INTRODUCTION
TO
PL/B PROGRAMMING

Sunbelt Computer Software

DISCLAIMER

Document Revision: 1.3

TABLE OF CONTENTS

TABLE OF CONTENTS

TABLE OF CONTENTS

TABLE OF CONTENTS

TABLE OF CONTENTS

TABLE OF CONTENTS

TABLE OF CONTENTS

COURSE OBJECTIVES

- Understand the basic concepts of PL/B programming.
- Understand the structure of the PL/B language.
- Learn how to implement PL/B design concepts.
- Learn how to write, compile, and execute PL/B programs.
- Learn how to make full use of PL/B's powerful features.
- Explore the extensive disk access methods of PL/B.

CHAPTER ONE

INTRODUCTION AND OVERVIEW

KEY FEATURES OF PL/B

The key features of PL/B are outlined as follows:

- The language history
- The language level
- Operating system portability
- Highly efficient object code
- Support of structured programming
- Separate compilation of modules

LANGUAGE HISTORY

The evolution of PL/B has been a long process:

Datapoint

- DOS Datashare
- RMS Databus
- RMS Standard Databus

Sunbelt

- MS-DOS Products
  - SUNDB86A
  - SUNDB86M
- Linux products
  - PLBCMP
  - PLB
- Windows products
  - PLBWIN
  - PLBCE
  - PLBNET
- Server Products
  - Application Server
  - Data Manager
  - Web Server
- Android/iOS Apps
  - PlbWebCli

PL/B became an ANSI Standard Language in December 1994.

LANGUAGE LEVEL

In terms of language level, PL/B is a-high level language.

PL/B is a business programming language well suited to the needs of both small and large companies.

PL/B has strong user interface capabilities.

String handling verbs are at the heart of PL/B.

The language supports text, indexed sequential and associative index file access methods.

Communication verbs allow access to sockets.

Database verbs provide additional access methods to data.

Printing verbs provide access to printers and PDF generation.

Web technology verbs allow access to REST, HTTP, and SMTP protocols.

OPERATING SYSTEMS PORTABILITY

PL/B programs port to different computers with relative ease.

Sunbelt provides runtimes for Windows, and Linux.

PL/B programs running under the Web Server can be accessed by any computer with web browser software.

The publication of the ANSI standards for PL/B and the emergence of compilers meeting those standards provide an even better opportunity than before to write portable PL/B programs.

RICH SYNTAX

PL/B was conceived as an easy-to-learn language oriented toward data entry applications.

PL/B has grown to be a robust and full-featured language. It is used today on micro, mini and mainframe computers.

Portability, efficient code, and rich syntax contribute to PL/B's popularity.

Most of the facilities in PL/B are aimed at letting a programmer produce data processing applications that get the job done as quickly and easily as possible.

HIGHLY EFFICIENT OBJECT CODE

The efficiency of the object code is an advantage that interpretive languages have over other languages.

PL/B object code has a reputation for being compact and fast when compared to code from programs written in other high-level languages.

SUPPORT OF STRUCTURED PROGRAMMING

PL/B lends itself naturally to structured modular programming.

Three of the goals of structured programming are to write programs that are:

- Modular in nature
- Readable
- Easily modified

An application in PL/B can be written as a sequence of programs that in turn may contain subprograms.

Related subprograms can be collected in a file that can be tested, compiled, and debugged separately.

SEPARATE COMPILATION OF MODULES

Most implementations of PL/B allow code to be kept in separately compiled files.

In addition to simplifying testing and debugging, separate compilation of source code can be a time saving feature because a change made to one file does not require recompiling the whole program.

Choosing the subprograms in each file carefully makes program modification easier.

CHAPTER TWO

FUNDAMENTALS OF PL/B PROGRAMS

THE FORM OF A PL/B PROGRAM

Consider the following simple PL/B program. It illustrates several important features of the PL/B language.

```
*
.Sample Program
.
NAME   DIM     20

PHONE  DIM     20             // Customer phone number
AMOUNT FORM    7.2            // Payment Amount
.
       KEYIN   *ES,*P=10:10,"Name: ",NAME:
               *P=10:12,"Phone: ",PHONE:
               *P=10:14,"Payment: ",AMOUNT
.
       DISPLAY *HD,NAME," - ",PHONE," - ",AMOUNT
       STOP
```

THE FORM OF A PL/B PROGRAM - continued

Within PL/B source files,

- In PL/B, lines beginning with a plus (+), asterisk (*), or period (.) are comment statements.
- Line comments begin with the comment identifier (//) and extend to the end of the line.
- Data definitions are normally placed at the beginning of the source file or in separately included files.
- Execution begins with the first line that doesn't contain data definition and isn't a comment line.
- Execution proceeds sequentially line by line unless altered by a program control statement.
- Execution ends upon reaching the end of the source file or a STOP statement.

EXERCISE 1

1. Using the Sunbelt PL/B IDE Studio, click "File" and then "New Project" in the menu.
2. Assign a project name "labs".
3. Click "Browse" and create a folder for your lab exercises (i.e., "\labs").
4. Under "Compiler Output", click "Use Working Directory".
5. Click "OK"
6. Click "File" and then "New File" in the menu.
7. Type in "lab1.pls" for the file name and then click "Save".
8. Respond "Yes" to "Add New File as Program to Project?".
9. Type the following lines in the editing window.

```
*
. lab1.pls
.
REPLY     DIM       1
.
          KEYIN     "Hello, world ",REPLY
```

10. Click project "Build and Execute" or click the lightning bolt tool from the tool bar to compile the program.

INSTRUCTION TYPES

The PL/B language has four (4) basic types of instructions:

1. Compiler directives
2. Data definitions
3. Program execution
4. Comment Line

The first three instruction types use a format similar to the following:

```
LABEL     OPERATION      OPERAND(S)     COMMENTS
```

DATA DEFINITION LABELS

There are two types of data definitions: Character and Numeric.

Data Definitions may be declared anywhere within the program provided it is prior to being referenced by an operation.

Data Definitions are identified by a unique label.

Data Definitions reserve memory for the defined variables.

Data Definition Labels must adhere to the following rules:

- The LABEL must begin in the first column and may not be the only item on the line.
- The LABEL references the Data Definition declared on the line.
- The amount of memory required for a Data Definition is greater than just the amount required for it contents.

Examples:

```
NAME      DIM       30
BALANCE   FORM      7.2
DATA      FILE
```

PROGRAM EXECUTION LABELS

Program Execution Labels are execution points identified by a unique LABEL. Program Execution Labels must adhere to the following rules:

- They must begin in the first column of the line.
- They may be the only item on the line.
- They reference the first operation that follows.

Example:

```
1 RESET0    MOVE      "0" TO PAGECNTR
  ADDONE
2         ADD       "1" TO PAGECNTR
3         COMPARE   "60" TO PAGECNTR
4         GOTO      ADDONE IF LESS
5         GOTO      RESET0
```

The Program Execution Labels RESET0 and ADDONE reference lines 1 and 2 respectively and are referenced by 5 and 4 respectively.

Program Execution Labels may be alone on a line or may precede a valid PL/B operation.

All PL/B operations between Program Execution Labels are associated with the preceding Program Execution Label.

COMMENT INSTRUCTIONS

Comment Instructions may be placed anywhere in the program.

They are designated by a period (.), semicolon (;), asterisk (*) or plus sign (+) or double slashes (//) and must begin in the first position of the line.

Their primary purpose is for program documentation.

For example, the following lines contain Comment Instructions:

```
+===========================================
. Define the date variable
.-------------------------------------------
DATE      DIM  8    // Gets the date
```

Comment Instructions that begin with a period (.) or a semicolon (;) are simply printed.

Those beginning with an asterisk (*) generate a form feed if there are fewer than twelve lines remaining on the current page. Otherwise, they are simply printed.

Comment Instructions beginning with a plus sign (+) generate a form feed regardless of the number of lines remaining.

The double slash format of the comment instruction is a non-ANSI implementation.

LITERALS

Literals are data strings (alpha, numeric, or a combination) that are not declared as variables. Literals are useful in programs where a constant value is needed. They may be in one of the following formats:

- String - a sequence of characters - alpha, numeric and/or other special characters.
- Character - a single character.
- Number - a decimal integer number.
- Control - binary, decimal, hex or octal representation.

Character literals may be defined in any supported numbering base:

```
Decimal         32
Binary          0b00100000
Hexadecimal     0x20
Octal           040
```

The following rules apply to string literals:

- A string literal must be enclosed in double quotes (").
- A string literal may be continued from one line to another by ending the first line with a pair of colons (::).
- The Form Pointer is always one while the Length Pointer points to the last character.
- String literals are generally the first OPERAND in most operations (excluding DISPLAY, KEYIN and PRINT operations).

LITERALS – continued

The following examples show how string literals may be used as OPERANDS:

```
          ADD       "1" TO PAGENCTR
          MOVE      "YEAR-END" TO TITLE
          DISPLAY   "PAYROLL ENTRY - VERSION 1.0"
          KEYIN     "ENTER PART NUMBER: ",PARTNO
          MATCH     "  " TO REPLY
          SCAN      "YES",ANSWER
```

Control literals must be between the values of 0 and 255 (decimal). Note the following examples:

```
DECESC    INIT      27
BINESC    INIT      0b00011011
HEXESC    INIT      0x1B
OCTESC    INIT      033
          CMOVE     32 TO DECIMAL
          CMOVE     0b00100000 TO BINARY
          CMOVE     0x20 TO HEX
          CMOVE     040 TO OCTAL
```

THE FORCING CHARACTER

Since string literals are enclosed in quotes ("), a technique must be employed to include the quote sign within a literal.

