YOU QUIT TEXT EDITING.

EXAMPLE - CREATE A NEW PROGRAM

```
[]XE
INITIATE TEXT EDITOR
    FILE ACCESS NAME:  <CR>

    USE EDIT OPERATIONS
    TO CREATE PROBLEM

    HIT COMMAND (HELP-913) KEY


[]QE
QUIT EDITOR
    ABORT:  NO

OUTPUT FILE ACCESS NAME:  TI.PASCAL.SRC.MYPROG
                REPLACE:  NO
   MOD LIST ACCESS NAME:  <CR>
```

OPERATION OF THE TEXT EDITOR

CREATING A NEW FILE

- EXECUTE THE EDITOR
  - FOLLOW THE STEPS ALREADY OUTLINED (XE)
  - '*EOF' SHOULD APPEAR ON SCREEN
  - HIT 'F7' KEY TO ENTER COMPOSE MODE
  - HIT 'RETURN' (NEW LINE - 913)
  - ENTER YOUR PROGRAM
  - RETURN TO EDIT MODE ('F7' KEY AGAIN)

COMPOSE MODE

- ENTERED FROM EDIT MODE BY HITTING THE 'F7' KEY
- EACH TIME USER STRIKES 'RETURN', A BLANK LINE IS GENERATED
- TABS ARE PRESET TO COLUMNS 1, 8, 13, 26, & 31
- ALL EDIT FUNCTIONS ARE ACTIVE
- RETURN TO EDIT MODE VIA 'F7' KEY
- EXIT EDITOR VIA QUIT EDITOR (QE) COMMAND.

EDIT MODE

- USED TO EDIT A NEWLY CREATED FILE OR AN EXISTING FILE
- TEXT EDITOR IS IN EDIT MODE WHEN IT IS EXECUTED
- 'RETURN' (NEW LINE) KEY SIMPLY CAUSES CURSOR TO BE POSITIONED TO
- TABS ARE ACTIVE
- ALL EDIT FUNCTIONS ARE ACTIVE
- ENTER COMPOSE MODE BY HITTING THE 'F7' KEY
- EXIT EDITOR VIA QUIT EDITOR (QE) COMMAND.

EDIT FUNCTIONS

- ACTIVE IN BOTH EDIT AND COMPOSE MODES
- ALLOWS USER TO PERFORM CERTAIN EDITING FUNCTIONS BY STRIKING A PARTICULAR KEYBOARD CHARACTER.

FUNCTIONS	913 KEYTOP	911 KEYTOP	TTY CONTROL
ENTER COMMAND MODE	HELP	COMMAND	X
EDIT/COMPOSE FLIP (1)	F7	F7	V
DISP/SUPRS LINE NO (2)	F6	F6	F
CLEAR TO TAB	F5	F5	E
ROLL UP	ROLL UP	F1	A
ROLL DOWN	ROLL DOWN	F2	B
DUP TO TAB	F4	F$	D
NEW LINE	NEW LINE	RETURN	CAR RETN
TAB	TAB	TAB SKIP (3)	I
BACK TAB	BACK TAB	LFT FLD	T
INSERT LINE	INS LINE	UNLABELED GRAY KEY	O
DELETE LINE	DEL LINE	ERAS INP	N
INSERT CHARACTER	INS CHAR	INS CHR	****
DELETE CHARACTER	DEL CHAR	DEL CHR	****
CURSOR UP			U
CURSOR DOWN			J
CURSOR RIGHT			****
CURSOR LEFT (BACKSPACE)			****
HOME	HOME	HOME	****
ERASE	CLEAR	ERAS FLD	****

1). ALTERNATES MODES ON SUCCEEDING HITS

2). ALTERNATES DISPLAY OF LINE NUMBERS (74 DATA CHARACTERS) WITH NO DISPLAY OF LINE NUMBERS (80 CHARACTERS)

3). THE 'SHIFT' KEY MUST BE PRESSED SIMULTANEOUSLY WITH THE 'TAB SKIP' KEY TO ACHIEVE THE TAB FUNCTION ON THE 911 VDT

COMMAND MODE

- ENTERED BY HITTING THE COMMAND (HELP - 913)
- '[]' PROMPT IS DISPLAYED
- ALLOWS USER TO PERFORM EDITING FUNCTIONS BY ENTERING AN EDIT SCI COMMAND WITH PARAMETER SPECIFICATIONS.

COMMAND MNEMONIC		COMMAND DESCIRPTION
XE	-	EXECUTE TEXT EDITOR
QE	-	QUIT EDITOR
CL	-	COPY LINES
DL	-	DELETE LINES
DS	-	DELETE STRING
FS	-	FIND STRING
IF	-	INSERT FILE
ML	-	MOVE LINES
MR	-	MODIFY ROLL
MRM	-	MODIFY RIGHT MARGIN
RS	-	REPLACE STRING
MT	-	MODIFY TABS
SL	-	SHOW LINE

QUIT EDIT COMMAND (QE)

ALLOWS USER TO EXIT EDITOR AND SAVE CHANGES MADE TO AN EXISTING FILE OR SAVE FILE WHICH WAS CREATED.

TO QUIT EDITOR

- HIT COMMAND (HELP - 913) KEY TO ENTER COMMAND MODE
- [] PROMPT IS DISPLAYED
- ENTER QE
- THE FOLLOWING MESSAGE IS DISPLAYED

```
QUIT EDITOR
     ABORT:  NO
```

USER RESPONDS

Y	- ABORT EDIT SESSSION - NO CHANGES ARE SAVED
RETURN	- OUTPUT MESSAGE IS DISPLAYED (SEE NEXT PAGE)

```
 OUTPUT FILE ACCESS NAME:  TI.PASCAL.SRC.LAKES1      (1)
                REPLACE?:  NO        (2)
    MOD LIST ACCESS NAME:       (3)
```

(1) USER ENTERS

RETURN	-	TO USE DISPLAYED FILE AS OUTPUT FILE FOR EDITED TEXT
NEW FILE NAME	-	TO HAVE EDITED TEXT TO GO TO ANOTHER FILE AND PRESERVE OLD FILE.

(2) USER ENTERS

RETURN	-	INDICATES USER DOES NOT WISH TO REWRITE AN EXISTING FILE
YES	-	OUTPUT FILE IS TO BE REPLACED IF IT ALREADY EXISTS

(3) USER ENTERS

RETURN	-	NO LISTING OF MODIFICATIONS IS OBTAINED
FILE OR DEVICE NAME	-	LIST OF MODIFICATIONS TO FILE IS OUTPUT TO NAME SPECIFIED.

*THE OTHER COMMANDS ARE DESCRIBED IN SECTION 2.4.1 OF VOLUME IV OF THE DX10 MANUAL.

SUMMARY

TEXT EDITOR

- ALLOWS THE USER TO CREATE AND MODIFY TEXT FILES
- PROVIDES 3 MODES OF OPERATION

COMPOSE MODE

- USED TO CREATE NEW FILES
- USED TO INSERT LARGE BLOCKS OR RECORDS INTO EXISTING FILES
- ENTER FROM EDIT MODE VIA 'F7' KEY
- BLANK LINE GENERATED EACH TIME 'RETURN' (NEW LINE) IS ENTERED
- RETURN TO EDIT MODE VIA 'F7' KEY

EDIT MODE

- USED TO EDIT EXISTING FILES
- EDITOR IS IN EDIT MODE WHEN IT IS INITIALLY EXECUTED
- ENTER COMPOSE MODE VIA 'F7' KEY
- EXIT EDITOR VIA 'QE' COMMAND

COMMAND MODE

- USED TO PERFORM PREDEFINED EDIT FUNCTIONS
- ENTER FROM EDIT OR COMPOSE MODE VIA 'CMD' (HELP) KEY
- EXIT EDITOR VIA 'QE' COMMAND

DISPLAY A LISTING

ONCE A LISTING FILE HAS BEEN GENERATED, IT MAY BE DISPLAYED IN ONE OF TWO WAYS.

SHOW FILE COMMAND

PURPOSE: DISPLAYS THE CONTENTS OF A SPECIFIED FILE AT YOUR STATION

FORMAT:

```
[]SF

SHOW FILE

     FILE PATHNAME:  PATHNAME OF FILE TO BE DISPLAYED
```

PRINT FILE COMMAND

PURPOSE: TO PRINT THE CONTENTS OF A FILE TO A LISTING DEVICE.

FORMAT:

```
[]PF

PRINT FILE

                FILE PATHNAME:  PATHNAME OF FILE TO BE PRINTED
                  ANSI FORMAT:  NO
               LISTING DEVICE:  LISTING DEVICE NAME
 DELETE AFTER PRINTING?:  NO
   NUMBER OF LINES/PAGE:  <CR>
```

EXAMPLE:

```
[] PF
PRINT FILE
            FILE PATHNAME:  TI.PASCAL.LST.MYPROG
              ANSI FORMAT:  NO
           LISTING DEVICE:  LP01
  DELETE AFTER PRINTING?:  NO
    NUMBER OF LINES/PAGE:  <CR>
```

DATA TYPES

OBJECTIVES:

- BE ABLE TO DECLARE A VARIABLE TO BE OF THE FOLLOWING TYPES:

  BOOLEAN
  SCALAR
  SUBRANGE
  ARRAY
  RECORD

- BE ABLE TO ACCESS ELEMENTS OF AN ARRAY.

- BE ABLE TO ACCESS FIELDS OF A RECORD.

- GIVEN A SCALAR DECLARATION BE ABLE TO DETERMINE THE VALUE OF THE FUNCTIONS ORD, SUCC, AND PRED.

AGENDA

1. TYPE BOOLEAN

2. SCALARS

3. SUBRANGES

4. ARRAYS

WORKSHEET

5. PACKED ARRAYS

6. RECORDS

WORKSHEET

TYPE BOOLEAN

DEFINITION: THE VALUE OF A VARIABLE OF TYPE BOOLEAN IS EITHER TRUE OR FALSE

SYNTAX: VAR <BOOLEAN VARIABLE> : BOOLEAN;

EXAMPLE:
```
VAR SUCCESS : BOOLEAN;
      .
      .
SUCCESS := FALSE;
IF VALUE = 10 THEN SUCCESS := TRUE;
```

A BOOLEAN EXPRESSION CAN BE DIRECTLY ASSIGNED TO A BOOLEAN VARIABLE

EXAMPLE: THE ABOVE 'IF' STATEMENT CAN BE REWRITTEN AS

```
SUCCESS := VALUE = 10;
```

A BOOLEAN VARIABLE MAY BE TESTED INSTEAD OF A BOOLEAN EXPRESSION

EXAMPLE:
```
IF SUCCESS THEN <STATEMENT>
ELSE <STATEMENT>;
```

OPERATIONS ON BOOLEAN VARIABLES OR BOOLEAN EXPRESSIONS

LOGICAL OPERATORS:

THE FOLLOWING OPERATORS YIELD A VALUE OF TRUE OR FALSE WHEN APPLIED T
BOOLEAN VARIABLES:

NOT

AND

OR

OPERATOR PRECEDENCE:

* / MOD DIV	HIGHEST PRECEDENCE
+ -	
< > <> <= >= =	
NOT	
AND	
OR	LOWEST PRECEDENCE

NOT

NOT IS TRUE WHEN THE OPERAND IS FALSE

EXAMPLES: NOT (5 > 3) FALSE

NOT SUCCESS

WHEN SUCCESS = TRUE FALSE
WHEN SUCCESS = FALSE TRUE

NOT (I = 10)

WHEN I = 10 FALSE
WHEN I = 5 TRUE
WHEN I = 15 TRUE

AND

AND IS TRUE ONLY WHEN BOTH OPERANDS ARE TRUE

	T	F
T	T	F
F	F	F

EXAMPLES:

1 > 3 AND 10 = 10 FALSE

I <= 20 AND FINISHED

WHEN I = 10 FINISHED = TRUE TRUE
WHEN I = 20 FINISHED = FALSE FALSE
WHEN I = 30 FINISHED = TRUE FALSE

X <> Y AND NOT SUCCESS

WHEN X = Y SUCCESS = FALSE FALSE
WHEN X = 5, Y = 10, SUCCESS = TRUE FALSE
WHEN X = 10, Y = 4, SUCCESS = FALSE TRUE

OR

OR IS TRUE WHEN ONE OR BOTH OF THE OPERANDS IS TRUE.

	T	F
T	T	T
F	T	F

EXAMPLES:

1 < 3 OR 10 > 5	TRUE
1 < 3 OR 15 < 20	TRUE
1 > 3 OR 15 > 3	TRUE

I <= 10 OR SUCCESS

WHEN I = 5, SUCCESS = TRUE	TRUE
WHEN I = 11, SUCCESS = TRUE	TRUE
WHEN I = 11, SUCCESS = FALSE	FALSE

I <= N OR NOT FOUND

WHEN I = 5, N = 10, FOUND = FALSE	TRUE
WHEN I = 10, N = 10, FOUND = FALSE	TRUE
WHEN I = 11, N = 10, FOUND = TRUE	FALSE
WHEN I = 11, N = 10, FOUND = FALSE	TRUE

A BOOLEAN OPERATOR CAN SIMPLIFY A NESTED 'IF'

EXAMPLE:

```
IF X > 0 THEN
   IF (X / 2) < N THEN
```

CAN BE REPLACED WITH

```
IF X > 0 AND (X / 2) < N THEN
```

```
IF CODE < 1 THEN
   ERROR := TRUE
ELSE IF CODE > 5 THEN
   ERROR := TRUE;
```

CAN BE REPLACED WITH

```
IF CODE < 1 OR CODE > 5 THEN
   ERROR := TRUE;
```

IN EACH CASE DETERMINE IF THE STATEMENT WILL EXECUTE:

IF I < 10 AND NOT FINISHED THEN <STATEMENT>

WHEN I = 3, FINISHED = FALSE	YES
WHEN I = 10, FINISHED = FALSE	NO
WHEN I = 10, FINISHED = TRUE	NO
WHEN I = 5, FINISHED = FALSE	YES

IF I < 10 OR NOT FINISHED THEN <STATEMENT>

WHEN I < 10, FINISHED = FALSE	YES
WHEN I = 10, FINISHED = FALSE	YES
WHEN I = 10, FINISHED = TRUE	NO
WHEN I = 5, FINISHED = FALSE	YES

SCALARS

DEFINITION: AN ORDERED LIST OF IDENTIFIERS

SYNTAX: TYPE <NAME> = (<IDENTIFIER>, ..., <IDENTIFIER>);

* THE IDENTIFIERS MUST BE UNIQUE.
* SCALARS ARE NUMBERED STARTING WITH ZERO.
* YOU MAY NOT DO ARITHMETIC ON SCALAR VARIABLES.
* YOU CAN'T READ OR PRINT OUT A SCALAR VALUE.

EXAMPLES:

```
TYPE FAMILY = (MOTHER, FATHER, SON, DAUGHTER);
TYPE PET = (CAT, DOG, BIRD, FISH);
VAR PARTIES: (REPUBLICAN, DEMOCRAT);
VAR VOWELS: (A,E,I,O,U);
```

SCALARS MAY BE USED FOR THE FOLLOWING:

- AS AN ARRAY INDEX
- AS A FLAG
- TO IMPROVE CODE READABILITY

EXAMPLE:

```
TYPE COLOR = ( WHITE, RED, BLUE, YELLOW, PURPLE,
                GREEN, ORANGE, BLACK );
VAR C, C1: COLOR;
    SUCCESS : BOOLEAN;


IF C > WHITE THEN <STATEMENT>
```

SCALAR VARIABLES MAY BE USED AS A FLAG WHEN MORE THAN 2 STATES ARE NEEDED.

EXAMPLE:

```
TYPE SIGN = (POSITIVE, ZERO, NEGATIVE);

VAR ISIGN : SIGN;

    I : INTEGER;

BEGIN

  READ(I);

  IF I > 0 THEN ISIGN := POSITIVE

  ELSE IF I = 0 THEN ISIGN := ZERO

  ELSE ISIGN := NEGATIVE;

                .
                .
                .

  (* TEST ISIGN *)

  IF ISIGN = POSITIVE THEN WRITE(I)

  ELSE IF ISIGN = NEGATIVE THEN WRITE(-I)

  ELSE WRITE('I IS ZERO');
```

NOTES: SIGN IS A GENERAL TYPE. ISIGN IS A SPECIFIC INSTANCE OF A VARIABLE OF TYPE SIGN. POSITIVE, ZERO, AND NEGATIVE ACT LIKE CONSTANTS.

SCALAR OPERATIONS

YOU MAY DO NO ARITHMETIC ON SCALARS.

SCALARS MAY BE MANIPULATED WITH THE FOLLOWING FUNCTIONS:

PRED(X)	(X'S PREDECESSOR) GIVES THE SCALAR BEFORE X
SUCC(X)	(X'S SUCCESSOR) GIVES THE SCALAR AFTER X
ORD(X)	GIVES X'S NUMBER IN THE ORDERING. SCALARS ARE NUMBERED STARTING WITH 0.

EXAMPLES:

```
TYPE PRIMARY = (RED, BLUE, YELLOW);
TYPE SUIT = (SPADE, CLUB, HEART, DIAMOND);
VAR X, Y: PRIMARY;
          .
          .
X := RED;
Y := SUCC(X); (* Y WILL BE BLUE *)
```

```
ORD(SPADE) = 0
ORD(BLUE) = 1
SUCC(CLUB) = HEART
PRED(RED) = UNDEFINED      WILL GIVE AN ERROR
SUCC(DIAMOND) = UNDEFINED

IF X > SPADE THEN
   X := PRED( X );
```


156

SUBRANGES

DEFINITION: SUBRANGES ARE USED WHEN WE KNOW A VARIABLE WILL ONLY HAVE A SUBSET OF ITS POSSIBLE VALUES. (I.E. INTEGERS FROM 1 TO 10 OR GRADES FROM 0 TO 100.

* SUBRANGES OF THE TYPE REAL ARE NOT ALLOWED

SYNTAX: VAR <IDENTIFIER> : <LOWER BOUND> .. <UPPER BOUND>

THESE TWO DOTS ACTUALLY APPEAR IN THE PROGRAM

* BOTH LOWER BOUND AND UPPER BOUND MUST BE CONSTANTS
* YOU CANNOT SAY VAR I : 0 .. N + 1

EXAMPLE:

```
TYPE TESTSCORE = 0..100;
VAR MYSCORE, YOURSCORE : TESTSCORE;
                OR
VAR MYSCORE, YOURSCORE : 0..100
```

```
TYPE IQ = 0 .. 200;                    (* SUBRANGE OF INTEGER*)
     DAYS = (MON, TUE, WED, THU, FRI, SAT, SUN);
     WORKD = MON..FRI;  (*SUBRANGE OF SCALER TYPE 'DAYS'*)
     LETTER = 'A'..'Z'; (*SUBRANGE OF CHARACTERS*)
```

SUBRANGES MAY INTERSECT:

```
EXAMPLE:  VAR A : 10..20;      IN THIS CASE A := B IS LEGAL
          B : 15..20;                       A := C IS ILLEGAL
          C : 21..30;                       B := A BE CAREFUL!
```

RUNTIME OPTION CKSUB

- ENABLES OR DISABLES THE CHECKING OF SUBRANGE ASSIGNMENTS AND THE RESULTS OF PRED AND SUCC FUNCTIONS
- DEFAULT IS FALSE

ARRAYS

DEFINITION: SAME IDEA AS FORTRAN, BUT MORE GENERAL.

SYNTAX: VAR <ARRAY NAME> : ARRAY[<INDEX TYPE>] OF <BASE TYPE>

INDEX TYPES

* INTEGER
* LONGINT
* CHAR
* BOOLEAN
* SCALAR
* SUBRANGE

BUT NOT REAL !

EXAMPLE:

```
CONST N = 20;
TYPE IQ = 0..200;
     PEOPLE = (JANE, BILLIE, LES);
     C = ARRAY[1..N] OF CHAR;
-------------------------------------------------------

VAR INT : ARRAY[0..N] OF INTEGER;
    CH : ARRAY[CHAR] OF 0..64;    (* CH WILL BE INDEXED BY
                                  (* CHARACTERS              *)
    INTELL : ARRAY[PEOPLE] OF IQ;  (* INTELL WILL BE INDEXED
                                   (* BY JANE, BILLIE, OR   *)
                                   (* LES                   *)
    NAME : ARRAY[PEOPLE] OF C;
    NAMES : ARRAY[PEOPLE] OF ARRAY [1..N] OF CHAR;
       (* NAME AND NAMES DEFINE THE SAME ARRAY *)
```

ACCESSING ARRAYS

```
GIVEN   VAR I : INTEGER; A : CHAR;
            NAMES : (JANE, LES, BILLIE);

INT[ I ]                INDEXING BY AN INTEGER

INT[ I + 1 ]

CH['A']                 INDEXING USING CHARACTERS

INTELL[ JANE ]          INDEXING USING A SCALAR
```

MULTIDIMENSIONAL ARRAYS

```
SYNTAX:  VAR A: ARRAY[1..10] OF ARRAY[1..20] OF INTEGER

                              OR

         VAR A: ARRAY[1..10, 1..20] OF INTEGER
```

ACCESSING AN ELEMENT:

```
         A[I, J] := X

         I REFERS TO THE ROW

         J REFERS TO THE COLUMN
```

```
EXAMPLES:  TYPE STUDENT = (JAMES, JOHN, SAM, SALLY);

           VAR GRADES : ARRAY[STUDENT, 1..10] OF 0..100;

           GRADES[JOHN, 2]
```

**NOTE: ON THE 913, '(.' AND '.)' SUBSTITUTE FOR '[' AND ']'.

COMPILER OPTION CKINDEX

- ENABLES OR DISABLES CHECKING FOR ARRAY INDICES WITHIN RANGE.
- DEFAULT--FALSE

PROGRAM EXAMPLES

ASSUME VAR A : ARRAY[1..N] OF T;
SEARCH A FOR A VALUE V.

```
VAR I : INTEGER;

BEGIN

  SUCCESS := FALSE;

  I := 1;

  WHILE (I <= N) AND NOT SUCCESS DO

      IF A[I] = V THEN

         SUCCESS := TRUE          (* NO ';' *)

      ELSE I := I + 1             (* NO ';' *)

END;  (* SEARCH *)
```

INSERT V AFTER THE ITH ELEMENT

ASSUME THERE ARE LESS THAN N ELEMENTS IN THE ARRAY SO THAT THE LAST POSITION IS VACANT

```
PROCEDURE INSERT ( V : T; I : INTEGER);


BEGIN
   FOR J := N - 1 DOWNTO I + 1 DO
        A [ J + 1 ] := A [ J ];     (* MOVE EACH ELEMENT DOWN ONE *)
   A [ I + 1 ] := V                 (* INSERT V                   *)
END;
```

Write a procedure DELETE which will delete the I'th element from an Array. The index of the element to be deleted should be passed as a parameter.

WORKSHEET 7
---

1. WRITE THE DECLARATION STATEMENT TO DECLARE A VARIABLE 'FOUND' AS TYPE BOOLEAN.

DECIDE WHETHER THE FOLLOWING BOOLEAN EXPRESSIONS ARE TRUE OR FALSE

2. NOT (6 < 9)

3. NOT DONE WHEN DONE = FALSE

4. WHEN DONE = TRUE

5. (4 < 10) AND (1 > 0)

SUCCESS OR ENDLIST

6. WHEN SUCCESS = FALSE, ENDLIST = TRUE

7. WHEN SUCCESS = TRUE, ENDLIST = FALSE

8 WHEN SUCCESS = FALSE, ENDLIST = FALSE

9. REWRITE THE FOLLOWING NESTED 'IF' STATEMENT AS A SINGLE 'IF' STATEMENT USING A LOGICAL OPERATOR

```
IF I <= N THEN
   IF NOT SUCCESS THEN
      <STATEMENT>
```

10. WRITE THE DECLARATION STATEMENT TO DECLARE A TYPE LANGUAGE AS A SCALAR WHICH MAY BE ONE OF THE FOLLOWING: FORTRAN, PASCAL, COBOL.

GIVEN THE DECLARATION:

```
VAR DOGS : (COLLIE, SPANIEL, SHEPHERD, MUTT)
```

11. ORD(SPANIEL) =

12. SUCC(SHEPHERD) =

13. SUCC(MUTT) =

14. PRED(MUTT) =

15. USING A SUBRANGE, DECLARE A TYPE 'BIT' WHICH CAN HAVE VALUES FROM 0 TO 15.

16. WRITE THE VAR STATEMENT TO DECLARE AN ARRAY 'WORDS' CONSISTING OF FROM 1 TO 10 CHARACTERS.

17. DECLARE A 2-DIMENSIONAL ARRAY CLASS OF ARRAY OF GRADES. THERE ARE 20 STUDENTS IN THE CLASS AND EACH STUDENT HAS 10 GRADES.

18. WRITE A STATEMENT WHICH WILL ASSIGN THE 4TH STUDENT IN THE CLASS OF THE ABOVE ARRAY A GRADE OF 95 ON THE FIRST TEST.

PACKED ARRAYS

DEFINITION: A PACKED ARRAY ECONOMIZES STORAGE BY STORING SEVERAL COMPONENTS IN ONE WORD.

SYNTAX: VAR <NAME> : PACKED ARRAY[N1..N2] OF <TYPE>

EXAMPLE: VAR BOOL : PACKED ARRAY[1..16] OF BOOLEAN

'BOOL' WILL TAKE ONE 16-BIT WORD.

```
TYPE X = PACKED ARRAY[1..4] OF 0..7;
  (* WILL TAKE 1 16-BIT WORD *)

   Y = PACKED ARRAY[1..5] OF X;
(* Y WILL TAKE 60 BITS OR 4 WORDS *)
                OR
   Y = PACKED ARRAY[1..5, 1..4] OF 0..7;
```

NOTE: IF THE COMPONENTS OF AN ARRAY REQUIRE ONE WORD OR MORE OF STORAGE, PACKING THAT ARRAY HAS NO EFFECT ON THE STORAGE ALLOCATION.

PACK AND UNPACK

THE PROGRAMMER MAY PACK OR UNPACK AN ARRAY A BY USING THE PROCEDURES 'PACK(A,I,Z)' AND 'UNPACK(Z,A,I)'

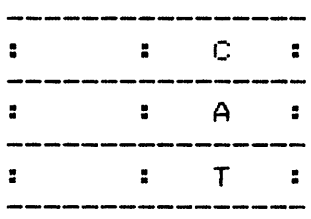
```
PACK(A,I,Z) MEANS FOR J := U TO V DO
                      Z[J] := A[J - U + I]

UNPACK(Z,A,I) MEANS FOR J := U TO V DO
                      A[J - U + I] := Z[J]
```

J IS THE 'FOR LOOP' INDEX
A IS THE UNPACKED ARRAY
Z IS THE PACKED ARRAY
I IS THE INDEX INTO THE UNPACKED ARRAY

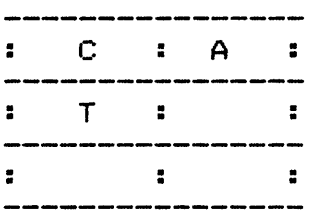
**NOTE: THE PASCAL COMPILER WILL UNPACK A PACKED ARRAY IF YOU REFERENCE AN ELEMENT OF THE ARRAY, AND THEN REPACK THE ARRAY.

TO SAVE TIME UNPACK THE ARRAY WHILE IT IS BEING ACCESSED THEN REPACK IT.

STORAGE FOR UNPACKED ARRAY



STORAGE FOR PACKED ARRAY



USING PACK AND UNPACK

```
VAR A : ARRAY [ 1..5 ] OF INTEGER;
    P : PACKED ARRAY [ 1..5 ] OF INTEGER;

A =
        1
        2
        3
        4
        5

PACK (A,1,P)
P =
        1
        2
        3
        4
        5

SET ELEMENTS OF A = 0
A =
        0
        0
        0
        0
        0

UNPACK (P,A,1)
A =
        1
        2
        3
        4
        5

SET ELEMENTS OF A = 0
A =
        0
        0
        0
        0
        0

UNPACK (P,A,2)
A =
        0
        1
        2
        3
        4

SET ELEMENTS OF P = 0
P =
        0
        0
        0
        0
        0

PACK (A,3,P)
P =
        2
        3
        4
        2
        3
```

```
VAR U : ARRAY [1..10] OF INTEGER
    P : PACKED ARRAY [1..5] OF INTEGER
    PACK(U,1,P)
    P=          1
                2
                3
                4
                5

    UNPACK (P,U,1)
    U =
                1
                2
                3
                4
                5
                0
                0
                0
                0
                0
```

STRINGS

DEFINITION: A STRING IS A PACKED ARRAY OF TYPE CHAR.

* THE LOWER INDEX OF THE ARRAY MUST BE 1

* A STRING MAY NOT BE LONGER THAN 70 CHARACTERS

* ANY CHARACTER MAY BE REPRESENTED IN A STRING BY A # FOLLOWED BY ITS 2-DIGIT HEXADECIMAL CHARACTER CODE. THIS ENABLES UNPRINTABLE CONTROL CHARACTERS TO BE INCLUDED IN STRINGS.

EXAMPLE:

```
VAR STRING : PACKED ARRAY[1..N] OF CHAR;
                    .
                    .
STRING := 'THIS IS A STRING';      (* OK IF N = 16 *)

STRING := 'ABC';                   (* OK IF N = 3 *)

STRING[1] := 'A';
```

RELATIONAL OPERATORS MAY BE USED ON STRINGS IF THEY ARE THE SAME TYPE AND SAME LENGTH.

EXAMPLE:

```
VAR STR1, STR2 : PACKED ARRAY[1..10] OF CHAR;

    STR3 : PACKED ARRAY[1..16] OF CHAR;

    STR4 : PACKED ARRAY[5..8] OF CHAR;
                    .
                    .
STR1 := STR2                (* OK *)

IF STR1 < STR2  THEN
   X := 1;                  (* OK *)

IF STR2 >= STR3 THEN        (* NOT LEGAL--STR2 AMD STR3 *)
   X := 2;                  (* ARE NOT THE SAME LENGTH  *)

STR4 := 'WORD';             (* NOT LEGAL--INDEX DOESN'T *)
                            (* START AT 1               *)
STR1 := 'STRING    ';       (* OK *)

STR1 := 'STRING';           (* ILLEGAL--MUST HAVE 10    *)
```

RECORDS

RECORDS ARE A USEFUL WAY OF GROUPING RELATED DATA TOGETHER.

SYNTAX:
```
RECORD
  <FIELD> : <TYPE>;
          .
          .
  <FIELD> : <TYPE>       (* NO ; *)
END
```

EXAMPLE: DECLARE A TYPE DATE WHICH IS A RECORD CONTAINING THE DAY, MONTH, AND YEAR.

```
TYPE DATA = RECORD
              DAY : 1..31;
              MO  : 1..12;
              YR  : INTEGER
            END;


VAR MYBIRTHDAY : DATE;

    HOLIDAYS : ARRAY[ 1..10] OF DATA;
```

EXAMPLE: DECLARE A TYPE EM_DATA WHICH IS A RECORD CONTAINING THE EMPLOYEES NAME (UP TO 20 DIGITS) AND HIS SOCIAL SECURITY NUMBER.

ACCESSING A FIELD IN A RECORD

SYNTAX: <RECORD>.<FIELD NAME>

EXAMPLE: GIVEN THE FOLLOWING RECORD, ASSIGN MYBIRTHDAY TO BE 9/20/47

```
TYPE DATE = RECORD
              DAY : 1..31;
              MO  : 1..12;
              YR  : INTEGER
            END;
VAR MYBIRTHDAY : DATE;
    HOLIDAYS : ARRAY[ 1..10 ] OF DATE;


MYBIRTHDAY.DAY := 20;
MYBIRTHDAY.MO := 9;
MYBIRTHDAY.YR := 1947;
```

ASSIGN THE FIRST HOLIDAY TO BE NEW YEARS DAY THIS YEAR.

NESTED RECORDS

EXAMPLE: DECLARE A RECORD WHICH WILL CONTAIN A STUDENT'S NAME (UP TO 20 CHARACTERS), HIS SCHOOL ENTRANCE DATE AND THE DATE HE GRADUATED.

```
TYPE STU_DATA = RECORD
                    NAME : PACKED ARRAY[ 1..20 ] OF CHAR;
                    ENTERED : RECORD
                                DAY : 1..31;
                                MO  : 1..12;
                                YR  : INTEGER
                              END;
                    GRADUATED : RECORD
                                  DAY : 1..31;
                                  MO  : 1..12;
                                  YR  : INTEGER
                                END
                END;


VAR STUDENT : STU_DATA;


STUDENT.NAME := 'JANE LOBDILL         ';
STUDENT.ENTERED.DAY := 15;
STUDENT.ENTERED.MO := 9;
STUDENT.ENTERED.YR := 1966;
STUDENT.GRADUATED.DAY := 20;
STUDENT.GRADUATED.MO  := 6;
STUDENT.GRADUATED.YR  := 1969;
```

EXAMPLE: CONSTRUCT A RECORD WHICH WILL CONTAIN THE FOLLOWING INFORMATION:

```
EMPLOYEE NUMBER         (6 DIGITS)
SOCIAL SECURITY NUMBER (9 DIGITS)
NUMBER OF DEPENDENTS
DATE EMPLOYED
```

PACKED RECORDS

```
EXAMPLE :  TYPE R = PACKED RECORD
                      A : PACKED ARRAY [1..10] OF 1..31;
                      J : 0..7;
                      K : 0..#FFF;
                      L : INTEGER
                    END;
```

SUMMARY

- A BOOLEAN VARIABLE MAY HAVE A VALUE OF TRUE OR FALSE.
- A BOOLEAN EXPRESSION MAY BE ASSIGNED TO A BOOLEAN VARIABLE.
- A BOOLEAN VARIABLE MAY BE TESTED INSTEAD OF A BOOLEAN EXPRESSION.
- 'AND', 'OR', AND 'NOT', ARE LOGICAL OPERATORS AND MAY BE APPLIED TO BOOLEAN VARIABLES.
- 'NOT' IS TRUE WHEN THE OPERAND IS FALSE.
- 'AND' IS TRUE WHEN BOTH OPERANDS ARE TRUE.
- 'OR' IS TRUE WHEN AT LEAST ONE OF THE OERANDS IS TRUE.
- A BOOLEAN OPERATOR CAN SIMPLIFY A NESTED 'IF'.
- SCALARS ARE AN ORDERED LIST OF IDENTIFIERS.
  YOU MAY DO NO ARITHMETIC ON SCALARS OR READ THEM OR PRINT THEM OUT.
  SCALARS ARE USED TO INDEX ARRAYS OR AS A FLAG.
  SCALAR OPERATIONS ARE PRED, SUCC, AND ORD.
- A SUBRANGE IS A SUBSET OF VALUES.
- ARRAYS ARE SIMILAR TO FORTRAN ARRAYS. THE INDEX MAY NOT BE OF TYPE REAL.
- RECORDS ARE A WAY OF GROUPING RELATED DATA TOGETHER.

WORKSHEET 8

1. WRITE THE DECLARATION FOR A STRING OF 10 CHARACTERS CALLED WORD.

WRITE STATEMENTS TO DO THE FOLLOWING:

2. ASSIGN 'WORD' THE VALUE 'DEVELOPING'

3. ASSIGN 'WORD' THE VALUE 'TOM'

4. ASSIGN 'WORD' THE VALUE ALL BLANKS.

5. DECLARE A RECORD 'DOG' WHICH CONTAINS THE FOLLOWING INFORMATION:

   BREED (UP TO 10 CHARACTERS)
   DOG'S NAME (UP TO 20 CHARACTERS)
   DOGS REGISTRATION NUMBER (UP TO 9 DIGITS)

6. ADD TO TYPE DOG ABOVE ANOTHER FIELD WHICH IS THE DOG'S BIRTHDAY. THE BIRTHDAY SHOULD HAVE 3 FIELDS, MONTH, DAY, AND YEAR.

7. GIVEN THE FOLLOWING DECLARATIONS ASSIGN STUDENT1 A GPA OF 3.5

```
TYPE DATA = RECORD
              NAME : PACKED ARRAY[ 1..20 ] OF CHAR;
              MAJOR : PACKED ARRAY [ 1..3 ] OF CHAR;
              GPA : REAL;
              HRS_COMPL : INTEGER
            END;
VAR STUDENT1 : DATA;
```

8. GIVEN THE FOLLOWING DECLARATIONS ASSIGN PROJECT1 A STARTING DATE OF MAY 12, 1977.

```
TYPE DATA = RECORD
              PROJ_NAME : PACKED ARRAY[ 1..10 ] OF CHAR;
              TOTAL_BUDGET: REAL;
              DATE_STARTED = RECORD
                               DAY : 1..31;
                               MO  : 1..12:
                               YR  : INTEGER
                             END
            END;
VAR PROJECT1 : DATA;
```

9. GIVEN THE FOLLOWING DECLARATIONS, WRITE ASSIGNMENT STATEMENTS TO ENTER A PURCHASE DATE OF JUNE 21, 1977 FOR AUTO #3.

```
TYPE CAR = RECORD
             DATE_PURCHASED = RECORD
                                DAY : 1..31;
                                MO  : 1..12;
                                YR  : INTEGER
                              END
           END;
VAR OUR_CARS : ARRAY[ 1..3 ] OF CAR;
```


173

PROGRAM CONTROL STATEMENTS II

OBJECTIVES:

- BE ABLE TO USE THE FOLLOWING PROGRAMMING STRUCTURES.
  REPEAT-UNTIL LOOP
  CASE STATEMENT
  WITH STATEMENT

AGENDA

1. PROGRAMMING STRUCTURES
   - REPEAT-UNTIL
   - CASE STATEMENT
   - WITH STATEMENT

WORKSHEET

REPEAT-UNTIL

SYNTAX:
```
REPEAT
   <STATEMENT>;
        .
        .
   <STATEMENT>;
UNTIL <BOOLEAN EXPRESSION>;
```

* REPEAT-UNTIL EXECUTES ONE OR MORE STATEMENTS REPEATEDLY UNTIL A CERTAIN CONDITION IS TRUE.

* A 'BEGIN-END' BLOCK IS NOT USED.

EXAMPLE:
```
VAR CH: CHAR;
     .
     .
REPEAT
   READ(CH);
   WRITE(CH);
UNTIL CH = '*';
```

WHILE-DO	REPEAT-UNTIL
START	START
TEST	STATEMENT
STATEMENT	TEST

* WHILE-DO MAKES THE TEST BEFORE THE FIRST ITERATION OF THE LOOP. REPEAT-UNTIL MAKES THE TEST AFTER THE FIRST ITERATION OF THE LOOP.

* USE 'REPEAT-UNTIL' WHEN THE VARIABLE BEING TESTED WILL NOT BE DEFINED ON THE FIRST PASS. (CH WAS NOT DEFINED UNTIL IT WAS READ.)

CASE STATEMENT

SYNTAX: CASE <EXPRESSION> OF

```
        <CASE LABEL>,...,<CASE LABEL> : <STATEMENT>;
                        .
                        .
        <CASE LABEL>,...,<CASE LABEL> : STATEMENT

        [ OTHERWISE <STATEMENT>;...;<STATEMENT> ]

    END
```

WHEN A CASE STATEMENT IS EXECUTED:

- THE EXPRESSION IS EVALUATED
- CONTROL PASSES TO THE STATEMENT WHOSE LABEL EXACTLY CORRESPONDS TO THE VALUE OF THE EXPRESSION.
- CONTROL PASSES TO THE NEXT STATEMENT FOLLOWING THE END OF THE CASE STATEMENT.

EXAMPLE: VAR I : INTEGER

```
CASE I OF
  0                       : X := 10;
  1, 3, 5                 : X := SIN( Y );
  6..10, 20, 30..40       : X := COS( Y )
END
```

IF I = 0 THEN I MATCHES THE LABEL ON STATEMENT 1 AND X WILL BE ASSIGNED THE VALUE 10.
IF I = 7 THEN I MATCHES THE LABEL ON STATEMENT 3 AND X WILL BE ASSIGNED THE VALUE COS(Y).

* IF I = 50 THE CASE STATEMENT IS UNDEFINED. THIS PROBLEM CAN BE AVOIDED BY USING THE 'OTHERWISE' CLAUSE.

```
CASE I OF
  0                       : X := 10;
  1, 3, 5                 : X := SIN( Y );
  6..10, 20, 30..40       : X := COS(Y);
  OTHERWISE WRITELN('X IS OUT OF BOUNDS')
END
```

EXAMPLES:

```
CASE DAY OF
     MON, TUE, WED, THU, FRI : BEGIN
                                 WRITELN(ID);
                                 WRITELN(HRS, JOBNUM)
                               END;
    SAT                      : WRITELN('SATURDAY');
    SUN                      :WRITELN('SUNDAY')
END
```

```
TYPE SIGN = (POSITIVE, ZERO, NEGATIVE);
VAR ISIGN : SIGN;
    I : INTEGER;

BEGIN
  READ(I);
  IF I > 0 THEN ISIGN := POSITIVE
  ELSE IF I < 0 THEN ISIGN := NEGATIVE
  ELSE IF I = 0 THEN ISIGN := ZERO;
                  .
                  .
  CASE ISIGN OF
       POSITIVE : WRITE(I);
       NEGATIVE : WRITE (-I);
       ZERO     : WRITE ('I = 0')
  END
```

EXAMPLE:

```
VAR CH : CHAR;

    VOWEL : BOOLEAN;

    CASE CH OF

         'A', 'E', 'I' : BEGIN

                           WRITE('IT''S A VOWEL');

                           VOWEL := TRUE

                         END;

         'B', 'C', 'D',
         'F', 'G', 'H',
         'J'             : BEGIN

                             WRITE('IT''S A CONSONANT');

                             VOWEL := FALSE

                           END;

         OTHERWISE WRITE('CH IS NOT A..J');
                   CH := ' '
   END
```

NOTES: THE LABELS WITHIN A CASE STATEMENT MUST BE UNIQUE, BUT LABELS IN DIFFERENT CASE STATEMENTS OR EVEN NESTED CASE STATEMENTS ARE INDEPENDENT.

IF THE VALUE OF THE EXPRESSION DOES NOT OCCUR AMONG THE LABELS, THE COMPUTATION IS UNDEFINED ! YOU WILL GET A RUN-TIME ERROR. YOU CAN AVOID THIS BY USING 'OTHERWISE'.

SEVERAL STATEMENTS MAY FOLLOW 'OTHERWISE'. A BEGIN-END IS NOT NECESSARY.

WITH STATEMENT

SYNTAX: WITH <VARIABLE LIST> DO <STATEMENT>

EXAMPLES:
```
TYPE DATE = RECORD
              DAY : INTEGER;
              MO  : (JAN, FEB, MAR, APR, MAY, JUN, JUL,
                     AUG, SEP, OCT, NOV, DEC);
              YR  : INTEGER
            END;
VAR TODAYSDATE : DATE;
              .
              .
TODAYSDATE.DAY := 1;
TODAYSDATE.MO := MAR;
TODAYSDATE.YR := 1978;
(* IS EQUIVALENT TO WRITING *)
WITH TODAYSDATE DO
  BEGIN
    DAY := 1;
    MO := MAR;
    YR := 1978
  END;
```

```
EXAMPLES:   P.NAME.LAST := 'HILL       ';
            P.NAME.FIRST := 'JANE       ';
            P.SS := 455809190;
            P.SEX := 'F';
            P.BIRTH.DAY := 7;
            P.BIRTH.MO := SEP;
            P.BIRTH.YR := 1947;
            P.DEPDTS := 1;
            P.MS := 'S'

        (* IS EQUIVALENT TO WRITING *)

            WITH P, NAME, BIRTH DO
              BEGIN
                LAST := 'HILL       ';
                FIRST := 'JANE       ';
                SS := 455809190;
                SEX := 'F';
                MO := SEP;
                DAY := 20;
                YR := 1947;
                DEPDTS := 1;
                MS := 'S'
              END
```

THE WITH STATEMENT MAY BE USED WITH ARRAYS

EXAMPLES:

```
TYPE FAMILY = (MOM, POP, KID1, KID2, KID3);

     DATE = RECORD
              MO  : 1..12;
              DAY : 1..31;
              YR  : INTEGER;
            END;

VAR VACCINE : ARRAY[FAMILY] OF DATA;
                    .
                    .
VACCINE[KID1].MO := 4;

VACCINE[KID1].DAY := 23;

VACCINE[KID1].YR := 1978;

                OR

WITH VACCINE[KID1] DO

  BEGIN

    MO := 5;

    DAY := 23;

    YR := 1978

  END
```

NOTE:

```
WITH A[I] DO

  BEGIN
                          IS NOT ALLOWED
    I := I + 1            --------------

  END
```

GIVEN THE EXPRESSION: 'WITH X DO Z'
X MUST NOT CONTAIN ANY VARIABLES SUBJECT TO CHANGE BY Z.

SUMMARY

- 'REPEAT-UNTIL' WILL EXECUTE ONE OR MORE STATEMENTS UNTIL A CERTAIN CONDITON IS TRUE.

- 'WHILE-DO' TESTS THE CONDITION BEFORE EXECUTING THE STATEMENT. 'REPEAT-UNTIL' MAKES THE TEST AFTER EXECUTING THE STATEMENT.