The pound sign (#) has been designated as a Forcing Character for this purpose. The pound sign informs the compiler that the next character is part of the string.

The compiler recognizes the Forcing Character and does not include it. If the Forcing Character is required in the string, it must be used twice for each one desired (i.e., "##" to use # in a string literal).

An alternate method for specifying a (") in a string is also provided. By placing two quotes ("") together in a string literal, the first quote acts as a forcing character for the second quote. The quote is not a universal forcing character, i.e., the only character that may be forced using a quote is a quote.

The Forcing Character rules do not apply in operations where the operand must be a single character only (i.e., CMOVE, CMATCH). Literals used with single character operations may include both the quote sign and the pound sign (i.e., CMATCH "#”" TO INPUT).

LABELS

Labels are subject to the following conditions:

- Labels must use the following naming conventions:

  Valid alphabetic characters ({a-z} or {A-Z}, case insensitive)
  0 through 9
  $ (dollar sign)
  _ (underscore)

- Labels must begin with # if it is a local label within an inclusion file.

- Labels must begin with an alpha character, dollar sign ($), or underscore.

- Labels must be unique.

- Labels must begin in the first column of a source line.

- Labels may be from one (1) to thirty-two (32) characters in length with the maximum default length being thirty-two characters.

- Labels more than the defined label size (default is thirty-two (32) characters) are reduced to the first n-1 and the last character.

LABELS – continued

The following are examples of valid LABELs:

```
A
A1
A1$_ABC
$01
LONGLABEL
```

The following are examples of invalid LABELs:

```
6TEEN
AMT-SOLD
```

OPERANDS

OPERANDs are the PL/B instruction parameters; either the parameters required by OPERATION or the data field definition. The general format of operands is as follows:

```
FIRST SEPARATOR SECOND
```

FIRST is the first operand that may be required by operation.

SEPARATOR is required if a second operand is used and may be either a comma (,) or a valid PL/B preposition.

SECOND is any additional operand (or operands) required by the operation.

If a comma separator is used, it cannot be preceded by a space. Otherwise, a valid preposition (BY, TO, OF, FROM, USING, WITH, IN or INTO as appropriate to operation) must be used with at least one preceding and trailing space.

The following are examples of valid PL/B operand syntax:

```
FIRST,SECOND
FIRST TO SECOND                (TO is a valid separator)
FIRST  TO   SECOND             (multiple spaces may be used)
PRIM,SEC1,SEC2,SEC3            (multiple secondary operands)
```

The following are examples of invalid PL/B operand syntax:

```
FIRST SECOND                   (missing separator)
FIRST ,SECOND                  (space preceding comma)
FIRST INSIDE SECOND            (INSIDE is not a valid separator)
```

OPERANDS – continued

Operations that may have a variable number of operands require a comma as the separator.

If the list exceeds a single line, a colon (:) is used in place of a valid separator to continue the list onto the next line.

A valid separator example within a DISPLAY operation:

```
DISPLAY   *N,"This is the 1st line of data ",DATA1:
          *N,"This is the 2nd line of data ",DATA2:
          *N,"This is the 3rd line of data ",DATA3
```

The same separator example without using line continuation:

```
DISPLAY   *N,"This is the 1st line of data ",DATA1;
DISPLAY   *N,"This is the 2nd line of data ",DATA2;
DISPLAY   *N,"This is the 3rd line of data ",DATA3
```

COMMENTS

Comments are programmer notes that follow any required operand(s) for an operation.

- The final operand and any comments must be separated by at least one space.
- Comments should not be confused with the comment instructions.
- The compiler also allows for the // character pair to denote the start of a comment. This is the preferred method of placing comments at the end of a line.

Example:

```
ADD  "1" to COUNTER      // Giving new result
```

DECLARING VARIABLES

PL/B supports three variable types:

1. Character string

2. ASCII numeric

3. Integers

All numeric instructions require numeric data definitions and most string instructions must use only character data definitions.

Data may be moved between the different field types although certain rules may apply regarding format, truncation, and rounding.

Let's revisit the previous program and examine the variables.

```
*
.Sample Program
.
NAME   DIM     20              // Customer name
PHONE  DIM     20              // Customer phone number
AMOUNT FORM    7.2             // Payment Amount
INFO   INIT    "Data:"         // Display String
.
       KEYIN   *ES,*P=10:10,"Name: ",NAME:
               *P=10:12,"Phone: ",PHONE:
               *P=10:14,"Payment: ",AMOUNT
.
       DISPLAY *HD,INFO,NAME," - ",PHONE," - ",AMOUNT
       STOP
```

DECLARING VARIABLES - continued

DIM, INIT, INTEGER, and FORM are data definition verbs that declare alphanumeric and numeric variables.

The DIM instruction defines a character string variable.

```
[label]   DIM       {size}
```

[label] is the data definition label and {size} is number of characters that may be stored in that variable. Initially the variable is empty.

The INIT instruction both defines a character string variable and initializes it to the specified value.

```
[label]   INIT      "{string}"
```

[label] is the data definition label and {string} is the initial value of the variable. The number of characters that may be stored in the variable is the length of {string}.

DECLARING VARIABLES - continued

The FORM instruction defines a numeric variable. The format of a FORM instruction is:

```
[label]   FORM      {digits}[.{decimals}]
```

[label] is the data definition label. {digits} is number of numeric characters that may be stored to the left of the decimal place. {decimals} is number of numeric characters that may be stored to the right of the decimal place.

The decimal point and number of decimals are optional in FORM statements.

The INTEGER instruction defines a numeric variable. The format of a FORM instruction is:

```
[label]   INTEGER  {size}
```

[label] is the data definition label. {size} is number of bytes used as storage and may be a value of one, two, three, four or eight.

The {size} of the integer determines the maximum value that may be stored.

DISPLAY AND KEYIN

Workstation I/O is accomplished in PL/B using the DISPLAY and KEYIN verbs.

DISPLAY outputs both literals and variables to the screen.

The verb also supports a wide variety of List Controls that allow the programmer a high degree of flexibility in screen layouts. These list control include the following:

List Control	**Effect**
`*ES`	Erase the screen
`*EL`	Erase the line
`*P=x:y`	Move to horizontal position 'x' and vertical position 'y'
`*HU`	Position to the top left corner of the screen
`*HD`	Position to the bottom left corner of the screen
`*R`	Roll the screen up one line
`*COLOR={color}`	Set the foreground color
`*BGCOLOR={color]`	Set the background color

Example:

```
     DISPLAY  *P=1:10,*EL,*COLOR=*RED,"Welcome!":
              *P=1:12,*EL,*COLOR=*BLUE,"Isn't PL/B Fun?"
```

DISPLAY AND KEYIN - continued

The KEYIN instruction accepts input from the keyboard and may also display data on the user's screen prior to accepting the input.

Variables encountered in the operand list allow the operator to type data as is appropriate.

Any character may be entered in an alphanumeric field and only number in a numeric field and the field size is honored.

The Enter Key advances to the next field.

Literals in the operand list of a KEYIN statement are output just as done with a DISPLAY statement.

DISPLAY AND KEYIN - continued

The KEYIN instruction also supports many list controls including most DISPLAY controls. Some addition KEYIN only list controls are:

List Control	Effect
`*DV`	Display the contents of the next variable rather than allowing input in to it.
`*RV`	Retain the value of the next variable if a null entry is made.
`*DVRV={variable}`	Combination of the above two list controls.
`*ESON`	Characters entered are echoed as asterisks.
`*DE`	Allow only numeric characters in the following string variable.
`*T[={seconds}]`	Terminate input after {seconds} and continue the program.
`*IT`	Input text in lowercase (affects all variables until either an *IN list control is encountered or the end of the program.
`*IN`	Revert to normal text input (i.e., uppercase).

Example:

```
NAME      DIM       20
.
          KEYIN     *P=1:10,"Enter your name: ",*RV,NAME
```

EXERCISE 2

1. Write a program that performs data entry using the following field definitions

Information	Data Type	Size
Account #	A/N	4
Name	A/N	20
Address	A/N	35
City	A/N	20
State	A/N	2
Zip	N	5
Balance	N	4.2

A/N = Alpha Numeric, N=Numeric

EXERCISE 2 – continued

2. Once step one is complete, put your program into a loop using the following:

   - Before your first executable statement, insert a label named "Start".

   - At the end of the program put the following:

```
ANS  DIM      1
     KEYIN    *P=1:24,*EL,"Another Entry?  ",ANS
     CMATCH   "Y",ANS
     GOTO     START IF EQUAL
     STOP
```

3. Modify the program to perform the following:

   - If enter is pressed on a field, allow the field to retain the data in the field.

   - Restrict the Account field to only allow numbers and enter text in lower case by default on the remaining fields.

CHAPTER THREE

DATA TYPES AND OPERATORS

FUNDAMENTAL DATA TYPES

PL/B supports three variable types:

- Character string
- ASCII numeric
- Integer fields

All numeric instructions require numeric data definitions and most string instructions must use only character data definitions.