- THE 'CASE' STATEMENT IS SIMILAR TO THE 'ELSE-IF' CHAIN. EXACTLY ONE STATEMENT WHOSE LABEL CORRESPONDS EXACTLY TO THE VALUE OF THE EXPRESSION WILL BE EXECUTED.

- LABELS IN A PARTICULAR 'CASE' STATEMENT MUST BE UNIQUE, BUT DIFFERENT 'CASE' STATEMENTS ARE INDEPENDENT.

- THE 'WITH' STATEMENT IS A WAY OF SHORTENING RECORD FIELD ASSIGNMENTS.

WORKSHEET 9

1. USING A 'REPEAT-UNTIL' CALCULATE THE SMALLEST I SUCH THAT 1 + 1/2 + 1/3 +...+1/I IS GREATER THAN A CONSTANT N.

2. GIVEN 2 VALUES V1 AND V2 AND A CHARACTER CH = '+', '-', '*', OR '/' USE A 'CASE' STATEMENT TO PERFORM THE CORRECT OPERATION ON THE TWO ARGUMENTS. I.E. IF CH = '+' THEN Z := V1 + V2.

3. REWRITE THE FOLLOWING USING A 'WITH' STATEMENT:

```
EMPLOYEE[ ENUM ].SSN := 456966162;
EMPLOYEE[ ENUM ].HIRED.MO := 6;
EMPLOYEE[ ENUM ].HIRED.DAY := 25;
EMPLOYEE[ ENUM ].HIRED.YR := 1966;
EMPLOYEE[ ENUM ].GRADE := 12;
EMPLOYEE[ ENUM ].DEDUCTS := 2;
EMPLOYEE[ ENUM ].INSCODE := 2;
```

OBJECTIVES:

- BE ABLE TO WRITE A STATEMENT TO OUTPUT AN INTEGER USING A SPECIFIED NUMBER OF COLUMNS.
- BE ABLE TO WRITE A STATEMENT TO INPUT A NUMBER FROM A SPECIFIED NUMBER OF COLUMNS.
- BE ABLE TO WRITE A STATEMENT TO OUTPUT A REAL NUMBER BY SPECIFYING THE FIELD WIDTH AND THE NUMBER OF DECIMAL PLACES TO BE PRINTED.
- BE ABLE TO DESCRIBE THE FORMAT OF INPUT OR OUTPUT GIVEN A FORMATTED WRITE OR READ STATEMENT.
- BE ABLE TO WRITE A FORMATTED READ STATEMENT FOR A VARIABLE OF TYPE CHAR

FORMATTED READ STATEMENT

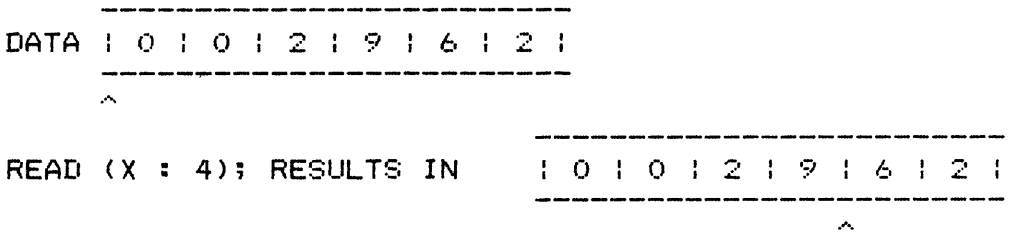
SYNTAX: READ (<FILE> , X : W)

- W IS THE FIELD WIDTH - X WILL BE READ FROM THE NEXT 'W' COLUMNS.

- FIELDS NEED NOT SEPARATED BY A BLANK

- REAL X OR INTEGER X
  A STRING OF 'W' CHARACTERS WHICH FORM THE NUMBER ARE READ
  BLANKS ARE READ AS ZEROS

EXAMPLE: FOR VAR X : INTEGER;

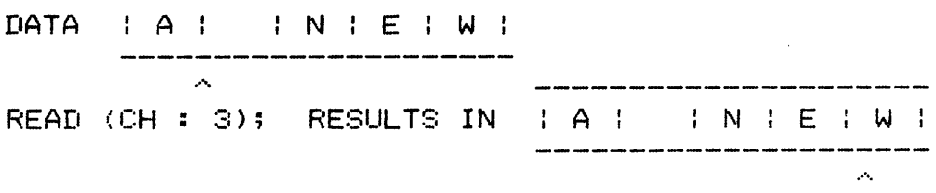


THE FIRST TIME THE READ COMMAND IS EXECUTED, X WILL BE ASSIGNED A VALUE OF 29.

- CHARACTERS:

  IF A FIELD WIDTH IS SPECIFIED
  - THE FIRST NON BLANK CHARACTER IS READ
  - THE FILE POINTER IS MOVED OVER THE SPECIFIED NUMBER OF COLUNMS

EXAMPLE: VAR CH : CHAR;



AND CH = 'N'

FORMATTED WRITE STATEMENT

SYNTAX : WRITE (FILE, X : M)

- M IS THE MINIMUM FIELD WIDTH
- THE VALUE X WILL BE WRITTEN WITH M CHARACTERS
  IF X REQUIRES LESS THAN M CHARACTERS THEN LEADING BLANKS ARE INSERTED.
  IF X REQUIRES MORE THAN M CHARACTERS THEN ADDITIONAL SPACE IS USED.
- REAL X WILL BE WRITTEN IN FLOATING POINT REPRESENTATION
- NO AUTOMATIC SPACING BETWEEN NUMBERS

EXAMPLES: VAR X, Y, Z : INTEGER; (X = 5, Y = 23, Z = 5062)

WRITE (X : 4, Y : 4); RESULTS IN THE FOLLOWING OUTPUT

___5__23

WRITE (X : 3, Z : 4); RESULTS IN

__55062

** NOTE: BLANKS MAY BE INSERTED BY USING A BLANK AS A TEXT STRING IN THE STATEMENT.

WRITE (X : 3, ' ' , Z : 4)

FIXED POINT NOTATION: X MUST BE REAL, FIXED, OR DECIMAL

SYNTAX : WRITE (FILE, X : M : N)

- M IS THE TOTAL NUMBER OF DIGITS INCLUDING THE DECIMAL
- N IS THE NUMBER OF DIGITS AFTER THE DECIMAL
- NO MORE PRECISION IS PRINTED THAN THE VALUE ACTUALLY CONTAINS

EXAMPLES :	X = 63.25	VALUE PRINTED
	WRITE (FILE, X:5:2)	63.25
	WRITE (FILE, X:4:1)	63.2
	WRITE (FILE, X:6:3)	63.25

** NOTE: IF M IS NOT LARGE ENOUGH TO INCLUDE THE INTEGRAL PART OF THE NUMBER, THE NUMBER WILL BE PRINTED IN FLOATING FORM.

X = 85.36951
WRITE (X : 7 : 5) RESULTS IN 8.536951E1

WORKSHEET 10

1. USING A FORMATTED WRITE STATEMENT, PRINT 2 INTEGERS, I1 AND I2, EACH HAVING 2 DIGITS. THE NUMBERS SHOULD BE SEPARATED BY A BLANK. WRITE TO THE DEFAULT FILE 'OUTPUT'.

2. GIVEN THE FOLLOWING CODE, SHOW THE OUTPUT:

```
VAR X, Y : INTEGER;
     .
     .
X := 5;
Y := 10;
WRITE (X);
WRITE (Y);
WRITELN;
WRITE (X : 4, Y : 4);
```

3. IN EACH CASE, SHOW THE OUTPUT IF WRITE (R : 5 : 2) EXECUTED:

   WHEN R := 67.3724

   WHEN R := 6.73724

   WHEN R := 6.73724E1

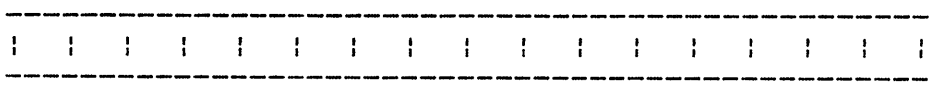
4. GIVEN THE DECLARATION VAR CHARVARIABLE : CHAR;
   AND THE FOLLOWING INPUT FILE:

```
+-----------------------------------------+
| A |   | C |H | R | I | S | T | M | A | S |
+-----------------------------------------+
    ^
```

   SHOW THE POSITION OF THE FILE POINTER AND THE VALUE OF CHARVARIABLE AFTER THE STATEMENT READ (CHARVARIABLE : 3) IS EXECUTED.

5. GIVEN THE DECLARATION VAR K : INTEGER;
   WRITE A FORMATTED READ STATEMENT TO READ ONE INTEGER OF 4 DIGITS FROM THE DEFAULT FILE 'INPUT'.

6. USING THE ABOVE READ STATEMENT, WRITE A SHORT LOOP TO READ AND 'PROCESS' 4 INTEGERS.
USE THE CHART BELOW TO SHOW THE FORMAT OF THE DATA.
THE FOUR NUMBERS YOU WISH TO READ ARE 1131, 211, 4337, 8940. ALL FOUR NUMBERS SHOULD APPEAR ON THE SAME LINE OF THE DATA FILE.



NESTED PROCEDURES AND FUNCTIONS

OBJECTIVES

STUDENTS SHOULD BE ABLE TO DECLARE AND UTILIZE NESTED PROCEDURES TO PERFORM SPECIFIED OPERATIONS ON A SET OF PARAMETERS.

STUDENTS SHOULD BE ABLE TO SPECIFY WHETHER A VARIABLE IS GLOBAL OR LOCAL TO A ROUTINE.

STUDENTS SHOULD BE ABLE TO SPECIFY WHICH ROUTINES IN A PROGRAM MAY CALL A SPECIFIED PROCEDURE OR FUNCTION AND WHEN A 'FORWARD' DECLARATION IS REQUIRED.

AGENDA

1. NESTED PROCEDURES
   - SYNTAX
   - PARAMETER PASSING
2. SCOPE OF VARIABLES
   - GLOBAL
   - LOCAL
3. PROCEDURE ACCESSING RULES
   - PROCEDURES AT THE SAME LEVEL
   - FORWARD DECLARATIONS
   - NESTED PROCEDURES
4. PASSING VARIABLY DIMENSIONED ARRAYS
5. COMMON AND GLOBAL VARIABLES
6. TOP DOWN DESIGN

PASCAL PROGRAM SYNTAX

```
      PROGRAM < PROG. NAME >;

          < LABEL DECLARATIONS >;
          < CONSTANT DECLARATIONS >;
          < TYPE DECLARATIONS >;
          < VARIABLE DECLARATIONS >;
          < COMMON DECLARATIONS >;
          < ACCESS DECLARATIONS >;
          < PROCEDURE AND FUNCTION DECLARATIONS >;

          BEGIN

              [ BODY OF PROGRAM  ( BLOCK ) ]

          END.
```

PASCAL PROCEDURE DECLARATION SYNTAX

```
     PROCEDURE < PROCEDURE NAME > [ ( < PARAMETER LIST > ) ];

          [ DECLARATIONS ]

          BEGIN

              [ BODY OF PROCEDURE ( BLOCK ) ]

          END;
```

**NOTE: EACH PROCEDURE HAS ITS OWN DECLARATION SECTION. THIS DECLARATION SECTION MAY CONTAIN PROCEDURE AND FUNCTION DECLARATIONS. THIS MEANS THAT A PROCEDURE MAY BE NESTED WITHIN ANOTHER PROCEDURE. PROCEDURES MAY BE NESTED 16 LEVELS DEEP. THE MAIN PROGRAM IS CONSIDERED LEVEL 1.

EXAMPLE

```
PROGRAM SCOPE1;

  VAR    X, Y : INTEGER;

  PROCEDURE ONE ( Q : INTEGER );

    VAR   A : INTEGER;

    PROCEDURE TWO ( R : INTEGER );

      VAR   B : INTEGER;

      BEGIN (* TWO *)
            .
            .              (*  X, Y, A, B  ARE ACCESSIBLE *)
            .
      END; (* TWO *)

    BEGIN (* ONE *)
          .
          .                (*  X, Y, A  ARE ACCESSIBLE *)
          .
    END; (* ONE *)

  PROCEDURE THREE ( S : INTEGER );

    VAR   C : INTEGER;

    BEGIN (* THREE *)
          .
          .                (*  X, Y, C  ARE ACCESSIBLE *)
          .
    END; (* THREE *)

  BEGIN (* SCOPE1 *)
        .
        .                  (*  X, Y  ARE ACCESSIBLE *)
        .
  END. (* SCOPE1 *)
```

NOTE: SPACE FOR LOCAL VARIABLES IS ALLOCATED WHEN THE PROCEDURE IS CALLED, AND THE SPACE IS RELEASED WHEN THE PROCEDURE IS EXITED.

DUPLICATING VARIABLE NAMES

**RULE: IF A VARIABLE IS DECLARED IN A PROCEDURE WHICH HAS THE SAME NAME AS A VARIABLE WHICH IS GLOBAL TO THAT PROCEDURE, THE NEWLY DECLARED VARIABLE WILL BE 'ACTIVE' UNTIL THE PROCEDURE IS EXITED.

EXAMPLE:

```
PROGRAM TEST;

   VAR I : INTEGER;

   PROCEDURE A ( .. );

      VAR I : REAL;

      PROCEDURE B ( .. );

         VAR I : BOOLEAN;

           * HERE I IS BOOLEAN


        * HERE I IS REAL


    * HERE I IS INTEGER
```

PROCEDURE ACCESSIBILITY RULES

** A PROGRAM OR PROCEDURE MAY CALL ITSELF

** A PROCEDURE MAY CALL ANOTHER PROCEDURE WHICH IS DEFINED WITHIN IT

** A PROCEDURE MAY CALL ANOTHER PROCEDURE AT THE SAME LEVEL IF IT HAS ALREADY BEEN DEFINED.

FORWARD DECLARATIONS

A PROCEDURE MUST BE 'DEFINED' (MUST HAVE BEEN DECLARED IN A PROCEDURE DECLARATION) BEFORE IT MAY BE CALLED.

EXAMPLE:

```
PROGRAM SCOPE;

   PROCEDURE A ( N : INTEGER );

         B ( ... )    *2*

   PROCEDURE B ( R : REAL );

         A ( ... )    *1*
```

1 THIS CALL TO 'A' IS LEGAL SINCE 'A' WAS DEFINED ABOVE (BEFORE) 'B'

2 THIS CALL TO 'B' IS NOT LEGAL SINCE 'B' HAS NOT BEEN DEFINED TO THE COMPILER WHEN THE CALL TO 'B' IS COMPILED

SOLUTION:

```
PROGRAM SCOPE;

   PROCEDURE B ( R : REAL ); FORWARD;    (*  1  *)

   PROCEDURE A ( N : INTEGER );

         B ( ... );    (*  2  *)

   PROCEDURE B;  (* NOTE THAT NO FORMAL PARMS ARE SPECIFIED *)

         A ( ... );
```

NOTES ON FORWARD DECLARATION

(* 1 *) - THE FORMAL (DUMMY) PARAMETER LIST SHOULD BE SPECIFIED ON THE FORWARD PROCEDURE DECLARATION

(* 2 *) - WHEN THE CALL TO 'B' IS MADE, 'B' HAS BEEN DEFINED IN THE FORWARD PROCEDURE DECLARATION

NESTED PROCEDURES

** A PROCEDURE MAY CALL ANY PROCEDURE ITS PARENT MAY CALL.

** A PROCEDURE MAY NOT CALL ANY PROCEDURE AT A LOWER LEVEL

```
PROGRAM T

    PROCEDURE A

        PROCEDURE B

    PROCEDURE C
```

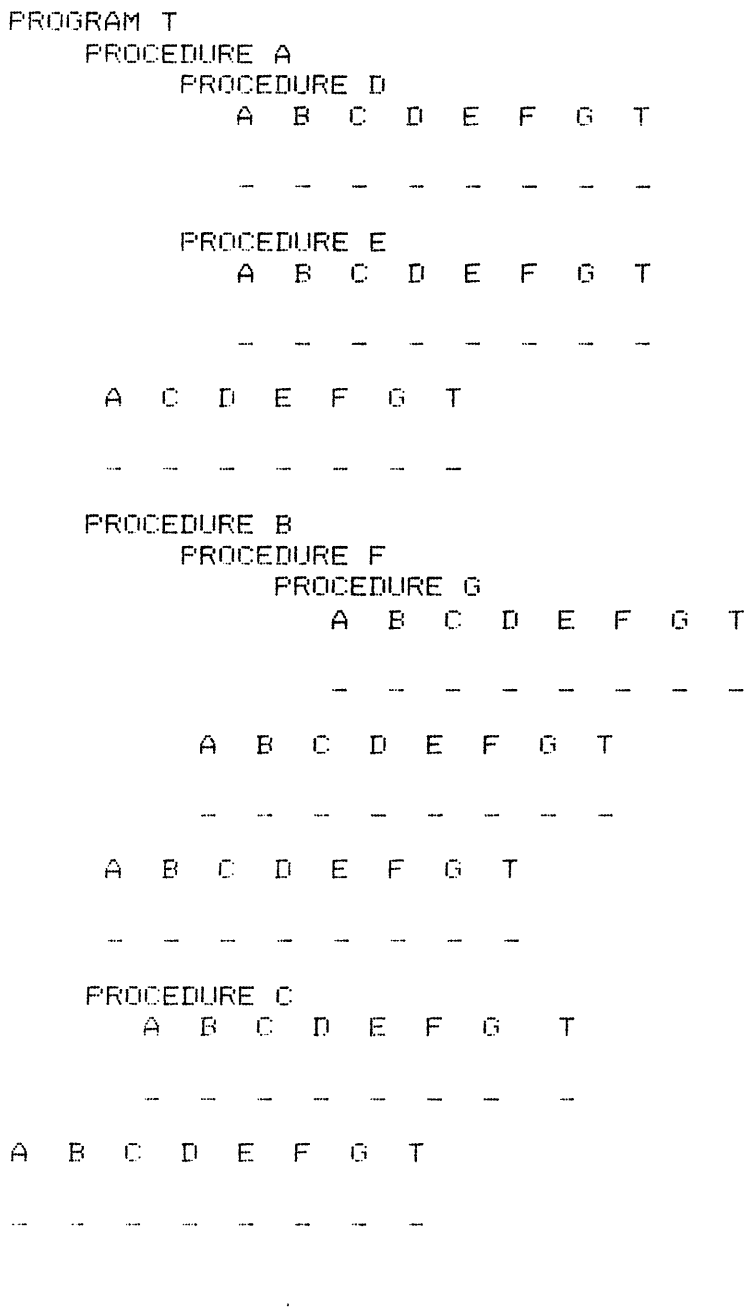
PROGRAM T MAY CALL: ITSELF, PROCEDURE A, PROCEDURE C

PROCEDURE A MAY CALL: ITSELF, PROGRAM T, PROCEDURE B, PROCEDURE C IF C IS FORWARD REFERENCED.

PROCEDURE B MAY CALL: ITSELF, PROCEDURE A, PROGRAM T, AND PROCEDURE C IF C IS FORWARD REFERENCED

PROCEDURE C MAY CALL: ITSELF, PROGRAM T, PROCEDURE A

EXERCISE


PROGRAM T
PROCEDURE A
PROCEDURE D
A B C D E F G T
PROCEDURE E
A B C D E F G T
A C D E F G T
PROCEDURE B
PROCEDURE F
PROCEDURE G
A B C D E F G T
A B C D E F G T
A B C D E F G T
PROCEDURE C
A B C D E F G T
A B C D E F G T

WORKSHEET 11

USE THE FOLLOWING INDICATORS TO SPECIFY WHETHER CALLS TO THE INDICATED PROCEDURES ARE LEGAL.

L - LEGAL, NO FORWARD DECLARATION REQUIRED
F - LEGAL, IF FORWARD DECLARATION IS REQUIRED
N - NOT LEGAL.

```
PROGRAM T
   PROCEDURE A

      A   B   C   D   E   F   G  T
1.
      -   -   -   -   -   -   -  -

   PROCEDURE B
      PROCEDURE C

         A   B   C   D   E   F   G  T
2.
         -   -   -   -   -   -   -  -

      PROCEDURE D

         A   B   C   D   E   F   G   T
3.
         -   -   -   -   -   -   -   -


      A   B   C   D   E   F   G   T
4.
      -   -   -   -   -   -   -   -

   PROCEDURE E
      PROCEDURE F
         PROCEDURE G

            A   B   C   D   E   F   G   T
5.
            -   -   -   -   -   -   -   -

         A   B   C   D   E   F   G   T
6.
         -   -   -   -   -   -   -   -

      A   B   C   D   E   F   G   T
7.
      -   -   -   -   -   -   -   -

   A   B   C   D   E   F   G   T
8.
   -   -   -   -   -   -   -   -
```

VARIABLE DIMENSION ARRAY PARAMETERS

PURPOSE: TO ALLOW A PROCEDURE TO OPERATE ON DIFFERENT SIZE ARRAYS

SYNTAX:

```
PROCEDURE TEST ( X : ARRAY [ 1..? ] OF INTEGER );
```

NOTES:

- THE LOWER BOUND INDEX MUST MATCH THAT OF THE ACTUAL PARAMETER
- THIS IS THE ONLY TIME WHEN A 'COMPLETE' TYPE DECLARATION IS ALLOWED FOR A FORMAL PARAMETER
- THE BUILT-IN PASCAL FUNCTION 'UB' RETURNS THE MAXIMUM INDEX FOR THE ACTUAL ARRAY

EXAMPLE:

```
PROGRAM TEST;

   VAR    A : ARRAY [ 1..10 ] OF INTEGER;
          B : ARRAY [ 1..20 ] OF INTEGER;
        MAX : INTEGER;

   PROCEDURE FINDMAX (       X : ARRAY [ 1..? ] OF INTEGER;
                         VAR BIG : INTEGER );

      BEGIN (* FINDMAX *)
         BIG := X[ 1 ];
         FOR I := 2 TO UB( X ) DO
            IF X[ I ] > BIG THEN
               BIG := X[ I ]
      END; (* FINDMAX *)

   BEGIN (* TEST *)

      FINDMAX ( A, MAX );

      FINDMAX ( B, MAX );

   END. (* TEST *)
```

FINDING THE UPPER BOUND OF MULTI-DIMENSIONED ARRAYS

UB (X , 1) RETURNS THE MAXIMUM INDEX OF THE FIRST DIMENSION OF ARRAY X

UB (X , 2) RETURNS THE MAXIMUM INDEX OF THE SECOND DIMENSION OF ARRAY X

EXAMPLE:

```
VAR ARAY : ARRAY [ 1..10, 1..20 ] OF REAL;

PROCEDURE ONE ( X : ARRAY [ 1..? , 1.. ? ] OF REAL );
  VAR M, N : INTEGER;
          .
          .
  M := UB ( X , 1 );      (* M = 10 *)

  N := UB ( X , 2 );      (* N = 20 *)
```

ALTERNATIVES TO PASSING PARAMETERS TO PROCEDURES

PASSING A VALUE TO A PROCEDURE AS A PARAMETER AIDS IN DEBUGGING
- TYPE CHECKING
- THE ABILITY TO PASS VAR'ED AND UNVAR'ED PARAMETERS
- ENHANCES THE READABILITY OF THE PROGRAM

THERE ARE SITUATIONS IN WHICH YOU MAY NOT WISH TO PASS A VALUE AS A PARAMETER
- BECAUSE OF SPACE CONSIDERATIONS
- YOU MAY WANT TO SHARE VARIABLES WITH OTHER PROGRAMS

COMMON VARIABLES

SYNTAX: COMMON <VARIABLE> : <TYPE>;

- FOLLOWS VAR DECLARATION SECTION
- VARIABLE NAMES NOT MORE THAN 6 CHARACTERS
- COMMON VARIABLES ARE NOT ALLOCATED ON THE STACK
- EXIST DURING ENTIRE EXECUTION OF THE PROGRAM
- USED TO DECLARE VARIABLES WHICH MAY BE SHARED WITH OTHER ROUTINES OR EXTERNAL ROUTINES.
- TO ACCESS A COMMON VARIABLE A ROUTINE MUST CONTAIN AN ACCESS DECLARATION

ACCESS DECLARATIONS

SYNTAX : ACCESS <VARIABLE>, <VARIABLE>, ... ;

- AN ACCESS DECLARATION MUST APPEAR IN EACH ROUTINE THAT ACCESSES A COMMON VARIABLE.

EXAMPLE:
```
PROCEDURE P ;
COMMON X : INTEGER;
    PROCEDURE Q;
    VAR Y : INTEGER ;
        PROCEDURE R ;
        ACCESS X ;
```

ACCESSING GLOBAL VARIABLES

- A ROUTINE MAY ACCESS ANY VARIABLE THAT IS 'GLOBAL' TO IT
- THE OPTION (*$ GLOBALS *) REQUIRES AN ACCESS DECLARATION FOR EACH GLOBAL VARIABLE USED IN A ROUTINE.

(*$ GLOBALS *)

- ONLY GLOBAL VARIABLES NAMES IN AN 'ACCESS' DECLARATION MAY BE USED BY A ROUTINE
- ROUTINE LEVEL OPTION
- DEFAULT : FALSE

TOP DOWN DESIGN

OBJECTIVE

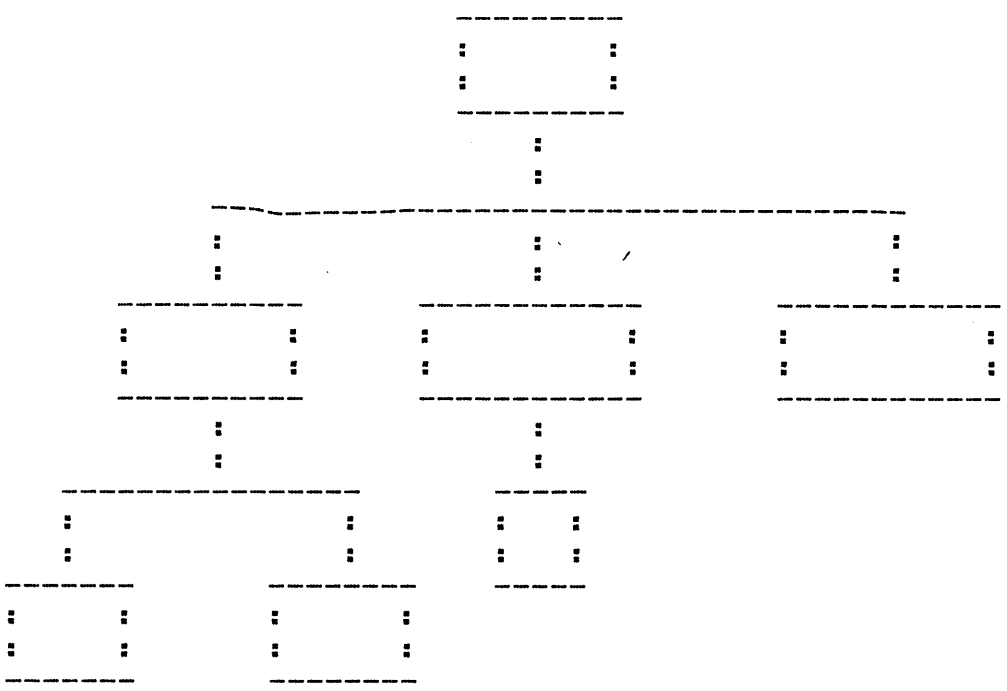
- THE GOAL IS TO EVENTUALLY OBTAIN A MODULAR DECOMPOSITION OF THE PROBLEM. FOR THIS TO BE EFFECTIVE, EACH MODULE SHOULD BE:
  - SMALL IN SIZE. USUALLY A MODULE SIZE OF 20 TO 200 LINES IS PREFERRED. THIS WILL DEPEND ON THE PROGRAMMING GROUP, THE LANGUAGE BEING USED AND THE PROBLEM. MODULES LARGER THAN THIS ARE DIFFICULT TO MANAGE.
  - COHESIVE. THE RELATION OF THE DIFFERENT PARTS OF THE SAME MODULE TO EACH OTHER SHOULD BE AS STRONG AS POSSIBLE. THE MODULE SHOULD BE RESPONSIBLE FOR A SINGLE FUNCTION, NOT A COLLECTION OF UNRELATED TASKS.
  - ISOLATED. THE INTERACTIONS BETWEEN DIFFERENT MODULES SHOULD BE AS WEAK AS POSSIBLE. IF THE MODULES ARE TRUELY FUNCTIONAL AND COHESIVE, THEY SHOULD BE ABLE TO INTERACT WITH OTHER MODULES OF THE PROGRAM IN VERY STRAIGHT FORWARD WAYS. IF POSSIBLE, INFORMATION SHOULD MOVE BETWEEN MODULES BY THE USE OF ARGUMENTS PASSED TO PROCEDURES AND FUNCTIONS. THE USE OF GLOBAL VARIABLES THAT ARE MODIFIED BY DIFFERENT MODULES OF A PROGRAM CAN LEAD TO INTERACTIONS BETWEEN MODULES THAT ARE DIFFICULT TO UNDERSTAND.

METHOD

- BEGIN WITH AN EXACT STATEMENT OF THE PROBLEM.
- INITIALLY, EXPRESS THE SOLUTION IN ENGLISH OR WHATEVER NOTATIN IS CONVENIENT.
- REFINE THE SOLUTION INTO SUCCESSIVELY MORE DETAILED LEVELS. POSTPONE DETAILS UNTIL THEY ARE NEEDED AT LOWER LEVELS.
- GIVE AS MUCH THOUGHT TO DATA STRUCTURES AS TO THE PROGRAM STRUCTURE.

TOP DOWN IMPLEMENTATION AND TESTING

TEST MAJOR INTERFACES FIRST, USING STUBS FOR LOW LEVEL MODULES.



ADVANTAGES:

- MAJOR INTERFACES ARE TESTED FIRST, SO MAJOR BUGS AND DESIGN WEAKNESSES ARE DISCOVERED FIRST.
- PRELIMINARY VERSION OF ENTIRE SYSTEM IS AVAILABLE EARLY.
- COMPUTER TIME SPENT ON TESTING IS DISTRIBUTED THROUGHOUT THE PROJECT.
- THE DEBUGGING EFFORT IS USUALLY CONCENTRATED ON THE LAST PROCEDURE ADDED TO THE PROGRAM. SINCE THE PROGRAM WORKED WHEN THE PROCEDURE WAS A STUB, MANY BUGS WILL BE FOUND QUICKLY SINCE MOST OF THEM WILL BE IN THE PROCEDURE JUST ADDED TO THE PROGRAM.

THE STUB

THE PROCEDURE STUB IS AN IMPORTANT PART OF TOP DOWN IMPLEMENTATION.
IN PASCAL THE STUB USUALLY LOOKS LIKE ONE OF THE EXAMPLES BELOW:

```
FUNCTION SQRT( X : REAL ): REAL;

(* FUNCTION TO TAKE THE SQUARE ROOT OF X *)

BEGIN(* SQRT *)
     (***** NOT YET IMPLEMENTED *****)
     SQRT := X/2;      (* RETURN A VALUE SO THE PROGRAM CAN BE RUN *)
END(* SQRT *);
```

- OR -

```
PROCEDURE LOOKUP( KEY : ALFA; VAR I : INTEGER );

(* TABLE LOOKUP ROUTINE, I IS LOCATION OF KEY IN THE TABLE *)

BEGIN(* LOOKUP *)
     (* -----PROCEDURE STUB----- *)
     I := 1;      (* RETURN SOME VALUE FOR THE PROGRAM TO USE *)
END(* LOOKUP *)
```

TO AID IN THE DEBUGGING AND OPTIMIZATION OF A LARGE PROGRAM, SMARTER
STUBS MAY BE USED. FOR EXAMPLE:

```
PROCEDURE INVERT( VAR A, B : ARRAY [ 1..?, 1..? ] OF REAL );

(* PROCEDURE TO INVERT MATRIX A AND PUT RESULT IN B *)

BEGIN(* INVERT *)
     (* >>>>>>> NOT YET IMPLEMENTED <<<<<<< *)
     WRITELN(' INVERT REACHED, INPUT MATRIX IS: ');
     FOR I:=1 TO UB( A, 1 ) DO
     BEGIN
         FOR J:=1 TO UB( A, 2 ) DO WRITE ( A[ I, J ]:10:5 );
         WRITELN;
     END (* FOR *);
     B := A;  (* RETURN AN ANSWER ( ALTHOUGH NOT THE RIGHT ONE )  *)
              (* FOR THE REST OF THE PROGRAM TO WORK WITH         *)
END(* INVERT *);
```

PASCAL MEMORY ALLOCATION

OBJECTIVE

STUDENTS SHOULD BE ABLE TO FIND THE STORAGE REQUIREMENTS FOR A PARTICULAR DATA TYPE IN THE 990 PASCAL DOCUMENTATION.

AGENDA

1. STACK AND HEAP ALLOCATION
2. DATA TYPES AND MEMORY USAGE

PASCAL MEMORY ALLOCATON

MEMORY FOR VARIABLES IS STORED IN TWO MEMORY AREAS:

- STACK
- HEAP

STACK MEMORY

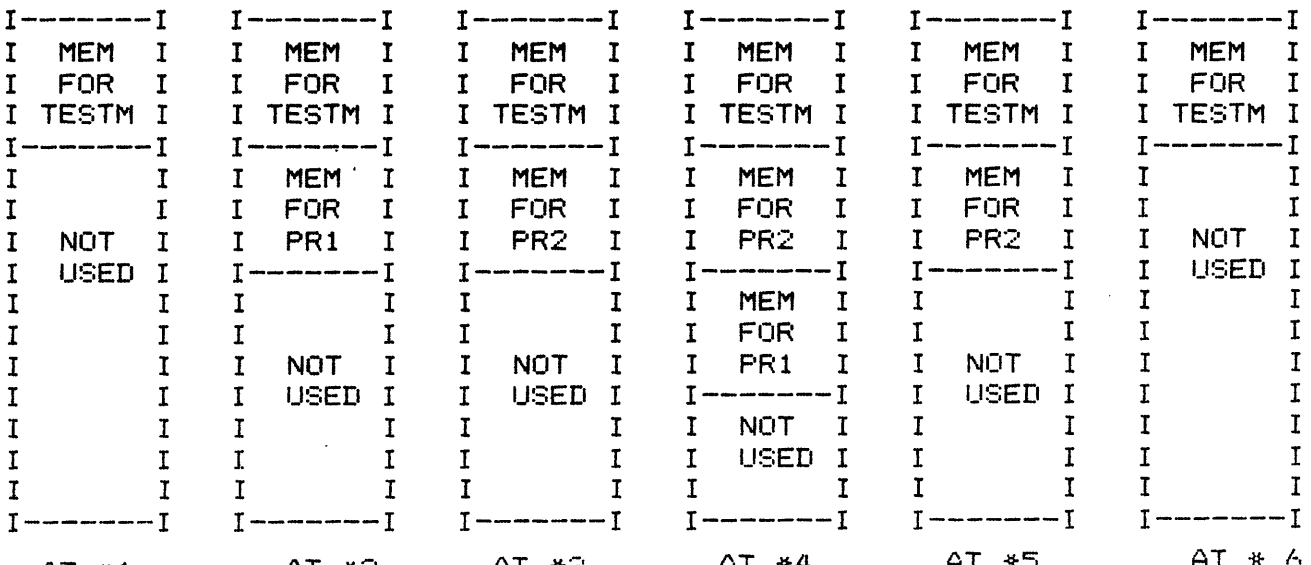
- ALL VARIABLES WHICH ARE DECLARED IN A PROGRAM (STATIC VARIABLES) ARE ALLOCATED MEMORY FROM STACK MEMORY.
- STORAGE LOCATIONS FOR A ROUTINE ARE NOT ALLOCATED UNTIL THE ROUTINE IS CALLED.
- WHEN A ROUTINE IS EXITED, STORAGE FOR THAT ROUTINE IS RELEASED.

HEAP MEMORY

- ALL VARIABLES ALLOCATED USING THE 'NEW' PASCAL FUNCTION CALL (DYNAMIC VARIABLES) HAVE THEIR STORAGE LOCATIONS IN 'HEAP' MEMORY.
- HEAP MEMORY IS A MEMORY 'POOL' WHICH MAY BE USED BY YOUR PROGRAM TO ALLOCATE TEMPORARY MEMORY.
- IF THE SIZE AND NUMBER OF YOUR VARIABLES ARE KNOWN WHEN WRITING YOUR PROGRAM, YOU SHOULD DECLARE THEM IN THE PROGRAM.

```
     PROGRAM TESTM;

          [ TESTM VARIABLE DECLARATIONS ]

          PROCEDURE PR1

               [ PR1 VARIABLE DECLARATIONS ]

               BEGIN (* PR1 *)

               END (* PR1 *)

          PROCEDURE PR2

               [ PR2 VARIABLE DECLARATIONS ]

               BEGIN (* PR2 *)

*4 ------------------ CALL TO PR1

*5 ------------------

               END; (* PR2 *)

          BEGIN (* TESTM *)

*1 ----------

*2 ---------- CALL TO PR1

*3 ---------- CALL TO PR2

*6 ----------

          END.  (* TESTM *)
```

STACK MEMORY



```
PROGRAM WRONG;

   PROCEDURE COUNT ( FLAG : BOOLEAN );
     VAR N : INTEGER;
     BEGIN (* COUNT *)
        IF FLAG THEN N := 0;
        N :] N + 1;
        WRITELN ( 'N IS ', N )
     END; (* COUNT *)

   PROCEDURE CNT ( FLAG : BOOLEAN );
     VAR M : INTEGER;
     BEGIN  (* CNT *)
        IF FLAG THEN M := 0;
        M := M + 1;
        WRITELN ( 'M IS ', M )
     END;

   BEGIN (* MAIN *)
     WRITELN ( 'WRONG EXECUTION BEGINS');
     WRITELN;
     COUNT ( TRUE );
     CNT   ( TRUE );
     COUNT ( FALSE );
     CNT   ( FALSE );
     COUNT ( FALSE );
     CNT   ( FALSE );
     COUNT ( FALSE );
     CNT   ( FALSE );
     COUNT ( TRUE );
     CNT   ( TRUE );
   END.
```

```
WRONG EXECUTION BEGINS

N IS          1
M IS          1
N IS          2
M IS          3
N IS          4
M IS          5
N IS          6
M IS          7
N IS          1
M IS          1
```

DATA TYPE STORAGE REQUIREMENTS

TYPE	STORAGE UNPACKED	STORAGE PACKED
INTEGER	2 BYTES	2 BYTES
LONGINT	4 BYTES	4 BYTES
REAL	4 BYTES	4 BYTES
REAL (N)		
N <= 7	4 BYTES	4 BYTES
N >= 8	8 BYTES	8 BYTES
BOOLEAN	2 BYTES	1 BIT
POINTER	2 BYTES	2 BYTES

*SEE PARAGRAPH 6.7.3 OF PASCAL USER MANUAL FOR OTHER TYPES.

ARRAY STORAGE REQUIREMENTS

UNPACKED:

N = NUMBER OF ELEMENTS
S = ELEMENT SIZE UNPACKED
STORAGE = N*S BYTES

PACKED:

*NOTE: A SINGLE ELEMENT OF AN ARRAY WILL NEVER BE SPLIT ACROSS WORD BOUNDARIES.

N = NUMBER OF ELEMENTS
S = SIZE OF EACH ELEMENT IN BITS

S < 16

MAXIMUM # OF ELEMENTS ARE PACKED IN EACH WORD

E = # OF ELEMENTS / WORD
Q = 0 - IF N MOD E = 0
1 - IF N MOD E > 0

STORAGE = (N DIV E) + Q WORDS

EXAMPLE:

```
VAR TEST : PACKED ARRAY [1..11] OF CHAR
```

STORAGE = 11 DIV 2 + 1 = 6 WORDS

S > 16

E = # OF WORDS/ELEMENT

STORAGE = N*E WORDS

RECORDS

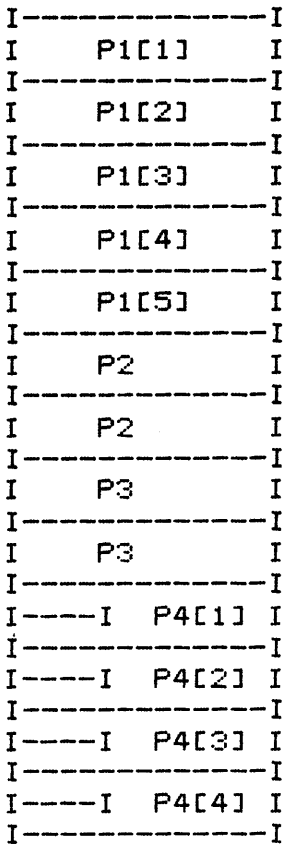
UNPACKED:

STORAGE = SUM OF STORAGES REQUIRED BY EACH COMPONENT.

EXAMPLE:

```
TYPE REC : RECORD
             P1 : ARRAY [ 1..5 ] OF INTEGER;
             P2 : REAL;
             P3 : LONGINT;
             P4 : ARRAY [ 1..4 ] OF CHAR
           END;
```

A VARIABLE OF TYPE 'REC' REQUIRES 13 WORDS



PACKED RECORDS

```
EXAMPLE :   TYPE R = PACKED RECORD
                        A : PACKED ARRAY [1..10] OF 1..31;
                        J : 0..7;
                        K : 0..#FFF;
                        L : INTEGER
                    END;
```

RECORD STORAGE RULES :

1. IF RECORD IS NOT PACKED THEN A FIELD OCCUPIES THE NUMBER OF WORDS REQUIRED BY ITS TYPE.

2. IF THE PRECEEDING FIELD IS LESS THAN 1 WORD, THEN THE NEXT FIELD IS STORED IN THE REMAINDER OF THE WORD IF IT WILL FIT.

3. IF IT DOESN'T FIT THEN IT IS LEFT JUSTIFIED AT THE BEGINNING OF THE NEXT WORD AND THE PREVIOUS FIELD IS RIGHT JUSTIFIED IN THE PREVIOUS WORD.

4. IF THE PRECEEDING FIELD OCCUPIES MORE THAN 1 WORD THEN THE NEXT FIELD IS STORED LEFT JUSTIFIED IN THE NEXT WORD.

```
TYPE REC = PACKED RECORD
             A : PACKED ARRAY [1..10] OF 1..31;
             J : 0..7;
             K : 0..#FFF;
             L : INTEGER
           END;
     R   = PACKED RECORD
             J : 0..7;
             A : PACKED ARRAY [1..10] OF 0..31
           END;
```

STORAGE FOR R

				J
A[1]	A[2]	A[3]		
A[4]	A[5]	A[6]		
A[7]	A[8]	A[9]		
A[10]				

STORAGE FOR REC

ASSIGNMENT 2

OBJECTIVES

- TO BE ABLE TO USE PROCEDURES TO HELP MODULARIZE YOUR PROGRAMS.
- TO BE ABLE TO MANIPULATE ARRAYS AND RECORDS
- TO BE ABLE TO USE THE CONTROL STRUCTURES [illegible] TO CODE A PASCAL PROGRAM.
- TO BE ABLE TO READ IN AND WRITE OUT ARRAYS AND RECORDS.

DATA: TI.PASCAL.DATA.DATA2
THE DATA FOR BOTH PARTS OF ASSIGNMENT 2 IS A LIST OF 10 EMPLOYEES AND THEIR SOCIAL SECURITIES. THE FORMAT IS A SOCIAL SECURITY NUMBER (9 DIGITS) FOLLOWED BY A SPACE, FOLLOWED BY A NAME (UP TO 20 DIGITS).

2A. READ EACH NAME AND SOCIAL SECURITY NUMBER INTO AN ARRAY OF RECORDS. ECHO PRINT EACH NAME AND NUMBER AS IT IS READ IN. AFTER EACH NAME AND NUMBER HAS BEEN STORED IN THE ARRAY, PRINT OUT THE RECORD TO BE SURE IT HAS BEEN STORED CORRECTLY.

STRATEGY:
- CREATE A RECORD TYPE WHICH HAS THE FOLLOWING FIELDS
  SOCIAL SECURITY NUMBER (9 DIGITS): THIS MUST BE OF TYPE LONGINT
  NAME (UP TO 20 CHARACTERS): THIS SHOULD BE A PACKED ARRAY OF TYPE CHAR

- CREATE A TYPE 'A' WHICH IS AN ARRAY OF RECORDS
- DECLARE A GLOBAL VARIABLE OF TYPE A.
- LET THE LENGTH OF THE ARRAY BE A CONSTANT
- WRITE THE FOLLOWING PROCEDURES:

READ_ECHO
- READS THE SOCIAL SECURITY NUMBER AND NAME INTO THE RECORD IN THE ARRAY.
- ECHO PRINTS THE NAME AND NUMBER AS THEY ARE READ IN.
  - TO ECHO THE NAME, WRITE EACH CHARACTER OUT AS IT IS READ IN.
- PARAMETERS
  ARRAY NAME
  ARRAY LENGTH

- THE MAIN ROUTINE WILL RESET THE DATA FILE AND CALL READ_ECHO.