Data may be moved between the different field types although certain rules may apply regarding format, truncation, and rounding.

DIM

DIM instructions define only the Physical Length while INIT instructions define the Physical Length, Physical String, Logical Length, and Logical String.

```
MYVAR     DIM       30
```

The Physical Length is the size specified when the variable is defined.

Each variable contains a Form Pointer that indicates the beginning of the logical string. It also contains a Length Pointer that specifies the end of the logical string.

Also note that:

- The physical size of a character string variable is set at definition time.
- The Length Pointer (LP) may never exceed Physical Length (PL) of a variable.
- The Form Pointer (FP) may never exceed the Length Pointer (LP).
- A null variable is indicated by a Form Pointer (FP) of zero (0).
- DIM variables are blank filled.
- The Physical Length is set to the declared size while both the Length Pointer and Form Pointer are set to zero indicating a Null String.

INIT

The INIT instruction defines a character string variable and assigns it an initial value.

```
MYVAR     INIT      "Hello, World"
LIST      INIT      061,"2",0x33
```

Character string literals must be enclosed in quotation marks.

Control string literals must be valid representations of binary, decimal, hex or octal values.

Also note that:

- The Form Pointer is set to the first character in the string.
- The Length Pointer and Physical Length are set to the last character in the string.
- Other than being initialized by the compiler, INITs behave just as DIMs.

FORM

FORM variables contain numeric data in a valid decimal number format. FORMs may be used in any mathematical and lend themselves to accounting functions.

The format of a numeric variable is set at definition. When a new value is assigned to a FORM, it is aligned to this definition.

The physical length of a FORM variable may not exceed 32 bytes (including the decimal).

The results of any arithmetic instruction are aligned to the decimal.

Spaces are allowed in FORM definitions only if representing suppressed leading zeroes.

A FORM definition may only contain one decimal.

Negative values are indicated by a minus sign (-) immediately preceding the most significant digit. A FORM definition may only have one minus sign.

Examples:

```
FORM7      FORM      7          (valid format)
FORM72     FORM      7.2        (valid format)
FORMP4     FORM      .4         (valid format)
FORMD      FORM      "222.12"   (valid format)
NEG10      FORM      "•••-10"   (valid format)
.
NEG10      FORM      "-•••10"   (invalid format)
NEG10      FORM      "----10"   (invalid format)
NEG10      FORM      "-10•••"   (invalid format)
```

ROUNDING AND TRUNCATION

When arithmetic instructions involving FORMs exceed the size of the destination variable, truncation, rounding or both may occur.

Truncation may occur on either the right (least significant) or left (most significant). Left truncation usually sets the OVER flag.

Two forms of rounding are supported. The first is the traditional method that was used on Datapoint DOS systems. The second method is the method specified by the ANSI standard for PL/B. The default rounding method is the first method. The only difference is when the rounded digit is a five and all lost digits are zero.

If the rounding (next least significant) digit being lost is zero to four, the rounded (next most significant) digit remains unchanged.

In the traditional method, if the rounding digit is five and the lost (next least significant) digit(s) is zero then positive numbers increment the rounded digit by one. Negative numbers leave the rounded digit unchanged. In the ANSI method in all cases the rounded digit is incremented by one.

If the rounding digit is five and the lost digit(s) is greater than zero or if the rounding digit is greater than five, the rounded digit is incremented by one.

INTEGER

The INTEGER instruction defines an unsigned, integer numeric variable of one to four bytes.

Integers are extremely useful for high-speed loop or arithmetic processing.

All INTEGERS are unsigned (positive values only).

If an initial {value} is not specified, the variable is set to zero.

Any operation in which the result is larger than the destination causes the high order (most significant) bytes exceeding the destination variable's size to be discarded and the OVER flag is set TRUE.

An INTEGER may be used any place a FORM is valid except for KEYIN, DISPLAY, and PRINT.

Examples:

```
INT4      INTEGER      4
INT1      INTEGER      1,"0"
INT2      INTEGER      2,"300"
```

VARIABLE ASSIGNMENTS

The MOVE instruction transfers the contents of the source operand to the destination operand. The contents of the source operand remain unchanged.

```
          MOVE      {source}{sep}{dest}
```

{source} is a variable or literal that remains unchanged.

{dest} is the destination variable that receives the value of {source}.

Examples:

```
STRING    DIM       30
NAME      DIM       20
.
          MOVE      "Hello",STRING
          MOVE      NAME,STRING


COUNTER   FORM      2
INDEX     FORM      "15"
.
          MOVE      "99",COUNTER
          MOVE      INDEX,COUNTER
```

OPERATORS AND EXPRESSIONS

In this section, we will investigate how PL/B implements the following:

- Arithmetic operators
- Relational and Logical operators
- Some other useful operators

These operators allow you to manipulate data in your programs and to perform tests on the data.

ARITHMETIC OPERATORS

PL/B supplies the following binary (two operand) arithmetic operators:

Operator	Meaning
+	Addition
-	Subtraction
*	Multiplication
/	Division

Examples:

```
     MOVE       (INDEX + 2),NWORK7
     MOVE       ((R * R) * 3.14159),C
```

RELATIONAL AND LOGICAL OPERATORS

Relational operators compare two values. The operators are listed below:

Operator	**Comparison**
<	Is less than
<=	Is less than or equal to
=	Is equal to
>=	Is greater than or equal to
>	Is greater than
!= or **<>**	Is not equal to

Logical operators normally take relational expressions as operands. The "!" operator takes one operand. The others take two operands.

Operator	**Meaning**
& or **AND**	And
| or **OR**	Or
! or **NOT**	Not

Unlike some languages, PL/B does not use a special data type to represent true and false. Instead,

- False is represented by the value zero
- True is represented by any value other than zero

LOGICAL EXPRESSIONS

Expressions used with structured code are a way of altering program flow based on a certain condition (or conditions) being evaluated and found to be true. This allows many PL/B instructions to be replaced with code that is more efficient and easier to read.

Expressions may also be used anywhere a numeric variable is normally used.

Expression must be enclosed in parenthesis.

Examples of logical expressions:

```
     (8 > 2 & 12 < 30)          is true

     !(8 > 2 AND 12 < 30)       is false
```

CHAPTER FOUR

PROGRAM CONTROL STATEMENTS

PROGRAM CONTROL STATEMENTS

The execution of a PL/B program begins at the first executable instruction.

It continues sequentially until a Program Control Instruction alters the sequence, the program terminates or it is interrupted to execute another program.

Some Program Control Instructions have the ability to alter the flow conditionally based upon Flags. These flags are set to true or false and are tested as follows:

```
     GOTO      LABEL IF OVER
     GOTO      LABELX IF NOT EOS
```

PL/B supports the conventional method of program control, (GOTO, CALL, RETURN) and structured code (IF, ELSE, ENDIF, LOOP, REPEAT, etc.). These two types may be intermixed within the same program.

FLAGS

Program execution may be governed by certain indicators or Flags. These Flags may be either set (TRUE) or not set (FALSE). There are three types of Flags:

- Condition
- Pseudo Condition
- Function

Condition Flags

Four Condition Flags are available:

- EQUAL (or ZERO)
- LESS
- OVER
- EOS

The EQUAL (or ZERO) Flag indicates a test has matched or a mathematical instruction has generated a ZERO result.

The LESS Flag may be used for testing relative sizes or values in a less than (<) fashion.

The OVER Flag is not used as a greater than (>) evaluation. It typically indicates that a file access has set a BOF or EOF condition. Also, the OVER Flag can be set when a mathematical instruction result is too small and the result value is truncated.

The EOS (End Of String) Flag determines when manipulation of a character string has reached certain limitations.

SETFLAG

Condition flags may be set as the result of string or numeric comparison or directly using the SETFLAG instruction.

Commonly, the condition flags can be set in a subroutine to report a condition to the calling program.

Examples of SETFLAG are

```
SETFLAG   ZERO
SETFLAG   NOT EOS
```

Pseudo Condition Flags

There is one Pseudo Condition Flag: the GREATER Flag. While not actually a Condition Flag, it has been included to ease programming.

The TRUE condition of the GREATER Flag signifies that both the LESS and EQUAL (or ZERO) Flags are FALSE, while a FALSE GREATER Flag signifies that one or both of them are TRUE.

Therefore, it may be used for greater than (>) testing.

Function Flags

There are also flags associated with 40 Function Keys and 11 other Control Keys (Function Flags), and an all-encompassing FKEY Flag.

Though all of the following Function Flags are supported by the compiler, their support within a specific program or application may be limited by the terminal or system in use.

All of the Function Flags, including the FKEY Flag, are cleared at the beginning of each KEYIN instruction.

The appropriate Function Flag is set whenever a supported Function or Control Key is entered at the terminal.

The FKEY Flag is also set when any of the Function Flags is set. Only one of the Function Flags may be set at a time, since it is cleared either once it has been tested and determined to be TRUE or upon occurrence of another KEYIN.

The FKEY Flag determines if any Function Flag has been set. Any test of the FKEY Flag clears all function key flags.