STRUCTURE FOR PROGRAM 2A:

2B. ADD THE FOLLOWING TO YOUR PROGRAM 2A

PRINT_ARRAY : - A PROCEDURE WHICH WRITES OUT THE ELEMNETS OF THE ARRAY ONE NAME AND NUMBER PER LINE.

- WRITE OUT A MESSAGE TO LABEL THE LIST SUCH AS ' EMPLOYEE LIST '

- PARAMETERS:
  ARRAY NAME
  NUMBER OF ELEMENTS

- MODIFY YOUR MAIN PROGRAM TO CALL PRINT_ARRAY

STRUCTURE FOR PROGRAM 2B

DATA STRUCTURES - POINTERS

OBJECTIVES:

- BE ABLE TO DECLARE POINTERS AND RECORDS CONTAINING POINTERS.
- BE ABLE TO USE 'NEW' TO ALLOCATE A RECORD POINTED TO BY A GIVEN POINTER.
- BE ABLE TO WRITE STATEMENTS MANIPULATING POINTERS:
  ASSIGN VALUES
  ASSIGN VALUES TO A FIELD IN A RECORD POINTED TO BY A POINTER
- BE ABLE TO DRAW A DIAGRAM INDICATING HOW A POINTER IS MOVED WHEN A GIVEN STATEMENT IS EXECUTED.

AGENDA

1. USING POINTERS
   - NIL
   - NEW
   - LINKED LISTS
   - DOUBLY LINKED LISTS

WORKSHEET

POINTERS

DEFINITION: A POINTER IS A VARIABLE THAT HOLDS THE ADDRESS OF (OR "POINTS TO") A BLOCK OF MEMORY.

SYNTAX: VAR <NAME> : @<TYPE>;

OR

TYPE <TYPE NAME> : @<TYPE>;

A POINTER IS DECLARED AS POINTING TO A SPECIFIC TYPE OF DATA. IT MAY ONLY POINT TO THAT TYPE.

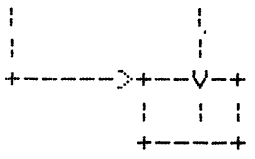
USING POINTERS:

```
TYPE DATA = RECORD
              INT : INTEGER;
              PNTR : @DATA;
            END;
VAR P, Q : @DATA;
```

P	REFERENCES THE VALUE OF P OR THE ADDRESS OF A BLOCK OF 'DATA'.
P := Q	MAKES P POINT TO THE SAME PLACE AS Q
P@	REFERENCES THE BLOCK OF DATA POINTED TO BY P
P@ := Q@	COPIES THE DATA POINTED TO BY Q INTO THE BLOCK POINTED TO BY P.

IF P POINTS TO A RECORD, P@.<FIELD> REFERENCES A FIELD IN THAT RECORD.

P := Q

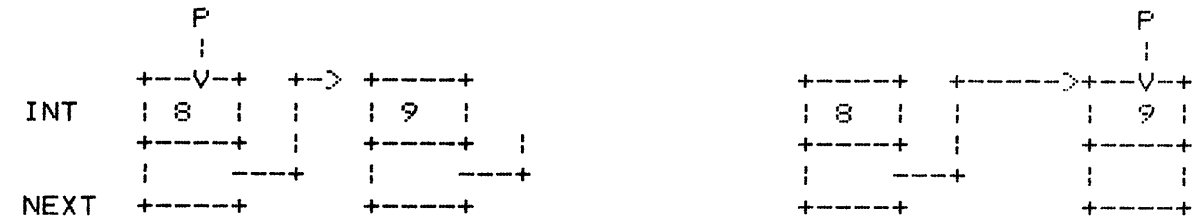


P@ := Q@

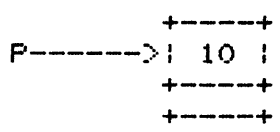
P@.INT := 4;

Q@.INT := 4;

```
EXAMPLE:   TYPE PNTR = @BLOCK;
           TYPE BLOCK = RECORD
                          INT : INTEGER;
                          NEXT : @BLOCK
                        END;

           VAR P : PNTR;      (* P IS A POINTER TO TYPE BLOCK *)

           P := P@.NEXT HAS THIS EFFECT:
```



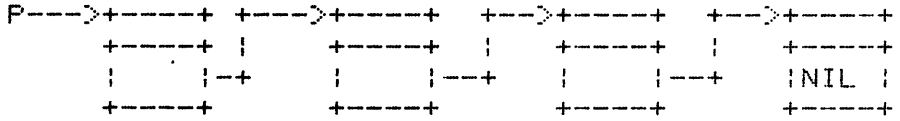

```
           P@.INT := 10   HAS THIS EFFECT:
```




```
 **NOTE:  A FORWARD DECLARATION IS ALLOWED HERE.
```

NIL

```
DEFINITION:  NIL IS A SPECIAL VALUE THAT ALLOWS A POINTER TO POINT
             TO NOTHING.

             A POINTER MAY BE ASSIGNED TO OR COMPARED WITH NIL.

                  P@.NEXT := NIL;     IF P@.NEXT = NIL THEN


             IF A POINTER P IS NIL THEN REFERENCING P@ WILL GIVE
             AN ERROR.

             NIL IS USED TO MARK THE END OF A CHAIN.
```




```
RUNTIME OPTION  CKPTR
---------------------

   -  ENABLES OR DISABLES CHECKING FOR NIL VALUES OF VARIABLES OF
      TYPE POINTER AT EXECUTION TIME.

   -  DEFAULT IS FALSE
```

NEW (P)

DEFINITION: ALLOCATES A BLOCK OF MEMORY OF THE TYPE P POINTS TO.
P IS SET TO POINT TO THE NEW BLOCK.

```
EXAMPLE:  TYPE REC = RECORD
                       ITEM : ARRAY[1..10] OF CHAR;
                       NEXT : @REC
                     END;

          VAR WORD : @REC;     (* WORD IS A POINTER TO TYPE REC *)
                     .
                     .
          NEW( WORD );         (* CREATES A RECORD AND SETS WORD TO
                                  POINT TO IT *)
          WORD@.NEXT := NIL;   (* SET THE POINTER FIELD TO NIL *)
```

**NOTE: WHEN A NEW BLOCK IS ALLOCATED, SET ALL POINTERS TO NIL.
THIS IS NOT DONE FOR YOU.

DISPOSE (P)

DEFINITION: MAKES THE SPACE POINTED TO BY P AVAILABLE FOR REUSE.

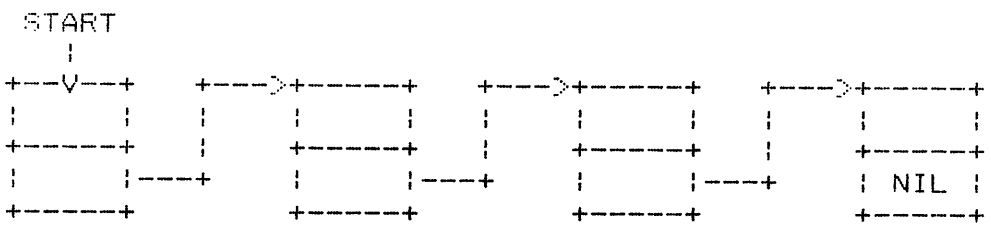
- P MUST POINT TO A DYNAMIC VARIABLE
  I.E. ONE ALLOCATED BY NEW

- IF P = NIL AN ERROR OCCURS

- AFTER THE STORAGE IS DEALLOCATED, P IS SET TO NIL.

LINKED LISTS

DEFINITION: A LINKED LIST IS A CHAIN OF RECORDS JOINED BY POINTERS.



EXAMPLE : SEARCH A LINKED LIST FOR THE VALUE NUM.

```
VAR PNTR : LINK;

PNTR := FIRST      (* SET PNTR TO THE FIRST OF THE LIST *)
                   (* FIRST COULD BE PASSED AS A        *)
                   (* PARAMETER                         *)

WHILE PNTR@.SS <> NUM DO

  PNTR := PNTR@.NEXT;
```

EXAMPLE: SEARCH WITH THE POSSIBILITY OF AN UNSUCCESSFUL SEARCH.

```
PNTR := FIRST;

WHILE (PNTR <> NIL) AND (PNTR@.SS <> NUM) DO

  PNTR := PNTR@.NEXT;
```

THE POINTER OF THE LAST RECORD WILL POINT TO NIL.

```
PROCEDURE READ_ECHO ( VAR WORD : PTR );

VAR I  : INTEGER;
    CH : CHAR;
BEGIN
   I := 0;
   REPEAT
      I := I + 1;
      READ ( CH );
      WRITE ( CH );
      WORD@.A [ I ] := CH
   UNTIL CH = ' ' OR EOLN;
   IF EOLN THEN
      BEGIN
        READLN;
        WRITELN
      END
   END; (* READ_ECNO *)
```

READ N SOCIAL SECURITY NUMBERS INTO A LINKED LIST

```
TYPE REC = RECORD
              SS : LONGINT;
              NEXT : @REC
            END;
     LINK = @REC;

VAR  FIRST : LINK;

PROCEDURE READ_SOCIAL_SECURITY_NUMBERS( VAR FIRST : LINK );

VAR  S, N, I : INTEGER;
     P       : LINK   ;

BEGIN
   RESET ( INPUT );
   READ ( N );
   FIRST := NIL;
   FOR I := 1 TO N DO
     BEGIN
        READ ( S );          (* READ THE SOCIAL SECURITY NUMBER *)
        NEW ( P );           (* P POINTS TO A NEW RECORD        *)
        P@.SS := S;          (* PUT SOCIAL SECURITY NUMBER INTO *)
                             (* NEW RECORD                      *)
        P@.NEXT := FIRST;
        FIRST := P
END;
```

STACKS

DEFINITION: A STACK IS A LIST IN WHICH ALL INSERTIONS AND DELETIONS ARE MADE FROM THE TOP.

ADD THE ELEMENTS A, B, C TO A STACK:

C TOP

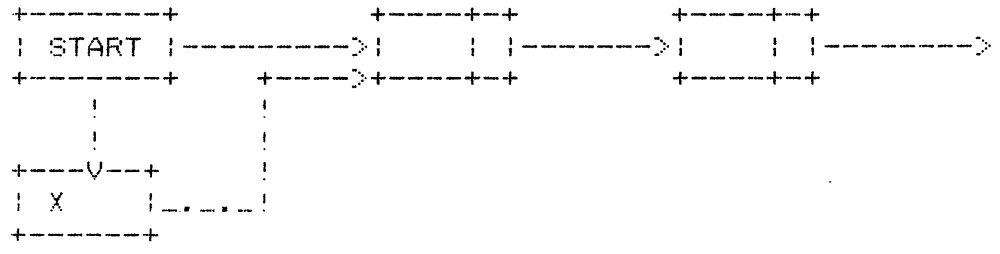
B

A

THE RESTRICTIONS ON A STACK IMPLY THAT IF A, B, AND C ARE ADDED TO A STACK IN THAT ORDER THEN THE FIRST ELEMENT TO BE REMOVED WILL BE C.

EXAMPLE: DECLARATIONS ARE:

```
TYPE BLOCK = RECORD
               DATA : T;
               LINK : @BLOCK
             END;
VAR START: @BLOCK;

PROCEDURE INITIALIZE_STACK;
BEGIN
  START := NIL;
END; (* INITIALIZE_STACK *)


PROCEDURE PUSH(X : T);
VAR P : @BLOCK;
BEGIN
  NEW(P);
  P@.DATA := X;      (* PUT X INTO RECORD *)
  P@.LINK := START;(* INSERT RECORD AT FRONT OF *)
                     (* THE STACK                *)

  START := P         (* MAKE START POINT TO THE FRONT *)
                     (* OF THE LIST                   *)
END; (* PUSH *)
```

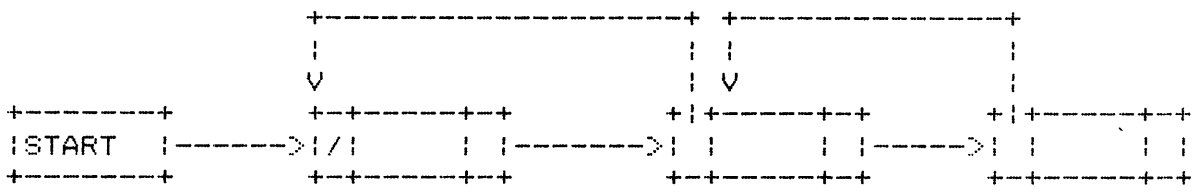


DOUBLY LINKED LISTS

DEFINITION: A DOUBLY LINKED LIST HAS 2 POINTERS -- ONE TO THE NEXT BLOCK OF DATA, AND ONE TO THE PREVIOUS BLOCK OF DATA.

EXAMPLE:

```
TYPE BLOCK = RECORD
               NEXT : @BLOCK;
               DATA : INTEGER;
               BACK : @BLOCK
             END;
```



* DOUBLY LINKED LISTS SIMPLIFY INSERTING AND DELETING BECAUSE THERE IS NO NEED TO KEEP A 'BACK' POINTER.

* DOUBLY LINKED LISTS TAKE UP MORE SPACE THAN SINGLY LINKED LISTS.

SUMMARY

- A POINTER IS A VARIABLE THAT HOLDS THE ADDRESS OF A BLOCK OF MEMORY. ONCE A POINTER IS DECLARED AS POINTING TO A PARTICULAR TYPE OF DATA, IT MAY NOT POINT TO ANY OTHER TYPE.

- NIL IS A SPECIAL VALUE THAT ALLOWS A POINTER TO POINT TO NOTHING. NIL IS USED TO MARK THE END OF A LINKED LIST.

- NEW IS A FUNCTION WHICH ALLOCATES A NEW BLOCK OF DATA FOR A POINTER TO POINT TO.

- A LINKED LIST IS A CHAIN OF RECORDS JOINED BY POINTERS.

- A DOUBLY LINKED LIST HAS 2 POINTERS--ONE TO THE NEXT BLOCK AND ONE TO THE PREVIOUS BLOCK.

WORKSHEET 12

1. WRITE THE DECLARATION STATEMENTS FOR THE FOLLOWING:

   DECLARE A TYPE 'BLOCK' WHICH IS A RECORD WITH 2 FIELDS. ONE FIELD IS AN INTEGER AND THE OTHER IS A POINTER TO TYPE BLOCK.

2. DECLARE VARIABLES P1 AND P2 WHICH POINT TO TYPE BLOCK.

RITE STATEMENTS WHICH DO THE FOLLOWING:

   ASSUME P1 AND P2 ARE POINTERS. P1 POINTS TO A BLOCK OF DATA. WRITE AN ASSIGNMENT STATEMENT TO MAKE P2 POINT TO THE SAME RECORD AS P1.

   ASSIGN A VALUE OF 10 TO THE INTEGER FIELD OF THE BLOCK POINTED TO BY P1.

. ALLOCATE A NEW RECORD POINTED TO BY P2.

6. SET THE POINTER FIELD OF THE NEW RECORD TO NIL.

7. COPY THE DATA FROM P1'S INTEGER FIELD INTO P2'S INTEGER FIELD.

8. SET THE POINTER FIELD OF P1'S RECORD TO POINT TO P2'S RECORD.

9. DRAW A PICTURE OF P1 AND P2 BEFORE AND AFTER THE STATEMENT FOR QUESTION 8 IS EXECUTED. ASSUME THAT INITIALLY BOTH POINTER FIELDS ARE NIL.

10. ASSUME TYPE REC = RECORD
                        DATA : REAL;
                        PTR : @REC
                      END;
               VAR Q : @REC;

DRAW A PICTURE OF THE POINTER Q AFTER EXECUTING NEW(Q).

11. DRAW A PICTURE OF A LINKED LIST OF 4 RECORDS OF TYPE REC ABOVE. THE POINTER START POINTS TO THE FIRST OF THE LIST. THE POINTER FIELD OF THE LAST RECORD IS NIL.

12. GIVEN THE FOLLOWING DECLARATIONS:

```
TYPE BLOCK = RECORD
                DATA : T;
                LINK : @BLOCK
             END;
VAR START : @BLOCK;
```

WRITE A PROCEDURE POP(X : T) WHICH WILL "POP" THE VALUE X OFF OF A STACK POINTED TO BY START. IF THE STACK IS NIL, CALL A PROCEDURE 'UNDERFLOW'.

APPROACH: TEST FOR NIL START (UNDERFLOW)
IF START IS NOT NIL THEN ASSIGN X THE VALUE OF THE
THE DATA FIELD.
MAKE START POINT TO THE NEXT RECORD IN THE LIST
(THE OTHER RECORD WILL STILL BE THERE BUT IT WILL
NOT BE ACCESSIBLE AS PART OF THE LST)

13. DECLARE A RECORD DLIST FOR A DOUBLY LINKED LIST. THE DATA FIELD IS INTEGER.

14. INDICATE ON THE DIAGRAM HOW THE POINTERS MUST BE MANIPULATED TO INSERT P'S BLOCK AT THE FRONT OF THE LIST.

SETS AND TYPE TRANSFERS

OBJECTIVES:

THE STUDENT SHOULD BE ABLE TO DECLARE A SET AND MANIPULATE SETS USING THE SET OPERATORS.

THE STUDENT SHOULD BE ABLE TO CHANGE THE TYPE OF A VARIABLE BY USING THE TYPE TRANSFER STATEMENT.

SETS

SYNTAX: TYPE < IDENTIFIER > := SET OF < BASE TYPE >

< BASE TYPE > MAY BE : ENUMERATION TYPE
DYNAMIC SET TYPE

MAY NOT INCLUDE NEGATIVE INTEGERS

MAXIMUM OF 1032 ELEMENTS

```
EXAMPLES : TYPE COLOR = (RED, ORANGE, YELLOW, GREEN, BLUE, VIOLET);
           MIX    = SET OF COLOR;
           STUDENTNAME = (JOE, JOHN, JIM, JANE, LES, MARTIN);

         VAR  INT, WORD, VOWELS : SET OF 'A'..'Z';
           SHADE : SET OF YELLOW .. BLUE;
           CLASS : SET OF STUDENTNAME;
           COURSE : ARRAY [ COURSENUMBER ] OF SET OF STUDENTNAME;
           HUE    : MIX;
```

CONSTRUCTING A SET:

```
HUE := [ YELLOW, GREEN ];
INT := [ 'I' .. 'N' ];
COURSE [ I ] := [ JIM, JILL ];
```

[] DENOTES THE EMPTY SET
IF M > N THEN [M..N] DENOTES THE EMPTY SET

SET OPERATORS:

+	UNION
*	INTERSECTION
-	SET DIFFERENCE
=	EQUALITY
<>	INEQUALITY
<=	SET INCLUSION
>=	SET INCLUSION
<	PROPER SET INCLUSION
>	PROPER SET INCLUSION
IN	SET MEMBERSHIP

EXAMPLES:

```
VOWELS := [ 'A', 'E', 'I', 'O', 'U' ];
WORD := [];
READ (CH);
WHILE NOT (CH IN [ ' ', ',', '.']) DO
   BEGIN
      WORD := WORD + [ CH ];
      READ (CH)
   END;
IF WORD * VOWELS = [] THEN WRITELN (' NO VOWELS');
IF WORD <= VOWELS THEN WRITELN(' ONLY VOWELS')


TYPE BIT3 = SET OF 0..2;
VAR  MASK1, MASK2, MASK3, SETA, SETB : BIT3;

MASK1 := [0];
MASK2 := [1];
MASK3 := [0,1];

(* MASKING BY INTERSECTION *)

SETB := SETA * MASK2;

(* EXCLUSIVE OR *)

SETB := SETA + SETB - SETA * SETB
```

**NOTES: BECAUSE OF OPERATOR PRECEDENCE, SET EXPRESSIONS MAY HAVE TO BE PARENTHESIZED TO PRODUCE THE DESIRED RESULTS.

A + B * C = A + (B * C)

ITERATION OVER SETS

FOR < CONTROL VARIABLE > IN < SET EXPRESSION > DO < STATEMENT >

EXAMPLE: FOR I IN [1, 3, 5, 7, 4, 6, 10] DO < STATEMENT >

```
VAR ALLERRORS : SET OF 1..400;
                .
                .
FOR I IN ALLERRORS DO..
```

DYNAMIC SETS

```
EXAMPLE:   TYPE COLORS   = ( RED, YELLOW, GREEN, ORANGE, PINK );
           COLORSET = SET OF COLORS;
           VAR  SHADE    : SET OF GREEN .. UB ( COLORSET );
```

PROGRAM EXAMPLES

```
PROGRAM SETOP;
(* EXAMPLE OF SET OPERATIONS *)
TYPE DAYS = (M, T, W, TH, FR, SA, SU);
     WEEK = SET OF DAYS;
VAR  WK, WORK, FREE : WEEK;
     D              : DAYS;

PROCEDURE CHECK ( S : WEEK);
VAR D : DAYS;
BEGIN
  WRITE (' ');
  FOR D := M TO SU DO
    IF D IN S THEN WRITE ('X') ELSE WRITE ('O');
  WRITELN
END; (* CHECK *)

BEGIN (* SETOP *)
  WORK := []; FREE := [];
  WK := [M..SU];
  D := SA;
  FREE := [D] + FREE + [SU];
  CHECK ( FREE );
  WORK := WK - FREE;
  CHECK ( WORK );
  IF FREE <= WK THEN WRITE ('O');
  IF NOT ( WORK >= FREE ) THEN WRITE (' JACK');
  IF [SA] <= WORK THEN WRITE (' FORGET IT');
  WRITELN
END. (* SETOP *)
```

PRIME NUMBER SEIVE

1. PUT ALL THE NUMBERS BETWEEN 2 AND N INTO THE "SEIVE"

2. SELECT AND REMOVE THE SMALLEST NUMBER REMAINING IN THE SEIVE.

3. INCLUDE THIS NUMBER IN THE "PRIMES".

4. STEP THROUGH THE SEIVE, REMOVING ALL MULTIPLES OF THIS NUMBER.

5. IF THE SEIVE IS NOT EMPTY, REPEAT STEPS 2--5.

```
CONST N = 1000;
VAR SEIVE, PRIMES : SET OF 2..N;
    NEXT, J       : INTEGER;

BEGIN  (* INITIALIZE *)
  SEIVE := [2..N];
  PRIMES := [];
  NEXT := 2;
  REPEAT (* FIND THE NEXT PRIME *)
    WHILE NOT (NEXT IN SEIVE) DO NEXT := SUCC ( NEXT );
    PRIMES := PRIMES + [NEXT];
    J := NEXT;
    WHILE J <= N DO (* ELIMINATE MULTIPLES OF NEXT *)
      BEGIN
        SEIVE := SEIVE - [J];
        J := J + NEXT
      END
  UNTIL SEIVE = []
END.
```

TYPE TRANSFER

TYPE TRANSFER IS A MEANS OF TEMPORARILY CHANGING THE TYPE OF AN EXISTING VARIABLE.