Function Flags – continued

The Function Flags are as follows:

Flag	Set when ...
FKEY	any of the following are depressed.
Fnn	`nn' corresponds to function keys 1 to 40.
UP	the UP arrow key was depressed.
DOWN	the DOWN arrow key was depressed.
LEFT	the LEFT arrow key was depressed.
RIGHT	the RIGHT arrow key was depressed
INS	the INSert key was depressed.
DEL	the DELete key was depressed.
PGUP	the PAGE UP key was depressed.
PGDN	the PAGE DOWN key was depressed.
HOME	the HOME key was depressed.
END	the END key was depressed.
ESC	the ESCAPE key was depressed.
ESCAPE	the ESCAPE key was depressed.
TAB	the TAB key was depressed.
BACKTAB	the Shifted TAB key was depressed.

GOTO

The GOTO instruction alters the program execution sequence by branching to a location designated by a Program Execution Label.

```
GOTO      {dest}
GOTO      {dest} IF [NOT] {flag}
GOTO      {dest} IF {expression}
```

{label} designates the Program Execution Label to which the program execution diverts.

Function Flags (Function keys) are cleared if tested and found to be TRUE. All other Flags remain unchanged.

The third format branches to {label] only if {expression} evaluates to a true condition. Otherwise, execution continues with the next sequential instruction.

STOP

The STOP instruction provides a method for terminating a program's execution.

Depending upon how the terminating program was initiated, STOP either returns control to the MASTER program or to the operating system.

Examples:

```
STOP

STOP      IF ZERO

STOP      IF F1
```

CHAIN

The CHAIN instruction loads another PL/B program and passes control to it.

```
     CHAIN    {program}
```

{program} must be provided and must contain the name of the program to be executed.

A CFAIL error occurs if the program file is not found, is incompatible with the initiating program, has common data area misalignment or has global data area incompatibility.

Any TRAP that has been set is cleared.

All Flags (Condition, Pseudo Condition and Function) are cleared (set to FALSE).

The subroutine call stack is cleared.

Examples:

```
NEXTPGM   INIT      "ARREPORT"

          CHAIN     NEXTPGM

          CHAIN     "ARREPORT MYFILE 1-10"
```

CALL

The CALL instruction initiates a subroutine.

Repetitive procedures may be coded into a single subroutine and initiated through a CALL.

When the CALL instruction is executed, the address of the next sequential instruction is placed as the top element of the subroutine call stack and control is transferred to the address of the first operand.

Examples:

```
     CALL      SUB1

     CALL      SUB2 IF ZERO

     CALL      SUB3 IF (X < 1)
```

RETURN

The RETURN instruction terminates a subroutine, returning program control to the top element of the subroutine call stack (LIFO - Last In First Out).

Examples:

```
     RETURN

     RETURN    IF ZERO

     RETURN    IF (Y > 4)
```

CALL/RETURN Example

```
NAME      DIM       30   // Customer Name
.
*
.Program Execution Begins
.
          .
          .
          .
          CALL      GETCUST
          DISPLAY   *HD,*R,"Customer: ",NAME
          .
.
.
          STOP
*
.Get the Customer Name
.
GETCUST
          KEYIN     *P=1:5,"Enter Name: ",NAME
          STOP      IF F1     // Allow exit
          RETURN
```

TRAP

The TRAP instruction is similar to the CALL instruction except that it only initiates the specified subroutine if a defined event occurs after the TRAP was executed. Some of the defined events normally result in program termination.

```
TRAP     {routine} IF {event}
TRAP     {routine} GIVING {message} IF {event}
TRAP     {routine} NORESET IF {event}
```

Any system error messages are placed in S$ERROR$ unless the GIVING option is used. The GIVING option designates that {message} receives the system error message.

Once a TRAP has been taken, it is cleared (potentially eliminating the protection from program termination) and must be reset unless the NORESET option is specified. The NORESET option indicates the TRAP is not cleared, remaining set until an appropriate TRAPCLR instruction or program termination.

The scope of a TRAP should be as small as possible.

TRAPCLR

The TRAPCLR instruction clears a previously set trap.

Example:

```
TRAP       PREP IF IO
OPEN       FILE,"DATA"
TRAPCLR    IO
```

NORETURN

The NORETURN instruction removes the top element from the subroutine call stack. This cancels the ability to RETURN from the last CALL instruction.

```
          NORETURN
```

Example:

```
FILENAME  DIM  30
.
START
          KEYIN     "Enter the file name: ",FILENAME
.
          TRAP      NOFILE IF IO
          OPEN      FILE,FILENAME
          TRAPCLR   IO
          GOTO      MENU
.
NOFILE
          NORETURN
          GOTO      START
*
.Show the menu
.
MENU
          ...
```

TRAP EVENTS

Disk Access Events

Event	An error occurred while attempting to ...
IO	access the disk .
CFAIL	CHAIN to another program.
DBFAIL	access an ODBC database
FORMAT	Read numeric data.
PARITY	Read or write to the disk.
SPOOL	route the print output to the designated or default printer device or file.

Keyboard Events

Event	Key pressed
INT	the operating system interrupt sequence
ESC	Escape
{char}	{char} key
Fnn	Function key

In addition, any special key defined within the screen definition file in use (UP, DOWN, LEFT, RIGHT, INS, DEL, PGUP, PGDN, HOME, or END) may also have a TRAP set for each.

TRAP – Continued

Example 1:

```
     TRAP      NOMASTER IF IO
     OPEN      APMASTER,"ACCTSPAY"
```

Example 2:

```
     TRAP      NOPROGRM NORESET IF CFAIL
     CHAIN     "APAYUPTE"
```

Example 3:

```
     TRAP      HELP NORESET PRIOR IF F1
     KEYIN     "Enter Account Number: ",ACCTNO;
     ...

HELP
     KEYIN     "Use the search function to review ":
               "account numbers", ANS;
     RETURN
```

EXERCISE 3

Using the previous exercise, provide the following:

1. Allow immediate program termination anytime the escape key is pressed.

2. Allow moving to the previous field if the up arrow key is pressed.

3. Chain to Exercise 1 if the F1 Key is pressed. Be sure to include logic to handle chaining errors

4. Note what happens when LAB1 terminates.

IF CONSTRUCT

There are several forms of the IF selection structure. The structure modifieds the program execution based on a test. The test may be a condition flag or an expression.

The simplest form has the following syntax:

```
IF        (TEST)
  STATEMENT1
ENDIF
```

The *test* is evaluated first. If it is true (any non zero value), statement1 will be executed; otherwise, statement1 will be bypassed.

More than one statement may be associated with the test.

```
IF        (TEST)
  STATEMENT1
  STATEMENT2
  STATEMENT3
ENDIF
```

Another form of the *if* statement is to implement two-way selection by using the optional *else* keyword.

```
IF        (TEST)
  STATEMENT1
ELSE
  STATEMENT2
ENDIF
```

Statement2 is executed only if the result of the *test* is false (zero).

IF CONSTRUCT - continued

By nesting *if-else* statements, we obtain a particularly useful multi-way selection structure. Nested *if-else* statements are typically written as follows:

```
     IF         (TEST1)
       STATEMENT1
     ELSEIF     (TEST2)
       STATEMENT2
     ELSEIF     (TEST3)
       STATEMENT3
     .
     .
     .
     ELSE
       STATEMENTn;
     ENDIF
```

In the default form, if none of the expressions are true, the default *else* action is executed. By including a default action, we are guaranteed that some action is executed.

EXAMPLE 1:

```
CODE FORM      1

     KEYIN     "Enter a number: ",CODE
.
     IF        (CODE = 1)
       DISPLAY "Freshman"
     ELSEIF    (CODE = 2)
       DISPLAY "Sophomore"
     ELSEIF    (CODE = 3)
       DISPLAY "Junior"
     ELSE
       DISPLAY "Senior"
     ENDIF
```

IF CONSTRUCT – continued

EXAMPLE 2:

```
IF        (X >= 1 AND X <= 100)
  DISPLAY "X is in range"
ELSE
  DISPLAY "X is NOT in range"
ENDIF
```

EXAMPLE 3:

```
IF        (CH >= "a" & CH <= "z")  // if lower case
  SUB     "32",CH                  // convert to upper
ELSEIF    (CH >= "A" & CH <= "Z")  // if upper case
  ADD     "32",CH                  // convert to lower
ENDIF
```

EXERCISE 4

Write a program that:

1. Allows the user to enter three FORM 2s named *Min*, *Max*, and T*est*. These numbers represent a minimum, a maximum, and a test value respectively.

2. Terminate the program if the F3 key is pressed when prompted for any of the three values.

3. Call a subroutine named "TestValues". TestValues should determine if Test is between Min and Max and set the ZERO flag if it is. Otherwise, the ZERO flag should be set false.

4. Once the subroutine returns, the program should display the results of the comparison and wait for an operator acknowledgement.

5. The program should repeat the process until the escape key is pressed.

Sample execution:

```
Enter values for
Min: 5
Max: 10
Test: 8

8 is between 5 and 10.


Enter values for
Min: 5
Max: 10
Test: 15

15 is NOT between 5 and 10.
```

BRANCH

The BRANCH instruction alters the program execution sequence to a specific Program Execution Label based on an index value.

```
     BRANCH   {index} TO LABEL1,LABEL2,...
```

The value of {index} remains unchanged and designates Program Execution Label the program execution is to shift. If the value is one, program execution changes to the first Program Execution Label (LABEL1). If the value is two, program execution changes to the second label (LABEL2), etc.

If {index} has a negative value, a zero value or a value greater than the number of Program Execution Labels in {labels}, control is passed to the next sequential instruction (i.e., no branching occurs).