SYNTAX: < VARIABLE > :: < TYPE IDENTIFIER >

EXAMPLE:

```
TYPE BYTE = 0..#FF;
     RECTYPE = PACKED RECORD
                 MSBYTE, LSBYTE : BYTE
               END;

VAR V : ARRAY [1..10] OF INTEGER;
    R : RECTYPE;

BEGIN
  V[ 1 ] := #F6C1;

(* VALID TYPE TRANSFERS ARE: *)
  R.MSBYTE := V[ 1 ] :: BYTE;    (* R.MSBYTE = C1 *)
  R.LSBYTE := #FF;               (* R = C1FF      *)
  V[ 2 ] :: BYTE := R.LSBYTE;    (* V[ 2 ] = 00FF *)
       .
       .
```

**NOTES : THE VARIABLE MUST NOT BE DECLARED TO BE A PROCEDURE, FUNCTION OR CONSTANT.

A VARIABLE WHICH IS A COMPONENT OF A PACKED STRUCTURE MAY ONLY BE TRANSFERRED TO A TYPE REPRESENTABLE WITHIN THE BOUNDARIES OF THAT COMPONENT.

I.E. R.MSBYTE :: INTEGER IS ILLEGAL

THE TYPE TRANSFER APPLIES ONLY IN THE VARIABLE IN WHICH IT IS STATED.

WORKSHEET 13

1. Declare a variable which is a set consisting of the people in this class.

2. Write an assignment statement to include only yourself in the above set.

3. Declare a set consisting of integers 0 through 100.

4. Write a type transfer statement which will access a variable of type CHAR as an integer and increment it by 10.

SEQUENTIAL AND RELATIVE RECORD FILES

OBJECTIVES

STUDENTS SHOULD BE ABLE TO DECLARE A FILE OF THE APPROPRIATE TYPE FOR A GIVEN APPLICATION

STUDENTS SHOULD BE ABLE TO WRITE A PASCAL PROGRAM WHICH PERFORMS I/O OPERATIONS TO AN UNFORMATTED FILE

STUDENTS SHOULD BE ABLE TO ASSIGN SYNONYMS TO ASSOCIATE A PASCAL FILE WITH A DX10 FILE

STUDENTS SHOULD BE ABLE TO CREATE THEIR OWN FILES WITH THE APPROPRIATE LOGICAL AND PHYSICAL RECORD LENGTHS FOR A GIVEN APPLICATION

AGENDA

1. FILES
   - TEXT FILES
   - UNFORMATTED FILES
2. SEQUENTIAL FILES
3. RELATIVE RECORD (RANDOM ACCESS) FILES
4. DX10 FILES
   - CHARACTERISTICS
   - ASSIGNING SYNONYMS
   - CREATING DX10 FILES

FILES

TEXT FILES

- RECALL THAT THE FOLLOWING TYPES MAY BE WRITTEN TO A TEXT FILE

CHAR	DECIMAL
INTEGER	FIXED
LONGINT	BOOLEAN
REAL(N)	STRINGS (PACKED ARRAYS OF CHARACTERS)

TEXT FILE CHARACTERISTICS

- ALL I/O IS DONE IN CHARACTER FORMAT
- IMPLICIT CONVERSION TO CHARACTER FORMAT FOR NON-CHARACTER DATA TYPES
- RANDOM ACCESS TEXT FILES ARE NOT ALLOWED
- STANDARD I/O FILES 'INPUT' AND 'OUTPUT' ARE TEXT FILES
- DATA IS ORGANIZED IN 'LINES'
- 80 BYTE (CHARACTER) RECORDS IF AUTO CREATED

NON-TEXT FILES (REFERENCE - PASCAL USER MANUAL, SEC. 6.5)

PURPOSE: TO SERVE AS SEQUENTIAL OR RANDOM ACCESS MASS STORAGE MEDIUM FOR MOST DATA TYPES

DECLARATION:

```
VAR < FILE NAME > : [ RANDOM ] FILE OF < TYPE >;
```

WHERE,

< TYPE > CANNOT BE

- FILE
- POINTER
- ARRAY OF FILES OR POINTERS
- RECORD WITH A FILE OR POINTER COMPONENT

EXAMPLES:

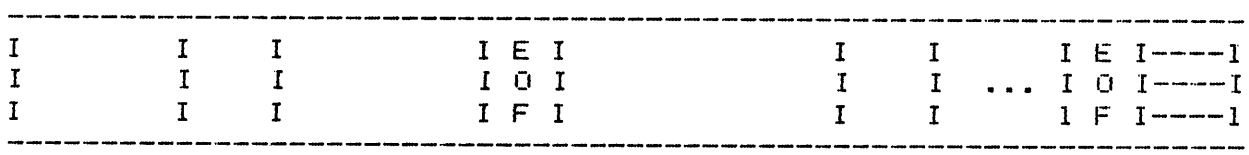
```
TYPE VECTOR = ARRAY [ 1..10 ] OF INTEGER;
      RTYPE = RECORD
                  DAT1 : INTEGER;
                  DAT2 : REAL;
                  DAT3 : ARRAY [ 1..5 ] OF CHAR
              END;

VAR  F1 : FILE OF VECTOR;
     F2 : FILE OF INTEGER;
     F3 : RANDOM FILE OF RTYPE;
```

CHARACTERISTICS OF NON-TEXT FILES

- DATA WRITTEN OR READ MUST BE OF THE SAME TYPE AS THE FILE
- DATA IS NOT CONVERTED TO/FROM CHARACTER FORMAT FOR I/O
- NON-TEXT FILES MAY BE ACCESSED BY RECORD NUMBER IF THEY ARE DECLARED TO BE 'RANDOM'
- DATA IS ORGANIZED IN RECORDS, NOT 'LINES' (I.E. EOLN DOES NOT APPLY TO NON-TEXT FILES)

SEQUENTIAL FILES

- SUPPORT MULTIPLE END-OF-FILES (EOF'S)
- MUST BE ACCESSED SEQUENTIALLY
- WHEN OPEN FOR INPUT, ACCESS IS READ-ONLY
- WHEN OPEN FOR OUTPUT, ACCESS IS EXCLUSIVE WRITE
- MAY NOT BE OPEN FOR READ/WRITE AT ONCE
- EACH RECORD IS EITHER A COMPONENT OF THE SAME TYPE AS THE FILE OR AN 'EOF' MARK



I/O PROCEDURES AND FUNCTIONS

REWRITE(F) - OPENS 'F' FOR OUTPUT
- FILE IS EMPTY AFTER OPEN (ERASED)
- EOF(F) IS SET TO 'TRUE'

EXTEND(F) - OPEN 'F' FOR OUTPUT
- POSITION TO WRITE FOLLOWING LAST RECORD IN FILE

RESET(F) - OPEN 'F' FOR INPUT
- 'F' IS POSITIONED TO THE FIRST COMPONENT OF THE FIRST LOGICAL FILE (REWIND)
- IF 'F' IS EMPTY THEN EOF(F) GETS SET TO 'TRUE' ELSE EOF(F) GETS SET TO 'FALSE'

CLOSE(F) - 'F' IS CLOSED

- AN EOF IS WRITTEN BEFORE CLOSING WHEN FILE IS OPEN FOR OUTPUT

READ(F, V) - NEXT COMPONENT OF 'F' IS ASSIGNED TO THE VARIABLE 'V'

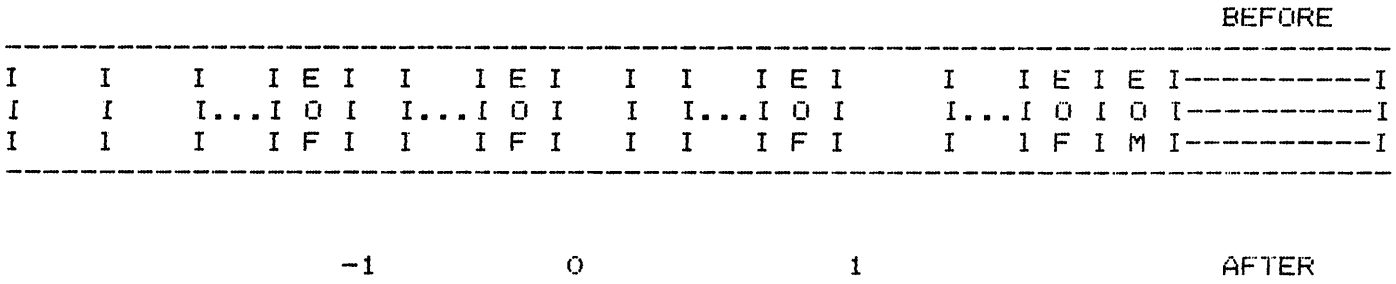
- EOF(F) BECOMES TRUE WHEN THE LAST COMPONENT OF A LOGICAL FILE IS READ

- USE 'SKIPFILES' TO PASS THE EOF

WRITE(F, E) - WRITE VALUE OF THE EXPRESSION 'E' AS THE NEXT COMPONENT OF 'F'

WRITEEOF(F) - WRITE AN EOF MARK ON 'F'

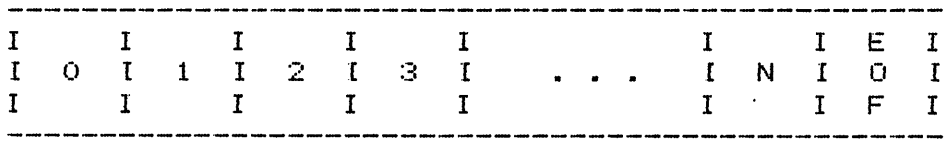
SKIPFILES (F, NFILE) - SKIP 'NFILE' (INTEGER) EOF MARKS AND POSITION FILE TO FIRST RECORD OF LOGICAL FILE



```
PROGRAM TESTFILE;

TYPE REC = RECORD
             PA : INTEGER;
             PB : REAL;
             PC : CHAR
           END;

VAR VAL1, VAL2 : REC;
    SEQFILE    : FILE OF REC;

BEGIN (* TESTFILE *)
   WITH VAL1 DO
   BEGIN
      PA := 47;
      PB := 3.4;
      PC := 'A';
   END;
   REWRITE( SEQFILE );     (* OPEN SEQFILE FOR OUTPUT *)
   WRITE ( SEQFILE, VAL1 );
   CLOSE ( SEQFILE );
   RESET ( SEQFILE );       (* OPEN SEQFILE FOR INPUT *)
   READ ( SEQFILE, VAL2 );
   CLOSE (SEQFILE );

(* OUTPUT THE VALUE OF THE INPUT VARIABLE 'VAL2' TO VERIFY THAT
   VALUES WERE WRITTEN AND READ CORRECTLY TO AND FROM THE
   SEQUENTIAL FILE                                          *)

   WRITELN;
   WRITELN ( 'VAL2.PA = ', VAL2.PA);
   WRITELN ( 'VAL2.PB = ', VAL2.PB);
   WRITELN ( 'VAL2.PC = ', VAL2.PC);
END.

MAXIMUM NUMBER IF IDENTIFIERS USED = 8
INSTRUCTIONS =   101 (LESS   0 WORDS OF DEAD CODE REMOVED)
  TESTFILE LITERALS =     108  CODE =    448  DATA =    176
```

RANDOM FILES

- SUPPORTS SINGLE EOF (AFTER 'LAST' RECORD)
- ACCESSED BY RECORD NUMBER
- MAY BE OPEN FOR BOTH READ AND WRITE AT ONCE



I/O PROCEDURES AND FUNCTIONS

REWRITE (F)	–	OPEN 'F' FOR INPUT/OUTPUT WITH EXCLUSIVE WRITE ACCESS
	–	FILE IS EMPTY (ERASED)
EXTEND (F)	–	OPEN 'F' FOR INPUT/OUTPUT WITH SHARED ACCESS
RESET (F)	–	OPEN 'F' FOR INPUT WITH READ-ONLY ACCESS
READ (F,REC,V)	–	ASSIGN THE 'REC' (INTEGER 0-N) COMPONENT OF 'F' TO THE VARIABLE 'V'
	–	IF 'REC' DOES NOT EXIST THEN EOF (F) = TRUE
WRITE (F,REC,E)	–	WRITE THE VALUE OF 'E' AS THE 'REC' COMPONENT OF 'F'
	–	'REC' >= 0.

```
PROGRAM RAND;

TYPE RECS = RECORD
              PA : INTEGER;
              PB : REAL;
              PC : CHAR;
            END;

VAR VAL1, VAL2 : RECS;
    RELREC     : RANDOM FILE OF RECS;
    RECNUM     : INTEGER;

BEGIN (* RAND *)
   RECNUM := 8;
   WITH VAL1 DO
   BEGIN
      PA := 22;
      PB := 6.9;
      PC := 'R';
   END;
   REWRITE ( RELREC );     (* OPEN AND ERASE FILE *)
   WRITE ( RELREC, RECNUM, VAL1 );
   READ ( RELREC, RECNUM, VAL2 );
   CLOSE ( RELREC );
(* WRITE THE VALUES OF HE RECORD READ BACK FROM THE FILE *)

   WRITELN;
   WRITELN( ' VAL2.PA = ', VAL2.PA,' VAL2.PB = ',VAL2.PB,
            ' VAL2.PC = ', VAL2.PC );
END. (* RAND *)

MAXIMUM NUMBER OF IDENTIFIERS USED = 9
INSTRUCTIONS =    94 (LESS   0 WORDS OF DEAD CODE REMOVED)
  RAND      LITERALS =    114  CODE =    432  DATA =    180
```

ASSOCIATING DX10 FILES WITH TIP FILE NAMES

- SYNONYMS ARE USED TO BIND A TIP FILE NAME TO AN ACTUAL FILE.

```
]AS
  --
ASSIGN SYNONYM

       SYNONYM: SEQFIL
                ------
         VALUE: TI.PASCAL.DATA.SEQFIL
                ---------------------
```

**NOTE: FILE SYNONYMS MUST BE ASSIGNED BEFORE PROGRAM IS EXECUTED.

**NOTE: THE FILE SPECIFIED AS 'VALUE' IN THE SYNONYM ASSIGNMENT NEED NOT EXIST, TIP WILL CREATE IT WHEN IT IS OPENED.

**NOTE: IF NO SYNONYM FOR A TIP FILE EXISTS, THE FILE IS CREATED ON THE SYSTEM DISK AS <NAME> NN, WHERE NN IS THE STATION ID.

EXAMPLE:

```
VAR TEST : FILE OF INTEGER;
```

FILE NAME BECOMES 'TEST03' IF YOU ARE EXECUTING FROM ST03

WORKSHEET 14

1. DECLARE A SEQUENTIAL FILE WHICH HAS COMPONENTS OF THE FOLLOWING TYPE:

```
TYPE BIGREC = RECORD
                TOP : INTEGER;
                BOTTOM : INTEGER;
                MATRIX : ARRAY [ 1..50 ] OF CHAR
              END;
```

2. DECLARE A RANDOM ACCESS FILE WHICH HAS COMPONENTS OF THE FOLLOWING TYPE

```
TYPE BVECTOR = PACKED ARRAY [ 1..100 ] OF BOOLEAN;
```

FOR EACH OF THE FOLLOWING, WRITE THE PASCAL STATEMENTS REQUIRED TO CARRY OUT THE OPERATION (NO DECLARATIONS NECESSARY)

3. OPEN AND ERASE THE SEQUENTIAL FILE 'SEQ'

4. SKIP OVER NEXT EOF IN THE SEQUENTIAL FILE 'SEQ'. IF END-OF-MEDIUM HAS BEEN REACHED, SET THE VARIABLE 'I' TO ZERO.

5. OPEN THE RANDOM FILE 'RELREC' FOR INPUT/OUTPUT. DO NOT ERASE THE FILE WHEN YOU OPEN IT.

6. WRITE THE VALUE OF THE VARIABLE 'VAL' TO RECORD 38 OF THE RANDOM FILE 'RELREC'

FILE ALLOCATION AND BLOCKING

ALLOCATION UNITS (ADU'S)

DISK SPACE IS ALLOCATED IN CHUNK'S CALLED ALLOCATION UNITS. THE SIZE OF AN ADU AND THE TOTAL NUMBER OF ADU'S ON A DISK IS DEPENDENT ON THE TYPE OF DISK BEING USED.

DISK STATISTICS

	DS31	DS10	T25	T50	T200
HEADS/DISK	2	4	5	5	19
TRACKS/DISK	406	1632	2040	4075	15485
SECTORS/TRACK	24	20	38	38	38
BYTES/SECTOR	288	288	288	288	288
SECTORS/ADU	1	1	2	3	9

PHYSICAL RECORDS

DEFINITION: A FILE'S PHYSICAL RECORD LENGTH IS THE NUMBER OF BYTES READ OR WRITTEN DURING AN ACCESS TO THE FILE.

CHARACTERISTICS:

- ALWAYS BEGIN ON A SECTOR BOUNDARY
- SHOULD BE AN INTEGRAL MULTIPLE OF THE SECTOR LENGTH

LOGICAL RECORDS

DEFINITION: A FILE'S LOGICAL RECORD LENGTH IS THE NUMBER OF BYTES REQUIRED BY A PROGRAM FOR A PARTICULAR I/O OPERATION.

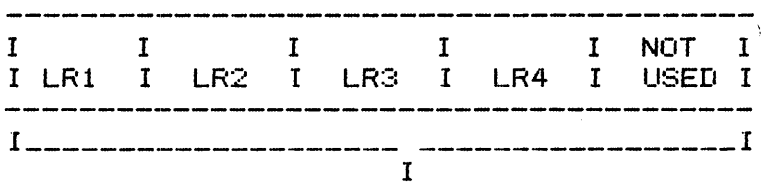
CHARACTERISTICS:

- USUALLY, MANY LOGICAL RECORDS MAY BE STORED IN A SINGLE PHYSICAL RECORD.
- FOR PASCAL FILES THIS VALUE IS THE NUMBER OF BYTES REQUIRED TO HOLD A RECORD OF THE TYPE THE FILE IS DECLARED TO BE.

BLOCKING

L = LOGICAL RECORD LENGTH
P = PHYSICAL RECORD LENGTH

LOGICAL RECORD/PHYSICAL RECORD = P DIV L



L = 200 SFBYTES

P = 864 BYTES

**NOTE: A LOGICAL RECORD WILL NOT USUALLY SPAN A PHYSICAL RECORD BOUNDARY

**NOTE: PHYSICAL RECORDS BEGIN ON A SECTOR BOUNDARY

EXAMPLE:

```
PROGRAM FILES;

      TYPE RELREC = RECORD
                      P1 : INTEGER;
                      P2 : REAL;
                      P3 : PACKED ARRAY [1..40] OF CHAR
                    END;

      TYPE SEQREC = RECORD
                      P1 : ARRAY [1..10] OF LONGINT;
                      P2 : REAL;
                      P3 : ARRAY [1..10] OF CHAR
                    END;

      VAR RADFILE : RANDOM FILE OF RELREC;
          SEQFILE : FILE OF SEQREC;
```

PASCAL WOULD CREATE THESE FILES AS FOLLOWS:

```
      RADFILE - PHYSICAL RECORD LENGTH 864 BYTES
                LOGICAL RECORD LENGTH 46 BYTES
                BLOCKING FACTOR = 18 TO 1

      SEQFILE - PHYSICAL RECORD LENGTH 864 BYTES
                LOGICAL RECORD LENGTH 64 BYTES
                BLOCKING FACTOR = 13 TO 1
```

CREATING YOUR OWN FILES

CREATE SEQUENTIAL FILE (CFSEQ) COMMAND

PURPOSE: TO CREATE A SEQUENTIAL FILE EXPLICITLY WITH THE CHARACTERISTICS REQUIRED BY YOUR PROGRAM.

FORMAT:

```
[] CFSEQ

CREATE SEQUENTIAL FILE

                PATHNAME:  < PATHNAME OF NEW FILE >

 LOGICAL RECORD LENGTH:  < LENGTH OF LOGICAL RECORDS IN BYTES >

PHYSICAL RECORD LENGTH:  < LENGTH OF PHYSICAL RECORDS IN BYTES >

    INITIAL ALLOCATION:  < NUMBER OF LOGICAL RECORDS TO BE ALLOCATED >

  SECONDARY ALLOCATION:  < LOGICAL RECORDS FOR SECOND ALLOCATION IF  >
                           FILE IS EXPANDABLE
           EXPANDABLE?:  YES/NO < IS FILE EXPANDABLE? >

       BLANK SUPPRESS?:  YES/NO < IS FILE BLANK SUPPRESSED? >

         FORCED WRITE?:  YES/NO < SHOULD THE FORCED WRITE OPTION BE
                                  USED? >
```

EXAMPLE:

```
[] CFSEQ
CREATE SEQUENTIAL FILE
                  PATHNAME:  TI.PASCAL.DATA.SEQFIL
     LOGICAL RECORD LENGTH:  80
    PHYSICAL RECORD LENGTH:  864
        INITIAL ALLOCATION:  15
      SECONDARY ALLOCATION:  2
                EXPANDABLE?:  YES
          BLANK SUPPRESSED?:  NO
              FORCED WRITE?:  NO
```

CREATE RELATIVE RECORD FILE (CFREL) COMMAND

PURPOSE: TO CREATE A RELATIVE RECORD (RANDOM) FILE EXPLICITLY WITH THE CHARACTERISTICS REQUIRED BY YOUR PROGRAM.

FORMAT:

[] CFREL

CREATE RELATIVE RECORD FILE

```
                PATHNAME: < PATHNAME OF NEW FILE>

 LOGICAL RECORD LENGTH: < LENGTH OF LOGICAL RECORDS IN TYPES >

PHYSICAL RECORD LENGTH: < LENGTH OF PHYSICAL RECORDS IN BYTES >

     INITIAL ALLOCATION: < NUMBER OF LOGICAL RECORDS TO BE ALLOCATED
                           INITIALLY                                  >

   SECONDARY ALLOCATION: < LOGICAL RECORDS FOR SECOND ALLOCATION IF
                           EXPANDABLE                                 >

            EXPANDABLE?:  YES/NO < IS FILE EXPANDABLE >

          FORCED WRITE?:  YES/NO < SHOULD THE FORCED WRITE OPTION BE
                                   USED                               >
```

EXAMPLE:

```
   [] CFREL
   CREATE RELATIVE RECORD FILE
                    PATHNAME: TI.PASCAL.DATA.SEQFIL
       LOGICAL RECORD LENGTH: 80
      PHYSICAL RECORD LENGTH: 864
          INITIAL ALLOCATION: 15
        SECONDARY ALLOCATION: 5
                  EXPANDABLE?: YES
                FORCED WRITE?: NO
```

ASSIGNMENT 3

It is desired to monitor the temperature in a room. A fan circulates air in the room. In setting up the system, it is observed that the temperature readout fluctuates due to the air turbulence in the room. To offset the effect of fluctuations, it is decided to smooth the temperature data by computing a moving average of the data over the most recent 10 samples of data. The smoothed data are then recorded on a file.