Additional lines may be used by ending each continued line with a colon (:).

Null entries in the list are allowed and are signified by two consecutive commas. When the index value is associated with a null entry, no BRANCH operation is performed just as when the index value is out of range or invalid.

```
     BRANCH   Index to Sunday,Monday,Tuesday:
              Wednesday,Thursday,Friday
*
.Saturday Routine
.
Saturday
     ...
     GOTO     START
*
.Sunday Routine
.
Sunday
     ...
     GOTO     START
```

PERFORM

The PERFORM instruction initiates a subroutine based on an index. When the instruction is executed, the address of the next sequential instruction is placed in the top element of the subroutine call stack.

```
     PERFORM   INDEX TO LABEL1,LABEL1,...
```

The value of {index} remains unchanged and designates which Program Execution Label in {labels} should be called. If the value is one, the subroutine designated by the first Program Execution Label in {labels} is called. If the value is two, the subroutine designated by the second label in {labels} is called, etc.

A RETURN instruction within the destination subroutine resumes execution at the next sequential instruction after the PERFORM. A NORETURN within the destination subroutine removes the stack return reference of the last PERFORM.

Null entries in the list are allowed and are signified by two commas with no label name between them. When the index value is associated with a null entry, no PERFORM operation is performed the same as when the index value is out of range or invalid.

```
    PERFORM  Index to Sunday,Monday,Tuesday:
             Wednesday,Thursday,Friday,Saturday
    GOTO     MENU
*
.Saturday Routine
.
Saturday
    ...
    RETURN
*
.Sunday Routine
.
Sunday
    ...
    RETURN
```

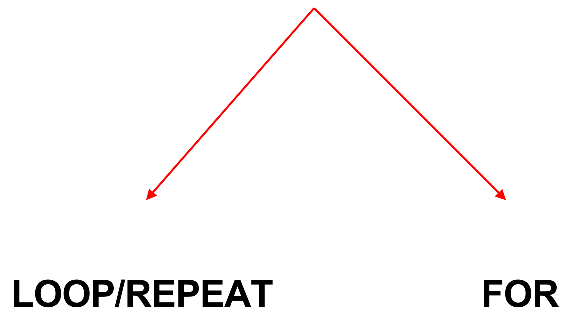
EXERCISE 5

Write a program that

1. Display a menu of the previous exercises
2. Allows user selection via a selection number
3. Execute the requested program
4. Display a message if an invalid selection is entered
5. Be prepared for chaining errors

LOOPING STRUCTURES

There are two repetition structures in the PL/B language:



LOOP/REPEAT

The LOOP/REPEAT structure has the following general form:

```
     LOOP
          [zero or more statements]
     [{TEST}]
          [zero or more statements]
     REPEAT   [{Test}]
```

A flag, NOT flag, expression, or NOT expression test is required.

LOOP is a starting point for a group of instructions to be repeated.

REPEAT returns the program execution to the first instruction after LOOP if the test results are positive.

The program instructions between LOOP and REPEAT are processed sequentially in a repetitive fashion.

When the WHILE structure generates a negative test result or the UNTIL generates a positive test result, program control is transferred to the first instruction following the REPEAT structure.

The execution of a LOOP/REPEAT structure may be terminated at any point in the structure, even inside of nested IF/ENDIF structures by using the BREAK instruction.

The following program prints the numbers from 0 to 10 inclusive.

```
X    FORM      2
.
     LOOP
     WHILE     (X <= 10)
     DISPLAY   *N,X
     INCR      X
     REPEAT
```

LOOP/REPEAT - continued

The following program prints from 10 to 1 inclusive.

```
X    FORM      "10"
.
     LOOP
     DISPLAY   *N,X
     DECR      X
     REPEAT    UNTIL ZERO
```

The following loop test is an expression

```
X    FORM      2
.
     LOOP
     WHILE     (X < 10)
     DISPLAY   *N,X
     INCR      X
     REPEAT
```

The following loop is never executed:

```
X    FORM      2
.
     LOOP
     WHILE     (X)
     DISPLAY   *N,X
     INCR      X
     REPEAT
```

EXERCISE 6

Write a program that:

1. Calls a subroutine named *prompt* that displays the following menu and allows user input.

A.	Add record
D.	Delete record
C.	Change record
F.	Find a record
L.	List all records

   Enter your selection:

2. Let the program redisplay the menu inside a loop if the input is not 'A', 'D', 'C', 'F', or 'L'.

3. Do not use *if* within the loop.

4. Terminate the loop once the user enters a valid response and exit the subroutine.

5. Display the response character before ending the program.

THE FOR LOOP

The *for* loop has the following general form:

```
FOR  {target}{sep}{start}{sep}{end}[{sep}{incr}]
{statements}
REPEAT
```

Upon initial entry into the FOR statement loop, the {start} value is assigned to the {target} variable.

Upon subsequent loops through the FOR statement the {target} value is incremented by the value of the {incr} operand.

If the {incr} operand is not specified, the default incrementing value is one (1).

Executing a GOTO in a FOR statement loop is not recommended. If this is done, indeterminate FOR loop program processing can be expected.

The FOR statement termination can occur upon initial entry should the initial {start} and {end} values meet the termination criteria.

THE FOR LOOP - continued

EXAMPLE 1:

```
FOR       COUNTER FROM "1" TO "9" USING "2"
DISPLAY   COUNTER
REPEAT
```

This loop displays the value of counter starting with a value of one and ending with a value of eight in increments of two.

EXAMPLE 2:

```
FOR       COUNTER,"100","1","-1"
DISPLAY   COUNTER
REPEAT
```

This loop displays the value of counter in reverse order starting with a value of one hundred and ending with a value of one.

EXAMPLE 3:

```
FOR       COUNTER FROM START TO END
...
REPEAT
```

This loop repeats starting with the value of the START variable and ending with the value of the END variable in increments of one.

THE FOR LOOP - continued

The following program demonstrates nested *for* loops.

```
I    FORM      1
J    FORM      1
K    FORM      1
.
     FOR  I FROM 0 TO 2
          FOR J FROM 0 TO 2
               FOR K FROM 0 TO 2
                    DISPLAY *N,I," ",J," ",K
               REPEAT
          REPEAT
     REPEAT
```

EXERCISE 7

Perform the following exercise in two steps:

1. This exercise is intended to gain some experience with nested *for* loops. The goal is to capture and print all the combinations of a dollar change, starting from half dollar down to pennies. It would be appropriate to print the number of iterations for the sake of comparison.

   The correct answer is 292 combinations.

   combinations: Represents the number of different ways possible you can arrange half dollars, quarters, dimes, etc., to make a full dollar.

   iterations: Represents how many times the most inner for loop was repeated.

   HINT: Define numeric variables for each denomination. Use nested *for* loops to initialize and increment the denomination variables. Within the innermost *for* loop, increment a counter (iterations). Also, determine if the value of the coins equal a dollar. If it does, increment a counter (combinations).

2. The number of iterations should not be greater than 100,000. It is possible to have 292 combinations in only 292 iterations, but it is not required.

BREAK AND CONTINUE

The BREAK and CONTINUE verbs alter the program execution sequence.

BREAK terminates the currently active LOOP/REPEAT even if there are several levels of active IF/ENDIF instructions.

CONTINUE forces execution to resume at the currently active LOOP instruction even if there are several levels of IF/ENDIF instructions.

```
     BREAK
     BREAK     IF [NOT] {condition}

     CONTINUE
     CONTINUE  IF [NOT] {condition}
```

Break Example:

```
A    DIM       2
.
     LOOP
     KEYIN     *N,"Enter a character: ",A;
     BREAK     IF F1
     DISPLAY   *N,"Character: ",A;
     REPEAT
```

Continue Example:

```
A    DIM       2
.
     LOOP
     KEYIN     *N,"Enter a character: ",A;
     CONTINUE  IF (A=5)
     DISPLAY   *N,"Character: ",A;
     REPEAT
```

SWITCH

The SWITCH instruction provides a multi-way branch that allows a series of conditional statements to evaluate multiple conditions.

```
SWITCH    {source}
CASE      {case1}
...
CASE      {case2}
...
DEFAULT
...
ENDSWITCH
```

The SWITCH and ENDSWITCH define the bounds of the construct.

No executable statements are allowed between the SWITCH and the first CASE statement.

At least one CASE statement must be specified between the SWITCH and the ENDSWITCH statements. Any number of additional CASE statements may be used.

An optional DEFAULT statement may be specified after the last CASE statement but before the ENDSWITCH. Only one DEFAULT statement is allowed.

The operand specified for the SWITCH statement defines a selection criteria used when evaluating each CASE statement.

Each CASE statement operand is compared or matched to the SWITCH operand.

The data type for a CASE operand must be the same as the corresponding operand for the SWITCH statement.

SWITCH – continued

Example:

```
DOW   FORM        1
DVAR1 INIT        "A"
.
      CLOCK       WEEKDAY,DOW
.
      DISPLAY     "The current day of the week is ";
.
      SWITCH      DOW
      CASE        "1"
      DISPLAY     "Sunday";
      CASE        "2"
      DISPLAY     "Monday";
      CASE        "3"
      DISPLAY     "Tuesday";
      CASE        "4"
      DISPLAY     "Wednesday";
      CASE        "5"
      DISPLAY     "Thursday";
      CASE        "6"
      DISPLAY     "Friday";
      CASE        "7"
      DISPLAY     "Saturday";
      DEFAULT
      DISPLAY     "Invalid";
      ENDSWITCH
.
      DISPLAY         "."
```

SELECT

The SELECT instruction provides a multi-way branch that allows a series of conditional statements to evaluate multiple conditions.