3A. The temperature samples are recorded on file PASCAL.DATA.DATA3. Your program should read a temperature from the file and echo print. Then insert it onto the front of a linked list. After all the data has been read, print out the linked list starting at the front to be sure the data has been read in correctly. (when you print out the list, the data should appear in reverse order.)

You should do the following:

Create a record containing a field for the temperature and a field for the pointer to the next record.

Create a pointer (perhaps called START) to point to the front of your list.

You should create the following procedures:

READ_ECHO : reads a temperature into a record.
sets the pointer field of that record to NIL.
echo prints the number
Parameters : a pointer to the record.

INSERT : Inserts the record into the front of the list
Parameters : pointer to the record

PRINT_LIST: Prints out the linked list

Your main program should do the following:

```
Reset the data file
initialize the pointer to the front of the list
check for EOF
  check for EOLN
    call READ_ECHO
    call INSERT
call PRINT_LIST
```

3B. Program 3A read in the data and created a linked list.
Program 3B will keep a record of the running averages.
To do this we will keep only the last 10 temperatures read
in the linked list. This means that we will read in 10 values,
and calculate the first average. After that each time we read in
a new value and insert it into the front of the list, we will
have to delete a value from the end of the list. Each time
a new value is read in, the average of the last 10 values should
be computed and printed out.

You should create the following procedure:

DELETE_LAST : Disposes of the last record in the list.
sets the pointer field of the new end record
to NIL.

Create the following function:

AVG : averages the 10 records on the list.

3C. File TI.DATA.DATA4 contains the following information on each employee.

```
   Employee number: 1 digit ( This is a small company )
   Name           : up to 20 characters
   social sec no. : 9 digits
   street         : 25 characters
   city, state    : 20 characters

Example: 1Jane Lobdill          455809190
         6708 Beckett Road
         Austin, Texas  78749
         2Les Wyatt
              .
              .
```

Using the employee number as the record number, read the file and store the information in a relative record file. Write the records back out in the order of employee number. Use any format you wish to write out the records.

STRATEGY:

- Declare a relative record file (RANDOM FILE OF...).
- Let n be a constant which is the number of employees.
  In this case n = 9.
- Create 2 procedures:
  - READ_ECHO : Reads the data into a 1PASCAL record and echo prints.
  - PRINT_FILE : reads data from the relative record file into a PASCAL record.
    writes the record out to the output file.

- Your main program should do the following:

```
Open the relative record file (rewrite or extend)
Open the input file
For each employee data do the following
  Initialize the street field and city,state field of the
     Pascal record to blanks
  Call READ_ECHO to read data into Pascal record
  Write the data out to the relative file
Call PRINT_FILE
Close the relative record file
```

3D Run your progream a second time and update your file using TI.DATA.DATA5

- The file you created during the first run is stored as .<filename>0x where x = terminal number.
  filename = the name you called the file in your program.

- To access the same file again, assign .<filename>0x a synonym which is the name you use in your program.

- This time open the file using the 'EXTEND' command

- There is no 'change-code' to tell you which field to change, so you will have to write the entire new record out.

- Print out the updated file in the order of employee no.

LANGUAGE LINKAGE CONVENTIONS

OBJECTIVES

TO AQUAINT INTERESTED STUDENTS WITH THE TECHNIQUES REQUIRED TO LINK IN AN EXTERNAL ROUTINE WRITTEN IN PASCAL, FORTRAN, COBOL, OR ASSEMBLY LANGUAGE.

TO ACQUAINT STUDENTS WITH THE TECHNIQUES REQUIRED TO LINK PASCAL PROGRAMS WHICH SHARE COMMON PROCEDURES AND RUNTIME

AGENDA

1. EXTERNAL PROCEDURE DECLARATIONS
2. EXTERNAL PASCAL ROUTINES
3. EXTERNAL FORTRAN ROUTINES
4. EXTERNAL ASSEMBLY LANGUAGE ROUTINES
5. REENTRANT PASCAL

LINKAGE CONVENTION TO OTHER LANGUAGES

EXTERNAL ROUTINES

ROUTINES OTHER THAN THOSE DEFINED IN YOUR PROGRAM MAY BE CALLED.

ROUTINES WHICH MAY BE CALLED INCLUDE:

- PASCAL ROUTINES
- FORTRAN ROUTINES
- REENTRANT FORTRAN ROUTINES
- COBOL ROUTINES
- ASSEMBLY LANGUAGE ROUTINES

CALLING AN EXTERNAL PASCAL ROUTINE

PASCAL ROUTINES WHICH ARE NOT DEFINED IN YOUR PROGRAM MUST BE DECLARED IN AN 'EXTERNAL' PROCEDURE DECLARATION.

SYNTAX:

PROCEDURE <NAME> (< PARM LIST >); EXTERNAL;

EXAMPLE:

```
PROCEDURE TEST ( VAR I : REAL ); EXTERNAL;

          .
          .
          .

      TEST ( RVAL );
```

**NOTES:

- THE EXTERNAL PROCEDURE MUST BE PART OF COMPLETE PASCAL PROGRAM
- THE EXTERNAL PROCEDURE MUST BE DEFINED AT THE SAME LEVEL AS THE EXTERNAL PROCEDURE DECLARATION IN THE CALLING PROGRAM.
- THE MAIN PROGRAM WHICH CONTAINS THE EXTERNAL PROCEDURE MUST CONTAIN THE COMPILER OPTION 'NO OBJECT'.

EXAMPLE:

MAIN

```
PROGRAM EXTTST;

VAR I : INTEGER;

PROCEDURE EXT ( VAR J : INTEGER ); EXTERNAL;

BEGIN (* EXTTST *)
     I : =5;          (*INITIALIZE I*)
     EXT ( I );       (*CALL EXTERNAL PROC*)
     WRITELN ( ' AFTER CALL I = ', I:3 );
END. (* EXTTST *)
```

EXT. PROC

```
PROGRAM DUMY;

     PROCEDURE EXT ( VAR J : INTEGER );

        VAR TEMP : INTEGER;

        BEGIN (* EXT *)
           TEMP := J + 1;
           J := TEMP;
           WRITELN ( ' HI FROM EXTERNAL ROUTINE ' )
     END; (* EXT *)

BEGIN (*DUMY*)
  (*$ NO OBJECT *)
END. (*DUMY*)
```

LINK CONTROL FILE

```
NOSYMT
FORMAT IMAGE,REPLACE,3
LIBRARY .TIP.OBJ
TASK EXT
  INCLUDE (MAIN)
  INCLUDE TI.PASCAL.OBJ.EXTST
  INCLUDE TI.PASCAL.OBJ.EXT
END
```

CALLING A FORTRAN SUBROUTINE

TO CALL A FORTRAN SUBROUTINE, A FORTRAN PROCEDURE DEFINITION MUST BE MADE.

```
PROCEDURE FTN ( I : INTEGER;
                R : REAL    ); EXTERNAL FORTRAN;

          .
          .
          .

      FTN (VAL1,VAL2);
```

**NOTE:
THE FORMAL PARAMETERS IN THE FORTRAN PROCEDURE DEFINITION SHOULD MATCH THOSE IN THE FORTRAN SUBROUTINE IN TYPE AND SIZE

"VAR'ED" PARAMETERS ARE CALLED BY REFERENCE AND "UN-VAR'ED" PARAMETERS ARE CALLED BY VALUE.

PASCAL PROGRAM TO CALL FORTRAN ROUTINE

```
PROGRAM FTNTST;

    VAR     VAL1 : INTEGER;
            VAL2 : REAL;

(*************************************************************************
*   THIS IS AN EXTERNAL FORTRAN PROCEDURE.  IT INCREMENTS THE VALUE     *
*   OF EACH OF IT'S TWO PARAMETERS AND RETURNS THOSE VALUES             *
*************************************************************************)

    PROCEDURE FTN ( VAR I : INTEGER;
                    VAR R : REAL ); EXTERNAL FORTRAN;

    BEGIN (* FTNTST *)
       VAL1 := 47;
       VAL2 := 3.4;
       FTN ( VAL1, VAL2 );
       WRITELN ( ' VAL1 = ', VAL1:3, '   VAL2 = ', VAL2:7 );
    END. (* FTNTST *)
```

SAMPLE EXTERNAL FORTRAN ROUTINE

```
C  THIS FORTRAN SUNBROUTINE IS CALLED BY A PASCAL PROGRAM.
C  IT HAS TWO ARGUMENTS
C
C     I - INTEGER
C     R - REAL
C
C  EACH OF THESE PARAMETERS IS CALLED BY REFERENCE AND
C  EACH IS INCREMENTED BY ONE AND THE VALUE RETURNED
C
      SUBROUTINE FTN ( I, R )
      I = I + 1
      R = R + 1.0
      WRITE ( 6, 50 )
50    FORMAT ( ' HI FROM FORTRAN ' )
      RETURN
      END
```

**NOTE: A FORTRAN subroutine does not have to be part of a complete FORTRAN program.

SAMPLE LINK CONTROL STREAM TO LINK EXTERNAL FORTRAN ROUTINE

```
NOSYMT
FORMAT IMAGE,REPLACE,3
LIBRARY .TIP.OBJ
LIBRARY .FORTRN,OSLOBJ
LIBRARY .FORTRN,STLOBJ
TASK FORT
     INCLUDE (MAIN)
     INCLUDE (FTNIO)
     INCLUDE TI.PASCAL.OBJ.FTNTST
     INCLUDE TI.PASCAL.OBJ.FTN
END
```

CALLING AN ASSEMBLY LANGUAGE ROUTINE

IN ORDER TO CALL AN ASSEMBLY LANGUAGE ROUTINE, YOU MUST USE EITHER THE FORTRAN OR PASCAL LINKAGE CONVENTIONS.

FORTRAN LINKAGE CONVENTIONS

WHEN A FORTRAN SUBROUTINE CALL IS MADE THE FOLLOWING CODE IS GENERATED.

```
CALL SUB1 ( A1, A2, . . . , AN )
```

GENERATES,

ADDRESS	CODE	COMMENTS
	REF SUB1	
XXXX + 00	BLWP @SUB1	BLWP TO SUBROUTINE
XXXX + 02		
XXXX + 04	DATA N	NUMBER OF ARG'S TO SUB1
XXXX + 06	DATA A1	ADDRESS OF ARG 1
XXXX + 08	DATA A2	ADDRESS OF ARG 2
	.	
	.	
	.	
XXXX + ??	DATA AN	ADDRESS OF ARG N

IF A PASCAL PROCEDURE IS DEFINED AS 'EXTERNAL FORTRAN', THE SAME CALLING CODE IS GENERATED AS FOR A CALL IN A FORTRAN PROGRAM.

EXAMPLE:

```
VAR  IVAL1 : INTEGER;
     IVAL2 : REAL;

PROCEDURE TEST ( I : INTEGER;
                 J : REAL    ); EXTERNAL FORTRAN;


     TEST (IVAL1,IVAL2);

               .
               .
               .
```

**NOTE:
THE CODE ABOVE WOULD GENERATE THE SAME CODE AS FOLLOWING FORTRAN

```
SUBROUTINE TEST ( I, J )

          .
          .
          .

CALL TEST ( IVAL1, IVAL2 )

          .
          .
          .
```

STRUCTURE OF THE ASSEMBLY ROUTINE

```
          IDT   'ASMSUB'       IDT NAME OF THE ASSM. SUB.
          DEF   ASMSUB          EXTERNAL DEFINITION OF TRAP VECTOR
ASMSUB    DATA WSP              - NEW WORKSPACE POINTER -
          DATA ENTER            - NEW PROGRAM COUNTER   -
WSP       BSS  32               RESERVE SPACE FOR REGISTERS
*****************************************************************
*                                                               *
*                     DATA AREA FOR SUB                         *
*                                                               *
*****************************************************************
ENTER     EQU  $                ENTRY POINT INTO ROUTINE
*****************************************************************
*                                                               *
*                  PROCEDURE AREA FOR SUB                       *
*                                                               *
*****************************************************************
*          RETURN LOGIC WHICH COULD BE USED FOLLOWS             *
          MOV  14,7             GET PARM COUNT ADDRESS
          MOV  *7,7             GET PARM COUNT
          INCT 14               INCREMENT PAST COUNT WORD
          SLA  7,1              R7 = R7*2 - TWO BYTES/WORD
          A    7,14             INCREMENT PAST PARAMETERS
          RTWP                  RETURN TO PASCAL PROGRAM
          END                   END ASSEMBLER DIRECTIVE
```

SAMPLE PASCAL PROGRAM TO CALL EXTERNAL ASSEMBLY ROUTINE

```
PROGRAM ASMTEST;

  VAR      I : INTEGER;
           R : REAL;
           C : ARRAY [ 1..5 ] OF CHAR;

(*************************************************************************
*   THIS IS AN EXTERNAL ASSEMBLY LANGUAGE ROUTINE WHICH USES THE        *
*   FORTRAN LINKAGE CONVENTIONS.                                        *
*************************************************************************)

  PROCEDURE ASMSUB ( VAR IVAL : INTEGER;
                     VAR RVAL : REAL;
                     VAR CVAL : CHAR ); EXTERNAL FORTRAN;

  BEGIN (* ASMTEST *)
     I := 5;
     R := 0.5;
     C [ 1 ] := 'A';
     C [ 2 ] := 'B';
     C [ 3 ] := 'C';
     C [ 4 ] := '1';
     C [ 5 ] := '2';
     WRITELN ( ' BEFORE CALL TO ASMSUB ' );
     ASMSUB ( I, R, C [ 3 ] );
     WRITELN ( ' AFTER CALL  I = ',I:2,'  R = ',R:15,'  C[3] =',C[3] );
  END. (* ASMTEST *)
```

SAMPLE LINK CONTROL STREAM FOR LINKING EXTERNAL ASSEMBLY ROUTINE

```
NOSYMT
FORMAT IMAGE,REPLACE,3
LIBRARY .TIP.OBJ
TASK TEST
   INCLUDE (MAIN)
   INCLUDE TI.PASCAL.OBJ.ASMTST
   INCLUDE TI.PASCAL.OBJ.ASMSUB
END
```

```
             SDSMAC      3.1 *    16:31:26 TUESDAY, APR 18, 1978.
ACCESS NAMES TABLE                                                  PAGE 0001
SOURCE ACCESS NAME=      .LOR.FTNASM
OBJECT ACCESS NAME=      DUMY
LISTING ACCESS NAME=     .LOR.EXT8
ERROR ACCESS NAME=
OPTIONS=                 TUNLST
MACRO LIBRARY PATHNAME=
ASMSUB         SDSMAC      3.1 *    16:31:26 TUESDAY, APR 18, 1978.
ASSEMBLY LANGUAGE SUB CALLED FROM PASCAL                            PAGE 0002
0002                    IDT  'ASMSUB'

 0003                   DEF  ASMSUB

 0004            *

 0005            *  TWO WORD TRAP VECTOR FOR BLWP FROM PASCAL PROGRAM

 0006            *

 0007 0000 0004' ASMSUB DATA WSP          -  NEW WORKSPACE POINTER -

 0008 0002 002A'        DATA ENTER        -  NEW PROGRAM COUNTER   -

 0009            *

 0010 0004       WSP    BSS  32           RESERVE NEW WORKSPACE AREA

 0011            *

 0012 0024 8000  RMASK  DATA >8000        REAL SIGN BIT MASK

 0013 0026 0000  INT    DATA 0            RESERVE SPACE FOR INTEGER PARAM

 0014 0028 0000  CHAR   DATA 0            RESERVE SPACE FOR CHAR PARAM

 0015            *

 0016            *  NOTE:  WHEN THIS ROUTINE IS ENTERED, THE OLD PROGRAM COUN

 0017            *         FROM THE PASCAL PROGRAM WHICH IS IN R14, CONTAINS

 0018            *         ADDRESS OF THE WORD CONTAINING THE NUMBER OF PARAM

 0019            *         IN THE EXTERNAL ROUTINE CALLING STATEMENT

 0020            *

 0021       002A' ENTER  EQU  $            BEGIN PROCEDURE AREA

 0022 002A C1CE       MOV  14,7          OBTAIN PARA BLOCK ADDRESS

 0023 002C 05C7       INCT 7             MOVE TO FIRST PARM ADDR

 0024 002E C237       MOV  *7+,8         R8 <-- PARM1 ADDR

 0025 0030 C818       MOV  *8,@INT       OBTAIN INTEGER PARM
      0032 0026'

 0026 0034 05A0       INC  @INT          INCREMENT INTEGER BY 1
      0036 0026'
```

```
0027 0038 C620          MOV   @INT,*8       RETURN INTEGER VALUE
     003A 0026'
0028 003C C237          MOV   *7+,8         R8 <-- PARM2 ADDR
0029 003E C258          MOV   *8,9          OBTAIN REAL PARM PART 1
0030 0040 E260          SOC   @RMASK,9      CHANGE SIGN BIT
     0042 0024'
0031 0044 C609          MOV   9,*8          RETURN REAL VALUE
0032 0046 C217          MOV   *7,8          R8 <-- PARM3 ADDR
0033 0048 C818          MOV   *8,@CHAR      OBTAIN CHAR PARM
     004A 0028'
0034 004C 05A0          INC   @CHAR         INCREMENT CHAR REPRESENTATION
     004E 0028'
0035 0050 C620          MOV   @CHAR,*8      RETURN CHARACTER VALUE
     0052 0028'
0036                *
0037                * SET UP FOR RETURN TO PASCAL PROGRAM
0038                *
0039                * SEE DESCRIPTION OF LINKAGE CONVENTIONS
0040                *
0041                * RECALL THAT R14 CONTAINS THE OLD PROGRAM COUNTER VALUE
0042                *
0043 0054 C1CE          MOV   14,7          GET PARM COUNT ADDR
0044 0056 C1D7          MOV   *7,7          GET PARM COUNT
0045 0058 05CE          INCT  14            INCREMENT PAST COUNT WORD
0046 005A 0A17          SLA   7,1           R7 = R7*2

0047 005C A387          A     7,14          INCREMENT PAST PARMS
0048 005E 0380          RTWP                RETURN TO PASCAL PROG.
0049                    END
NO ERRORS
```

REENTRANT PASCAL

PASCAL TASKS MAY SHARE
- Runtime Routines
- User defined routines

LINKING PASCAL TASKS WHICH SHARE RUNTIME ROUTINES

The usual method of linking Pascal programs uses a module called MAIN which is included as the first module in the link. The MAIN module is a partial link of a number of run-time routines which are always needed. For linking with separate procedure and task segments most of the routines in MAIN can be in the procedure segment, but some need to be in the task segment. Therefore, the partial link module MAIN cannot be used and the routines will have to be picked up individually.

LINK CONTROL FILE FORMAT

```
FORMAT IMAGE,REPLACE
LIBRARY .TIP.OBJ                ; Pascal run-time library
LIBRARY <user library>          ; User defined library
PROCEDURE <procedure name>      ; Procedure segment
INCLUDE (INIT$1)                ; Includes shared run-time routines
INCLUDE (CREATE$)
INCLUDE (GET$PA)
INCLUDE (HALT$)
INCLUDE (DUMP$P)
INCLUDE (RELEAS)
INCLUDE (CLS$)
INCLUDE (RESUM$)
INCLUDE (RSUMR$)
INCLUDE <other shared run-time routines>
          .
          .
INCLUDE <user defined routines>
          .
          .
SEARCH  .TIP.OBJ                ; Rest of sharable run-time
SEARCH  <user defined library>
TASK    <task name>             ; Beginning of TASK segment
INCLUDE (P$MAIN)                ; Must be first module in TASK segment
ALLOCATE                        ; Puts COMMONS and DSEGs here
INCLUDE <User program>
          .
          .
END
```

THE FOLLOWING ROUTINES ARE NOT SHARABLE AND MUST NOT BE PUT IN THE PROCEDURE SEGMENT:

```
DUMP$HEA
GO$
INIT$
INIT$D
P$INIT
TERM$
```

IN ORDER TO BE SAFE REMEMBER THE FOLLOWING RULES:

1. The PROCEDURE segment of the link must be identical in the link-edit of every task which will use it.
2. The library routine P$MAIN must be the first module in the TASK segment.
3. The Pascal main program must be in the task segment.
4. There must not be any references from routines in the PROCEDURE segment to labels defined after the ALLOCATE command.

Control files for 2 tasks which share a common run-time and 2 user defined procedures:

TASK1

```
FORMAT  IMAGE,REPLACE
LIBRARY .TIP.OBJ
LIBRARY USER.PASCAL.OBJECT
PROCEDURE TESTPROC
INCLUDE (INIT$1)
INCLUDE (CREAT$)
INCLUDE (GET$PA)
INCLUDE (HALT$)
INCLUDE (DUMP$P)
INCLUDE (RELEAS)
INCLUDE (CLS$)
INCLUDE (RESUM$)
INCLUDE (RSUMR$)
INCLUDE (ADD)
INCLUDE (FILL)
SEARCH  .TIP.OBJ
SEARCH  USER.PASCAL.OBJECT
TASK    TASK1
INCLUDE (P$MAIN)
ALLOCATE
INCLUDE (TASK1)
END
```

TASK2

```
FORMAT IMAGE,REPLACE
LIBRARY .TIP.OBJ
LIBRARY USER.PASCAL.OBJECT
PROCEDURE TESTPROC
DUMMY
INCLUDE (INIT$1)
INCLUDE (CREAT$)
INCLUDE (GET$PA)
INCLUDE (HALT$)
INCLUDE (DUMP$P)
INCLUDE (RELEAS)
INCLUDE (CLS$)
INCLUDE (RESUM$)
INCLUDE (RSUMR$)
INCLUDE (ADD)
INCLUDE (FILL)
SEARCH  .TIP.OBJ
SEARCH  USER.PASCAL.OBJECT
TASK    TASK1
INCLUDE (P$MAIN)
ALLOCATE
INCLUDE (TASK1)
END
```

** The second link control file has a DUMMY command after the PROCEDURE command so that the procedure is not stored on the program file

DIRECT INTERFACE WITH DX10

- Six routines for direct interface with DX10
- Allow I/O to a CRU device that is not supported by the system.
- Allow user to execute any supervisor call of the operating system.
- Routine must be declared externally

  - $SBO
  - $SBZ
  - $STCR
  - $TB

SUPERVISOR CALL ROUTINE

  - SVC$

```
PROCEDURE $LDCR(BASE,WIDTH,VALUE: INTEGER); EXTERNAL;

Example:  Var B, W, V : Integer;
                .
                .
          B := #200;     (* set base to #200 *)
          W := 8;        (* Write a character *)
          V := #41;      (* character is 'A' *)
          $LDCR( B, W, V);


PROCEDURE $SBO ( BASE : INTEGER ); EXTERNAL;

Example:  Var ADDR : Integer;
                .
                .
          ADDR := #400;      (* Set base to #400 *)
          ADDR := ADDR + 4;  (* Add displacement *)
          $SBO( ADDR );      (* Set bit to one   *)


PROCEDURE $SBZ ( BASE : INTEGER ); EXTERNAL;

Example:  Var BITOUT : Integer;
                .
                .
          BITOUT := #400;         (* Set Base to #400 *)
          BITOUT := BITOUT + 2;   (* Add Displacement *)
          $SBZ ( BITOUT );        (* SET BIT TO ZERO  *)


PROCEDURE $STCR(BASE, WIDTH, INTEGER; VAR VALUE: INTEGER); EXTERNAL;

Example:  Const BS1   = #200;
                SIZE = 8;
          Var   INCHR : Integer;
                   .
                   .
          $STCR( BS1, Size, INCHR );   (* Read Character *)


FUNCTION $TB ( BASE : INTEGER ): BOOLEAN; EXTERNAL;

Example:  Var ADD :  Integer;
              BUSY:  Boolean;
                   .
                   .
          ADD := #200;              (* Set Base to #200 *)
          ADD := ADD + 8;           (* Add Displacement *)
          BUSY := $TB (ADD);        (* Set Busy to value
                                       Of CRU Input     *)
```

```
PROCEDURE SVC$ ( P : PT ); EXTERNAL;
```

- The structure for the supervisor call block must be declared.