At least one WHEN statement must be specified between the SELECT and the ENDSELECT statements. Any number of additional WHEN statements may be used.

An optional DEFAULT statement may be specified after the last WHEN statement but before the ENDSELECT. Only one DEFAULT statement is allowed.

Each WHEN statement operand is compared or matched to the corresponding SELECT operand defined.

Each WHEN statement operand may also be specified as more than one selection criteria.

If a WHEN statement operand has more than one selection criteria and defines a range, the selection criteria items are separated by the keyword 'THRU'. This allows the matching condition to occur when the SELECT operand is determined within the range specified inclusive.

If a WHEN statement operand has more than one selection criteria item, the keyword 'OR' must separate the selection criteria items. In this case either selection criteria item matching the corresponding SELECT operand may satisfy a matching condition.

SELECT – continued

Example:

```
DVAR1     DIM       1
DVAR2     DIM       1
.
          KEYIN     "Enter First Character":
                    *UC,DVAR1,*N:
                    "Enter Second Character:":
                    *UC,DVAR2
          SELECT    USING DVAR1,DVAR2
.
          WHEN      "A" THRU "Z", "A" THRU "Z"
          DISPLAY   "Both characters alpha"
          WHEN      "A" THRU "Z", "0" THRU "9"
          DISPLAY   "First character alpha - second number."
          WHEN      "0" THRU "9", "A" THRU "Z"
          DISPLAY   "First character number - second alpha."
          WHEN      "0" THRU "9", "0" THRU "9"
          DISPLAY   " Both characters numbers"
          DEFAULT
          DISPLAY   "Try again: A-Z or 0-9."
.
          ENDSELECT
```

EXERCISE 8

Write a program that

1. Displays the following menu on the screen.

A.	Add Record.
D.	Delete Record.
C.	Change Record.
F.	Find Record.
L.	List All Records.
Q.	Quit.

2. The menu should be displayed within an infinite loop.

3. The user should be able to terminate the program by selecting the letter "Q".

4. Use a *switch* construct to invoke the appropriate subroutine; that is, when the letter "A" is selected, the subroutine *Add* should be called.

5. Include, for the time being, only a DISPLAY statement in each subroutine. For example, *Add* should be as follows:

```
Add
    DISPLAY    *N,"Add called."
    RETURN
```

EXERCISE 9 - OPTIONAL

1. Write a program that prints a calendar for a year. Prompt the user for which day of the week January first is on and whether the year is a leap year. The day that January first is on is coded as:

```
1 = Sunday
2 = Monday
2 = Tuesday
4 = Wednesday
5 = Thursday
6 = Friday
7 = Saturday
```

2. The leap year condition is coded as

```
N = not leap year
Y = leap year
```

Sample Input/Output:

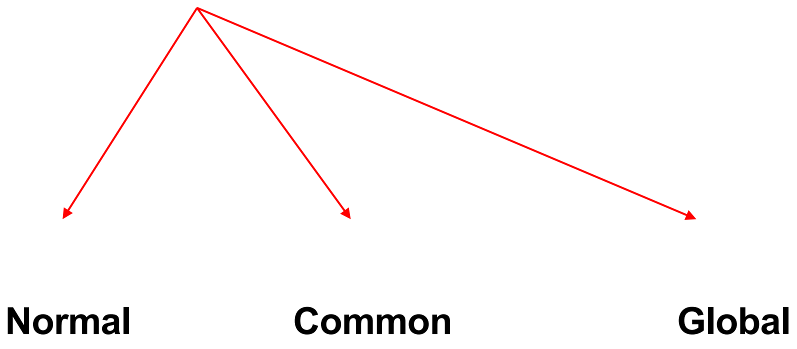
```
Enter day and leap year codes: 4 N

               January

Sun     Mon  Tue  Wed  Thu  Fri  Sat
                       1    2    3
4       5    6    7    8    9    10
11      12   13   14   15   16   17
18      19   20   21   22   23   24
25      26   27   28   29   30   31

               February

Sun     Mon  Tue  Wed  Thu  Fri  Sat
1       2    3    4    5    6    7
8       9    10   11   12   13   14
15      16   17   18   19   20   21
22      23   24   25   26   27   28
```

CHAPTER FIVE

COMMON AND GLOBAL DATA

COMMON AND GLOBAL DATA

PL/B offers three different storage classes for data variables.



COMMON DATA

Common Data Items are data or file definitions that are passed unchanged between programs.

Data definitions to be considered Common Data Items are designated by an asterisk (*) preceding the field size. The first program loaded initializes the defined Common Data Items.

Common Data Items must be of the same type and size and reside in the same position at the start of the User Data Area for each chained program.

Common Data Items must be contiguous and precede all other data or file definitions.

Three Data Definition Labels are reserved as Common Data for every program:

- S$ERROR$ contains any error codes resulting from a PL/B instruction.

- S$RETVAL passes a return value to the operating system with a STOP, DSCNCT, or SHUTDOWN. A value may be placed in it as the program's return value for operating system-level batch processing.

- S$CMDLIN contains any data following the program name entered on the command line is placed.

COMMON DATA - continued

Example:

Assume these variables are to be passed between programs. The first program declares them:

```
DATE       DIM        8
SYSTEM     INIT       "ACCTSPAY"
COUNT      FORM       4
LOOP       INTEGER    1
```

All successive programs would declare them as follows:

```
DATE       DIM        *8
SYSTEM     DIM        *8
COUNT      FORM       *4
LOOP       INTEGER    *1
```

Using this example, the variables would retain the values set by the previous program. They are not re-initialized and blank filled by this program.

GLOBAL DATA

Global Data Items are data and file definitions that are kept intact across CHAIN instructions and between modules.

Data and file definitions to be considered Global Data Items are designated by two methods.

A single percent sign (%) preceding the field size or following the file type places the item in the Global Data Area and initializes it to the definition in the program.

Double percent signs (%%) place the name of the item in the Global Data Area but do not initialize it in any way.

Before use, a variable defined with a double percent must be initialized in another module.

Global Data Items are saved in the Global Data Area based on the label for the item.

All occurrences of that Global Data Item with that label must be of the same type and, unless the %% form is used, must be the same size, for each chained program.

Global Data Items may be anywhere in a program or loaded module.

GLOBAL DATA – continued

Assume these variables are to be placed in the Global Data Area. The first program declares them:

```
DATE      DIM       %8
SYSTEM    INIT      %"ACCTSPAY"
COUNT     FORM      %4
LOOP      INTEGER   %1
```

Successive programs or modules would declare them (in any order) as follows:

```
LOOP      INTEGER   %1
SYSTEM    DIM       %8
COUNT     FORM      %4
DATE      DIM       %8
```

or:

```
SYSTEM    DIM       %%
COUNT     FORM      %%
DATE      DIM       %%
LOOP      INTEGER   %%
```

Using this example, the variables would retain the values set by the previous program. They are not initialized and blank filled by the subsequent programs.

EXERCISE 10

1. Modify Exercise 3 to chain to Exercise 10 upon pressing the F1 key.
2. Pass the account information from Exercise 3 to Exercise 10 via common variables.
3. Exercise 10 should display the variables and allow an operator acknowledgement.
4. Exercise 10 should terminate and return control to Exercise 3.

CHAPTER SIX

CHARACTER STRING INSTRUCTIONS

CHARACTER STRING INSTRUCTIONS

The instructions detailed in this chapter operate on character variables (DIMs and INITs). The following concepts are a necessary part of understanding these instructions.

Character string variables contain three internal values. The values are the:

- Physical Length
- Form Pointer
- Length Pointer

The Physical Length (PL) is set by the compiler or the SMAKE instruction.

The Physical String refers to all of the data within the Physical Length of the variable, even those bytes outside the range of the Logical String.

The Form Pointer (FP) indicates the start of the Logical String.

The Length Pointer (LP) indicates the end of the Logical String.

Logical Length (LL) refers to the number of bytes from the Form Pointer (FP) to the Length Pointer (LP) inclusive. The Logical Length never exceeds Physical Length.

Logical String (LS) refers to the data from the Form Pointer to Length Pointer inclusive and may never be greater than the Physical String.

A Null String is a variable whose Form Pointer is zero.

CHARACTER STRING INSTRUCTIONS – continued

PL/B supports a wide array of character string instruction. These instructions can be divided in several categories

- Attribute Instructions
- Basic string instructions
- Bitwise string instructions
- String transformation instructions
- Searching and scanning instructions

ATTRIBUTE INSTRUCTIONS

PL/B instructions that modify the attributes of a character variable are:

Verb	**Usage**
BUMP	Adds a value to the Form Pointer.
CLEAR	Sets Form Pointer and Length Pointer to zero
ENDSET	Sets the Form Pointer equal to Length Pointer.
LENSET	Sets the Length Pointer equal to Form Pointer.
MOVEFPTR	Moves the Form Pointer value to a numeric variable.
MOVELPTR	Moves the Length Pointer value to a numeric variable.
MOVEPLEN	Moves the Physical Length value to a numeric variable.
RESET	Sets the Form Pointer of a string variable.
SETLPTR	Sets the Length Pointer of a string variable.
SFORMAT	Changes the Physical Length value of a string variable

Example:

```
VAR1      INIT      "TESTDATA"
          DISPLAY   *HD,*N,*LL,VAR1     // Displays TESTDATA
          BUMP      VAR1
          DISPLAY   *HD,*N,*LL,VAR1     // Displays ESTDATA
          RESET     VAR1,5
          DISPLAY   *HD,*N,*LL,VAR1     // Displays DATA
          BUMP      VAR1,-1
          LENSET    VAR1
          RESET     VAR1
          DISPLAY   *HD,*N,*LL,VAR1     // Displays TEST
          CLEAR     VAR1
          DISPLAY   *HD,*N,*LL,VAR1     // Displays nothing
```

BASIC STRING INSTRUCTIONS

PL/B instructions that act upon the logical string of a character variable are:

Verb	**Usage**
APPEND	Append a string to another string.
CHOP	Move a variable to another while removing trailing spaces.
CMATCH	Match a character with another character..
CMOVE	Move a character.
COUNT	Count number of characters in a list of variables.
EXTEND	Append one or more blanks to a string..
MATCH	Match a variable with another variable.
MOVE	Move one variable to another.
SDELETE	Delete a string from within a string.
TYPE	Determine if a string variable has a valid numeric format.

BITWISE STRING INSTRUCTIONS

PL/B instructions that act upon the logical string of a character variable are:

Verb	**Usage**
AND	Bitwise AND two characters together.
NOT	Bitwise NOT a character and move to another variable.
OR	Bitwise OR a character with another character.
SET	Assigns a variable or list of variables to a value of one.
TEST	Bitwise test one character with another character.
XOR	Bitwise exclusive or one character with another character.

EXERCISE 11

In this exercise, we want to create four subroutines. Each subroutine should be called from the main program. All subroutines should perform the requested operation without use of any additional strings or the FINDCHAR instruction. The subroutines should also restore the form pointer to its value upon exit. A typical main program might appear as follows:

```
STRING    INIT      "Sunbelt Computer Software"
REPLY     DIM       1
.
          DISPLAY   *HD,*R,"Before: ",STRING
.
          CALL      ROUTINE   // appropriate subroutine call
.
          DISPLAY   *HD,*R," After: ",STRING
.
          KEYIN     *HD,*R,"Press any key...",REPLY
          STOP
```

The following code allows access the each character in a string:

```
ROUTINE
          MOVEFPTR  STRING,START   // save starting FP
.
          LOOP
          ...                      // your code here
          BUMP      STRING
          REPEAT    UNTIL EOS
.
          RESET     STRING,START   // restore FP
          RETURN
```

EXERCISE 11 – continued

1. Write a routine (count_char) that counts the occurrences of a particular letter in a string.

2. Write a routine (rep_char) that replaces all occurrences of a particular letter in a string with another letter.

3. Write a routine (invert_char) that changes all upper case letters to lower case and upper case to lower case.

   HINT: XORing a letter with 040 will convert upper case to lower case and vice versa.

4. Write a routine (del_char) that deletes all occurrences of a particular letter from a string.

STRING TRANSFORMATION INSTRUCTIONS

PL/B instructions that act upon the logical string of a character variable are:

Verb	**Usage**
COMPRESS	Compress a string.
DECOMPRESS	Decompress a string.
DECRYPT	Decrypt a string.
EDIT	Apply a special display format to a character variable.
ENCRYPT	Encrypt a string.
EXPLODE	Disassemble a delimited string into variables.
FILL	Fill character variables with a value.
IMPLODE	Append data from variables or objects into a single delimited variable.
LOAD	Move one of several variables to another variable.
LOWERCASE	Convert a string to lower case.
PACK	Append several variables to another variable.
PACKKEY	Append several variables to another variable respecting Physical Length.
PARSE	Select next several characters based on a mask.
REMOVE	Move a variable to another variable and adjust the Form Pointer of first variable.
REPLACE	Change characters to other characters.
STORE	Move one of several variables to destination variable.
UNPACK	Split a variable to several (usually smaller) variables.
UPPERCASE	Convert a string to upper case.

STRING SEARCHING AND SCANNING INSTRUCTIONS

PL/B instructions that act upon the logical string of a character variable are:

Verb	Usage
FINDCHAR	Search a string for a character from a list.
SCAN	Search a variable for a character string.
SEARCH	Search a set of variables for starting with a character string.

Examples:

```
STRING   INIT     "Sunbelt Computer Software"
FOUND    DIM      1
POSITION FORM     2
.
         FINDCHAR "ABC",STRING,FOUND
         MOVEFPTR STRING,POSITION     // Found=C,
                                      //  Position=3
         SCAN     "Inc",STRING
         MOVEFPTR STRING,POSITION   // Position=27
```

Search Example

```
STRING1  INIT     "ABC"
STRING2  INIT     "DEF"
STRING3  INIT     "GHI"
STRING4  INIT     "JKL"
MAX      FORM     "4"
INDEX    FORM     2
.
         SEARCH    "GHI" IN STRING1 TO MAX WITH INDEX


INDEX will contain a three (3) after this statement
```

CHAPTER SEVEN

MATHEMATICAL INSTRUCTIONS

MATHEMATICAL INSTRUCTIONS

The instructions detailed in this chapter operate on numeric variables (FORMs and INTEGERs). The following concepts are a necessary part of understanding these instructions.

The arithmetic instructions set the Condition Flags: ZERO (or EQUAL), LESS, and OVER.

These flags are set (TRUE) or cleared (FALSE) depending on the results of the operation. They usually indicate the following:

Flag	**Indication**
ZERO	The result of the instruction was zero.
LESS	The result of the instruction was less than zero.
OVER	Significant digits were lost in the result

When the OVER Condition Flag is set (TRUE), the LESS and ZERO condition indicators are unpredictable.

BASIC MATHEMATICAL INSTRUCTIONS

The basic PL/B instructions that act upon numeric variables are:

Verb	Usage
ADD	Add two numbers with optional giving clause.
COMPARE	Compare two numbers.
DIVIDE	Divide two numbers with optional giving clause.
MULTIPLY	Multiply two numbers with optional giving clause.
SUBTRACT	Subtract two numbers with optional giving clause.
INCR	Increment a variable by one.
DECR	Decrement a variable by one.

Examples:

```
          ADD       VALUE1,VALUE2

          ADD       VALUE1 TO VALUE2 GIVING VALUE3

          SUB       "1",INDEX
          GOTO      START IF ZERO


TOGGLE    FORM      1
.
          SUB       "1",TOGGLE
          IF        ZERO
            (zero logic)
          ELSE
            (non-zero logic)
          ENDIF
```

MORE MATHEMATICAL INSTRUCTIONS

PL/B instructions that act upon the logical string of a character variable are:

Verb	Usage
CALC	Perform algebraic type calculation.
CHECK10	Perform a check digit calculation.
CHECK11	Perform a check digit calculation.
NFORMAT	Format a numeric string.
POWER	Raise a value to a power.
SQRT	Calculate the square root of a value.

CALC

The CALC instruction allows multiple PL/B arithmetic operations (addition, subtraction, multiplication and division) to be combined within a single instruction. It uses one of the following formats:

```
    CALC {dest}=({num}{op}{num}...)
```

CALC analyzes the mathematical equation and generates the appropriate PL/B arithmetic operations to perform the equation(s).

The {num} variables or constants remain unchanged. The result of the operation(s) is placed in {dest}.

Internal CALC precision is dependent upon the type of variables and literals used within the expression. Take care when replacing multiple PL/B math instructions with a single CALC instruction.

The arithmetic operations in an equation are executed such that multiplication (*) and division (/) operations are performed before addition (+) and subtraction (-). Otherwise, the arithmetic operations are performed left to right.

Nesting of operations is allowed using parentheses.

Only the ZERO (or EQUAL) Condition Flag is set after a CALC operation.

Example:

```
    CALC     ANS=((((VAR1 + VAR2) / 10.5) - 2) * 2)
```

TRIGONOMETERIC INSTRUCTIONS

PL/B instructions that perform trigonometric functions are:

Verb	**Calculates the ...**
ARCCOS	arccosine.
ARCSIN	arcsine.
ARCTAN	arctangent.
COS	cosine.
COSH	hyperbolic cosine.
EXP	natural exponent.
LOG	natural log.
LOG10	base 10 log.
SIN	sin.
SINH	hyperbolic sin.
TAN	tangent.
TANH	hyperbolic tangent.

BITWISE INSTRUCTIONS

PL/B instructions that perform bitwise operations on integers are:

Verb	**Performs a ...**
ROTATELEFT	left circular shift an integer.
ROTATERIGHT	right circular shift an integer.
SHIFTLEFT	left logical shift.
SHIFTLEFTA	left algebraic shift.
SHIFTRIGHT	right logical shift.
SHIFTRIGHTA	right algebraic shift.

EXERCISE 12

In this exercise, we will create a simple calculator.

1. The program should prompt the user and accept input for the:

    - first value
    - operation (+, -, *, /)
    - second value

2. The program should calculate and display the result of the operation.

3. Allow the program to repeat the process as long as the user wishes.

4. Pressing the Escape key at any point should terminate the program

5. The program should use the answer as the default for the first value when repeating. The second value should be zeroed out.