- PT represents a pointer to the supervisor call block

```
Example:  TYPE DANDT = ARRAY [ 1..5 ] OF INTEGER;
               POINT = @DANDT;
               SCB = RECORD
                       CODE : INTEGER;
                       TIME : POINT
                     END;
               PT = @SCB;

          VAR  BLOCK : PT;
                  .
                  .
          NEW ( BLOCK );        (* Obtain supervisor call block *)
          NEW ( BLOCK@.TIME );(* Obtain array for date and time*)
          BLOCK@.CODE := #0300;    (* Assign code and zero *)
          SVC$ ( BLOCK );  (* get date and time *)
```

- This shows a date and time supervisor call
- This supervisor call requires a 4-byte block
  - first byte = code
  - 2nd byte = zero
  - 3rd & 4th = address of a five word array for the date and time

- All values for the date and time in the array are binary

- Element 1 of array (Block@.Time@[I]) is the year
  - year is binary equivalent of the last 2 digits of year
- Element 2 is the day
- Element 3 is the hour
- Elememt 4 is the minute
- Element 5 is the second

BATCH SCI

O BATCH INPUT TO SCI FROM SEQUENTIAL MEDIA

- SEQUENTIAL FILE (TYPE CREATED BY THE TEXT EDITOR)
- CARD READER
- ETC.

O EXECUTION IS IN THE BACKGROUND

EXECUTE BATCH (XB) COMMAND

PURPOSE: TO PLACE A BATCH SCI COMMAND STREAM INTO EXECUTION IN THE BACKGROUND AT A STATION.

FORMAT:

[] XB

EXECUTE BATCH

INPUT ACCESS NAME: SEQUENTIAL MEDIA CONTAINING SCI COMMANDS

LISTING ACCESS NAME: FILE OR DEVICE FOR LISTING BATCH COMMANDS

EXAMPLE:

[] XB

EXECUTE BATCH

INPUT ACCESS NAME: TI.PASCAL.BATCH.TEST1

LISTING ACCESS NAME: LP01

BATCH COMMAND STREAM

BATCH SCI COMMAND FORMAT

FORMAT:

< COMMAND > K1 = < VALUE >, ... , KN = < VALUE >

WHERE,

< COMMAND > - MNEMONIC FOR SCI COMMAND

K1, ... KN - KEYWORDS

EXAMPLE:

```
CFDIR   PATHNAME = TI.COBOL.SRC,
        MAX ENTRIES = 100
```

**NOTE: SOME ABBREVIATIONS FOR KEYWORDS ARE LEGAL.

GENERAL RULE: USE THE FIRST LETTER OF THE KEYWORD IN THE ABBREVIATION.

THE LETTERS IN THE ABBREVIATION MUST APPEAR IN THE SAME ORDER AS THEY APPEAR IN THE KEYWORD

THE ABBREVIATION USED FOR A KEYWORD MUST NOT BE A LEGAL ABBREVIATION FOR ANOTHER KEYWORD IN THAT SAME SCI COMMAND.

** CONSULT VOLUME V OF THE DX10 MANUAL FOR THE EXACT DEFINITION OF THE NEAR EQUALITY ABBREVIATION ALGORITHM.

BATCH COMMAND

PURPOSE: TO DELETE ALL 'SECRET' SCI SYNONYMS AND DEFAULT VALUES FOR BATCH EXECUTION. (THIS HELPS GUARD AGAINST SYNONYM TABLE OVERFLOW)

FORMAT:

BATCH

**NOTE: THE 'BATCH' COMMAND SHOULD BE THE FIRST COMMAND IN EVERY BATCH COMMAND STREAM.

ILLEGAL BATCH SCI COMMANDS

- ALL DEBUG AND TEXT EDITOR COMMANDS
- AT (ACTIVATE TASK) COMMAND
- KBT (KILL BACKGROUND TASK) COMMAND
- MS (MODIFY SYNONYMS) COMMAND
- SBS (SHOW BACKGROUND STATUS) COMMAND
- XB (EXECUTE BATCH) COMMAND
- XD (INITIATE DEBUG) COMMAND
- XGEN (EXECUTE GEN 990) COMMAND
- XHT (EXECUTE AND HALT TASK) COMMAND
- MVI (MODIFY DISK VOLUME INFORMATION)-COMMAND

OTHER INTERACTIVE BATCH RELATED COMMANDS

SHOW BACKGROUND STATUS (SBS) COMMAND

PURPOSE: TO DISPLAY A MESSAGE WHICH DESCRIBES THE STATUS OF THE CURRENT BACKGROUND ACTIVITY AT THE STATION.

FORMAT:

```
[] SBS
```

EXAMPLE:

```
[] SBS

SHOW BACKGROUND STATUS

ACTIVE - PRIORITY 3
```

KILL BACKGROUND TASK (KBT) COMMAND

PURPOSE: TO TERMINATE BACKGROUND ACTIVITY AT A STATION.

FORMAT:

```
[] KBT
```

```
*                    SAMPLE BATCH SCI COMMAND STREAM
*
*
*
*       REMEMBER THAT 'BATCH' SHOULD BE THE FIRST COMMAND
*       IN EVERY BATCH STREAM!
*
        BATCH
*
*       EXECUTE THE PASCAL COMPILER
*
      XTIP      SOURCE = "TI.PASCAL.SRC.BATCH",
                OBJECT = "TI.PASCAL.OBJ.BATCH",
               LISTING = "TI.PASCAL.LST.BATCH",
              MESSAGES = "TI.PASCAL.MSG.BATCH",
                  MEM1 = "",
                  MEM2 = "",
                  MEM3 = "",
                  MODE = "FOREGROUND"
*
*
*       THERE IS A WAY IN A BATCH STREAM TO DETERMINE IF A PASCAL COMPILATON
*       WAS SUCCESSFUL BEFORE GOING ON TO THE NEXT COMMAND IN THE STREAM.
*
*       THE '.IF' COMMAND ALLOWS FOR CONDITIONAL EXECUTION OF A SERIES OF
*       COMMANDS.
*
*       THE SYNONYM '$$CC' WILL BE EQUAL 0 IF THE COMPILER TERMINATED
*       NORMALLY, AND A NON-ZERO VALUE INDICATES THAT THE COMPILER
*       TERMINATED ABNORMALLY.
*
*      THE SYNONYM '$$CC' WILL BE:
*
*            0 - NO ERRORS DURING COMPILATION
*
*        #4000 - ONLY WARNINGS OCCURRED DURING COMPILATION
*
*        #A000 - NON-FATAL ERRORS OCCURRED
*
*        #8000 - FATAL ERRORS OCCURRED
*
*        #C000 - RUNTIME ERRORS OCCURRED
*
*       THUS TO CONDITIONALLY LINK THE FOLLOWING CONTROL STREAM COULD BE UED.
*
*           .IF @$$CC, EQ, 0
*
*
*                    [ LINK COMMAND STREAM ]
*
*
*           .ENDIF
*
*
*
*
*
*
*
*
*
*
```

```
*
*
*
*
*       LINKING COMMAND STREAM
*       ----------------------
*
        .IF @$$CC, EQ, 0                        ! COMPILATION TERMINATED NORMLLY
              .DATA  TI.PASCAL.LC.BATCH         ! PUT LINK STREAM ON FILE
                  NOSYMT
                 FORMAT IMAGE,REPLACE,3
                 LIBRARY .TIP.OBJ
                 TASK NEW
                   INCLUDE (MAIN)
                   INCLUDE TI.PASCAL.OBJ.BATCH
                 END
              .EOD
*
*
*       EXECUTE LINK EDITOR
*
              XLE        CONT ACC NAME        = "TI.PASCAL.LC.BATCH",
                         LINKED OUT ACC NAME  = "TI.PASCAL.PROG",
                         LIST ACC NAME        = "TI.PASCAL.LMAP.BATCH",
                         PRINT WIDTH          = "80"
*
*
        .ENDIF
*
*
*       PRINT OUT THE COMPILED LISTING
*       ------------------------------
*
*
*
        PF      FILE PATHNAME(S)="TI.PASCAL.LST.BATCH",
                AF="NO",
                LISTING D="LP01",
                DAP="NO",
                NUM OF LINES/PAGE =""
*
*
*       PRINT OUT MESSAGE FILE
*
*
*
        PF      FILE PATHNAME(S)="TI.PASCAL.MSG.BATCH",
                AF="NO",
                LISTING D="LP01",
                DAP="NO",
                NUM OF LINES/PAGE=""
*
*
*       END OF BATCH STREAM - REMEMBER TO USE THE EBATCH COMMAND
*
        EBATCH
```

COMPILER OPTIONS

LIST CONTROL OPTIONS

LIST (STATEMENT - TRUE)

- ENABLES OR DISABLES PROGRAM SOURCE LISTING
- WHEN SET TO FALSE, ONLY LINES WITH ERRORS AND THE ERROR MESSAGES ARE PRINTED

WIDELIST (PROGRAM - FALSE)

- ENABLES OR DISABLES SOURCE LINE NUMBER AND COMPOUND STATEMENT NUMBERS

MAP (ROUTINE - FALSE)

- ENABLES OR DISABLES A 'MAP' OF THE VARIABLES DEFINED IN THE ROUTINE

**NOTE: THIS IS USEFUL WHEN ATTEMPTING TO READ AN ERROR DUMP OR STACK AND HEAP MEMORY

MAP INFORMATION

<V NAME> DISP = <HEX VALUE> DRTC = T/F SIZE = (BYTES, BITS)

WHERE,

<V NAME> - VARIABLE NAME

<HEX VALUE> - DISPLACEMENT (IN HEX BYTES) INTO THE ROUTINES STACK FRAME WHERE THAT VARIABLES STORAGE MAY BE FOUND

T - VALUE IS ACTUAL IN THE STACK AS DEFINED BY 'DISP'

F - VALUE IN STACK IS A POINTER TO THE ACTUAL LOCATION WHERE THE VARIABLE IS STORED

BYTES - SIZE OF VARIABLE IN FULL BYTES

BITS - NUMBER OF BITS OVER THE FULL NUMBER OF BYTES REQUIRED FOR VARIABLE STORAGE.

** NOTE: TOTAL NUMBER OF BITS OF MEMORY REQUIRED BY A VARIABLE CAN BE CALCULATED AS:

SIZE (INBITS) = 8*BYTES + BITS

PAGE (STATEMENT - FALSE)

- FORM FEED COMPILED LISTING DEVICE IF TRUE
- PAGE BECOMES FALSE AFTER FORM FEED IS WRITTEN

WARNINGS (STATEMENT - TRUE)

- ENABLES OR DISABLES LISTING OF WARNING MESSAGES FROM COMPILER

OBJECT CONTROL OPTIONS

NULLBODY (ROUTINE - FALSE)

- WHEN TRUE ONLY DUMMY OBJECT IS WRITTEN FOR ROUTINE
- USEFUL WHEN COMPILING EXTERNAL ROUTINES

TRACEBACK (ROUTINE - TRUE)

- ENABLES OR DISABLES GENERATION OR TRACEBACK DATA IN THE OBJECT CODE TO ALLOW TRACING OF EVENTS WHICH LED TO TERMINATION IN A PROGRAM.

ASSERTS (STATEMENT - TRUE)

- ENABLES OR DISABLES RECOGNITION OF ASSERT STATEMENTS IN A PROGRAM

ASSERT STATEMENT

PURPOSE: TO GENERATE A RUNTIME ERROR IF A SPECIFIED BOOLEAN CONDITION IS FALSE.

SYNTAX:

ASSERT <BOOLEAN EXPRESSION>

SYMANTICS:

- IF <BOOLEAN EXPRESSION> IS TRUE, THEN CONTINUE EXECUTION OF PROGRAM
- IF <BOOLEAN EXPRESSION> IS FALSE, THEN GENERATE A RUNTIME ERROR.

RUNTIME CHECK OPTIONS

CKINDEX (STATEMENT - FALSE)

- ENABLES OR DISABLES CHECKING FOR ARRAY

CKOVER (STATEMENT - FALSE)

- ENABLES OR DISABLES CHECKING OVERFLOW WHEN EVALUATING, INTEGER, LONGINT, DECIMAL, AND FIXED EXPRESSIONS.

CKPREC (STATEMENT - FALSE)

- ENABLES OR DISABLES CHECKING FOR LOSS OF MOST SIGNIFICANT PRECISION DURING CONVERSION OF DECIMAL AND FIXED TYPES.

CKPTR (STATEMENT - FALSE)

- ENABLES OR DISABLES CHECKING FOR NIL VALUES OF VARIABLES OF POINTER TYPE AT EXECUTION TIME.

CKSET (STATEMENT - FALSE)

- ENABLES OR DISABLES CHECKING OF SET ELEMENT EXPRESSIONS.

CKSUB (STATEMENT - FALSE)

- ENABLES OR DISABLES THE CHECKING OF SUBRANGE ASSIGNMENTS AND RESULTS OF PRED AND SUCC FUNCTIONS.

CKTAG (STATEMENT - FALSE)

- ENABLES OR DISABLES THE CHECKING OF THE TAG FIELDS OF RECORD VARIANTS.

PROBER (ROUTINE - FALSE)

- ENABLES OR DISABLES A SUMMARY OF THE NUMBER OF TIMES A ROUTINE IS EXECUTED DURING A PROGRAM EXECUTION

- THE FOLLOWING LINK CONTROL COMMANDS MUST BE USED TO ENABLE USE OF THIS OPTION:

```
INCLUDE   (PRB$INIT)
INCLUDE   (PRB$TERM)
INCLUDE   (PRB$PERF)
```

EXAMPLE PROBER DISPLAY

PERFORMANCE PROBE DATA

ROUTINE DATA	NUMBER OF EXECUTIONS
CCHAR	1
CINT	5
DIGIO	1

PROBES (ROUTINE - FALSE)

- ENABLES OR DISABLES PRINTING OF SUMMARY OF THE USAGE OF THE PATHS OF EACH OCCURENCE OF THE FOLLOWING PROGRAM CONTROL STATEMENTS:

  - CASE
  - FOR
  - IF
  - REPEAT
  - WHILE

- THE FOLLOWING LINK CONTROL COMMAND MUST BE INCLUDED IF THE PROBES OPTION IS USED

```
INCLUDE   (PRB$INIT)     NOTE 1
INCLUDE   (PRB$TERM)     NOTE 1
INCLUDE   (PRB$COMP)
```

*NOTE1 - THESE TWO ARE THE SAME AS FOR PROBER AND ONLY NEED TO BE INCLUDED ONCE IF BOTH PROBER AND PROBES OPTIONS ARE USED.

EXAMPLE PROBES DISPLAY

COMPLETENESS PROBE DATA

ROUTINE NAME	#PROBES	%ACTIVATED	INACTIVE PROBES
CCHAR	4	75	3
CINT	2	100	
DIGIO	2	100	

TOTAL NUMBER OF PROBES = 8
TOTAL NUMBER OF ACTIVATED PROBES = 7
% OF PROBES ACTIVATED = 87

72COL (STATEMENT - TRUE)

- ENABLES OR DISABLES THE 72 CHARACTER LIMIT FOR SOURCE PROGRAM LINES.

FORINDEX (STATEMENT - FALSE)

- ENABLES OR DISABLES THE ISSUING OF WARNING MESSAGES WHEN NAMES OF FOR CONTROL VARIABLES ARE IDENTICAL TO NAMES OF OTHER ACCESSIBLE VARIABLES.

GLOBAL (ROUTINE - FALSE)

- ENABLES OR DISABLES A LIMITATION OF THE USE OF GLOBAL VARIABLES.

WHEN OPTION IS TRUE, ONLY GLOBAL VARIABLES NAMES IN AN ACCESS DECLARATION ARE ACCESSIBLE WITHIN A ROUTINE.

ROUND (STATEMENT - TRUE)

- ENABLES OR DISABLES THE ROUNDING OF RESULTS OF TYPE DECIMAL.

STANDARD (PROGRAM - FALSE)

- ENABLES OR DISABLES UNIFORM IMPLEMENTATION OF STANDARD TYPES.

- SETTING THE STANDARD OPTION TO TRUE RESULTS IN RANGE OF VALUES OF INTEGER TYPE BEING THE SAME AS THAT OF LONGINT TYPE, -2,147,483,648 THROUGH +2,147,483,647.

PASCAL PRETTY PRINTING STANDARD

1. Each statement must begin on a separate line.

2. Each line shall be less than or equal to 72 characters.

3. Comments that are appended at the end of a line of code and that are continued on successive lines must be written so that continued lines are under the initial comment fragment

4. The keywords REPEAT, BEGIN, END, and RECORD must stand on a line by themselves (except for supportive comments)

5. At least one blank line must appear before LABEL, CONST, TYPE, and VAR declarations; at least three blank lines must appear before PROCEDURE and FUNCTION declarations.

6. At least one space must appear before and after ':=' and '='. One space must appear after ':'.

ALIGNMENT RULES

1. PROGRAM, PROCEDURE, and FUNCTION headings begin at the left margin.

2. The BEGIN-END block for a program, procedure or function shall be lined up with the corresponding heading.

3. Each statement within a BEGIN-END, REPEAT-UNTIL, or CASE statement must be aligned.

INDENTATION RULES

1. The bodies of LABEL, CONST, TYPE, and VAR declarations must be indented from the beginning of the corresponding header keywords.

2. The bodies of BEGIN-END, FOR, REPEAT, WHILE, WITH, AND CASE statements, as well as the bovies of RECORD-END structures, must be indented from their corresponding header keywords.

3. An IF-THEN-ELSE statement must be displayed as follows:

```
IF <expression>          or        IF <expression> THEN
  THEN                                  <statement>
     <statement>                   ELSE
  ELSE                                  <statement>
     <statement>
```

CONVERTING DECIMAL NUMBERS TO HEXADECIMAL NUMBERS

PROBLEM: CONVERT THE NUMBER 732.03125 INTO A HEXADECIMAL NUMBER

THE INTEGRAL PART OF THE NUMBER MUST BE CONVERTED SEPARATELY FROM THE DECIMAL PART.

FIRST, CONSIDER THE INTEGRAL PART OF THE NUMBER. CONVERT 732 INTO A HEX NUMBER.

STEP 1: DIVIDE THE NUMBER BY 16. THE REMAINDER IS THE NEXT DIGIT STARTING TO THE LEFT OF THE DECIMAL POINT. REMEMBER THAT IF THE REMAINDER IS 10, 11, 12, 13, 14, OR 15, YOU MUST WRITE THE HEX DIGIT A, B, C, D, E, OR F.

732/16 = 45 R 12

THE REMAINDER 12 CAN BE REPRESENTED BY THE HEXADECIMAL DIGIT 'C'. THE FIRST DIGIT OF THE NUMBER WE ARE LOOKING FOR IS C.

STEP 2: CONTINUE DIVIDING THE QUOTIENT BY 16 UNTIL THE QUOTIENT IS ZERO. EACH TIME YOU DIVIDE BY 16, THE REMAINDER WILL BE THE NEXT DIGIT OF THE NUMBER WE ARE LOOKING FOR.

45 / 16 = 2 R 13

THE REMAINDER 13 MAY BE REPRESENTED BY THE HEXADECIMAL DIGIT 'D'. D BECOMES THE NEXT DIGIT OF OUR NUMBER.
--> D C.

2 / 16 = 0 R 2

2 BECOMES THE NEXT DIGIT OF OUR NUMBER. SINCE THE QUOTIENT OF OUR LAST DIVISION WAS ZERO, WE ARE THROUGH. THE INTEGRAL PORTION OF OUR HEXADECIMAL NUMBER IS 2DC.

NOW CONSIDER THE DECIMAL PORTION OF THE NUMBER. CONVERT .03125 TO HEX.

STEP 1: MULTIPLY THE NUMBER BY 16. THE DIGIT TO THE LEFT OF THE DECIMAL POINT WILL BE THE NEXT DIGIT OF OUR NEW NUMBER. REMEMBER IF THE NUMBER TO THE LEFT OF THE DECIMAL IS 10, 11, 12, 13, 14, OR 15, YOU MUST WRITE THE HEXADECIMAL DIGIT A, B, C, D, E, OR F.
CONTINUE MULTIPLYING BY 16 UNTIL THE DECIMAL PORTION OF THE NUMBER IS ZERO.

```
     .03125
X         16
-----------
  __.50000
```

THERE IS NO DIGIT TO THE LEFT OF THE DECIMAL SO OUR FIRST DIGIT IS '0'. OUR NUMBER SO FAR IS .0

```
     .50000
X         16
-----------
   8.00000
   --
```

THE NUMBER TO THE LEFT OF THE DECIMAL POINT IS 8. THE NEXT DIGIT OF OUR NUMBER IS 8. THE NUMBER IS NOW .08 SINCE THE DECIMAL PORTION OF THE ANSWER TO THE LAST MULTIPLICATION IS ZERO, WE ARE THROUGH. THUS:

$$.03125_{10} = .08_{16}$$

PUTTING BOTH HALVES OF THE NUMBER TOGETHER WE HAVE

$$732.03125_{10} = 2DC.08_{16}$$