CHAPTER EIGHT

SYSTEM INTERFACE INSTRUCTIONS

SYSTEM INTERFACE INSTRUCTIONS

The instructions detailed in this chapter provide an interface to the operating system in use. The instructions provide system information retrieval, program environment manipulation, and task control.

CLOCK

The most commonly used verb for information retrieval is CLOCK. Originally designed to return the current date and time, its capabilities have been enhanced to provide the following information:

Operation	Returns the ...
CPUTIME	accumulated CPU time for the process in microseconds.
DATE	current system date (mm-dd-yy).
DAY	current system day (ddd).
ENV	environment table variables
ERROR	last error code that occurred.
INI	value associated with the keyword specified in {dest} from the current ini file.
PORT	current system port ID and screen characteristics.
SECONDS	number of seconds since 12:00am 01/01/70.
SYSDATE	system day, date, and time
SYSPORT	unique system port ID on certain multi-user systems.
TIME	current system time (hh:mm:ss.xxxxxx).
TIMESTAMP	current system date and time (yyyymmddhhmmsssss).
VERSION	current name and version of the library/run-time/interpreter
WEEKDAY	current day of week number (Sunday = 1,...,Saturday = 7).
WINDOW	current subwindow and cursor coordinates,
YEAR	current system year (yy).

CLOCK ENV

The ENV keyword retrieves a specific environment setting or the next setting within the environment table depending upon the contents of the receiving variable as follows:

If the variable is null, the first entry in the environment table is retrieved.

If the string variable contains an environment table keyword (such as COMSPEC, PATH, etc.), the data for that specific entry is retrieved. The OVER flag is set if no match is found and cleared if a match is found. The EOS flag is set if the environment information is truncated to fit the variable.

If the variable contains an environment table keyword (as above), followed by an equal (=) sign, CLOCK retrieves the next entry in the environment table. The OVER flag is set if the end of the environment table is found, and cleared if the end of the environment table is not found.

Example 1: Retrieve the COMSPEC environment variable value.

```
DATA      DIM       100
.
          MOVE      "COMSPEC",DATA
          CLOCK     ENV,DATA
```

Example 2: Display the entire environment table.

```
ENTRY     DIM       50
.
          LOOP
          CLOCK     ENV,ENTRY
          UNTIL     OVER
          DISPLAY   "ENTRY: ",ENTRY
          REPEAT
          STOP
```

CLOCK INI

The INI keyword retrieves a specific value associated with a keyword within the current program information (ini) file.

If the variable is null, the OVER flag is set.

If the string variable contains a keyword defined in the file, the value assigned to that keyword is retrieved.

The OVER flag is set if no matching keyword is found, and cleared if a match is found.

The EOS flag is set if the value is truncated to fit the variable.

By default, the keyword is retrieved from the "[environment]" section of the file. To access keywords from other sections of the file, precede the keyword with the section name (no brackets) and a semicolon (;).

The runtime searches as many as three .INI files in the following locations:

1. the current directory (A)
2. the Windows directory (B)
3. the directory where the runtime is located (C).

Example:

```
ENTRY     DIM       50
          MOVE      "PLB_TERM",ENTRY
          CLOCK     INI,ENTRY
          DISPLAY   *N,"Terminal Type is: ",ENTRY;
          STOP
```

CLOCK VERSION

The VERSION information returned by the CLOCK verb is in a thirty-one (31) byte variable as follows:

Bytes	Definition
1-5	Library/run-time/interpreter version and revision
6	Blank
7-14	Library/run-time/interpreter name
15	Blank
16-17	System ID as follows: 1=MS-DOS 2=MultiLink 3=NetWare 4=Nestar 5=NTNX 6=Netbios (PC-NET) 7=PC-MOS 8=STARLAN 9=Reserved 10=Linux 11=PowerLan 12=MultiWare 13=DOS-Windows 14=NetWare Lite 15=DOS-OS/2 16=DR Multiuser DOS 17=DESQview 18=PLBWIN/PLBCON
18	Blank
19-26	Client engine name (PLBSERVE/WEBSERVER Only) cliwin PLBCLIENT clicon PLCCLICON Console cliwcepk PLBCLIENT Pocket PC cliwcept PLBCLIENT Palm Top webbrw Web Browser Client
27-31	Client version number

EXERCISE 13

In this exercise, we will use the clock instruction to return information about our operating system, environment settings, and INI data.

1. Begin this exercise by retrieving and displaying the standard date and time information available.

2. Enhance the program to display the VERSION information.

3. Add code to output the entire ENV settings.

4. Finally, use the INI keyword to retrieve the PLB_PATH and PLB_SYSTEM keyword values from the program information file.

PATH

The PATH instruction provides a means of retrieving or modifying the current directory path and creating or deleting another directory path.

```
    PATH    {mode}{sep}{path}
```

The supported modes are:

Mode	**Function**
CURRENT	Retrieve the current directory path
CHANGE	Change to a different existing directory path
CREATE	Create a new, non-existent directory path
DELETE	Delete an existing directory path
EXIST	Check for existence of specified directory path

If CURRENT is specified, the current directory path is transferred to {path}.

If {path} is too small to hold all the directory path information, only those bytes that fit are transferred and the EOS Condition Flag is set (TRUE).

If {path} contains a valid drive designator (C:, D:, etc.), the path information for that drive is retrieved. If null, the directory path for your current drive is retrieved.

Under Windows, any drive but the current drive is always in the root directory. Windows only keeps track of one current path and that is the current path on the current drive.

If CHANGE, CREATE, DELETE, or EXIST is specified, the Logical String of {path} is assumed to contain the directory path used for the operation.

If a CURRENT, CHANGE, CREATE, DELETE, or EXIST operation fails, the OVER Condition Flag is set (TRUE).

SEARCHPATH

The SEARCHPATH instruction defines the set of file paths used during OPEN, PREP, and CHAIN.

```
SEARCHPATH   {mode},{path}
```

The supported operations are:

Operation	**Function**
ADD	Add a directory to the search path
GET	Retrieve the search path
REMOVE	Delete a directory from the search path
SET	Initialize the search path

The default set of paths is initialized to the PLB_PATH environment variable found string in the User Environment Table or in the PLBWIN.INI file.

The SET operation defines a complete set of directories searched. Any information that was previously defined is lost.

For SET, each directory must be separated by a semi-colon character (;). A semi-colon is automatically placed as the first character.

If {path} is null in a SET operation, the directory SET is cleared. Care must be used as the original information retrieved from the PLB_PATH environment variable may be erased.

EXERCISE 14

In this exercise, we will use the PATH and SEARCHPATH instructions to manipulate the system settings

1. Begin this exercise by retrieving and displaying the current path. Save this value.

2. Test for the existence of the "demo" folder of your installation of Sunbelt Software.

3. Change the path to the "demo" folder of your installation of Sunbelt Software.

4. Prove the change succeeded by displaying the current path again.

5. Restore the original path and display it to insure success.

6. Retrieve and display the current search path. Save this value.

7. Add the "Windows" folder to the search path and display the current path to insure success.

8. Restore the original search path and display it to insure success.

TASK CONTROL

PL/B offers a number of instructions that allow programmatic execution of other programs. These programs may be written in PL/B or any other language. The task control instructions are:

- BATCH
- EXECUTE
- FORK
- ROLLOUT
- SHUTDOWN

BATCH

The BATCH instruction initiates the specified command line as a background task and immediately resumes processing without waiting for the result.

```
     BATCH    {command}[,[{in}][,{out}]]
```

Only the Logical String of any character variable parameters is used.

If {command} contains input or output redirection, it supersedes the corresponding {in} and/or {out} specification(s).

If {in} is omitted or null, standard input (stdin) is closed and {command} must be capable of executing without input, unless {command} explicitly redirects the input.

{out} is appended to, not overwritten. If omitted or null, standard output (stdout) is unchanged and appears in the stdout of the initiating program unless changed in {command}. If "/dev/null" (Linux) or "nul" (WINDOWS) is desired, it must be provided explicitly.

If {command} is null or insufficient system resources exist to initiate BATCH, the ZERO flag is set.

Example:

```
CMDLINE   INIT      "copy file1.txt file2.txt"
.
          BATCH     CMDLINE
          IF        ZERO
          DISPLAY   "ERROR: Batch failed"
          ENDIF
```

FORK

The FORK instruction provides a method of executing two PL/B instruction streams simultaneously. FORK provides the ability to perform non-interactive PL/B functions concurrently with interactive PL/B instructions. This is especially useful for off-line data processing, such as batch processing or report generation. The initiating program is considered the parent, the background task being the child process.

```
     FORK     [{in}[,{out}]
```

If insufficient system resources exist to initiate the child process, the ZERO flag is set to TRUE.

The OVER flag is set to TRUE in the child process and FALSE in the parent. The OVER flag may then select different code paths for the parent and child to execute.

If {out} is omitted, standard output (stdout) is unchanged and the child process' stdout appears in the parent's. If "/dev/null" for Linux or "nul" for MSDOS is desired, it must be provided explicitly.

If {in} is omitted, standard input (stdin) is closed and the child process must be capable of executing without input. Any KEYINs executed in the child process are treated as null entries.

When a child process begins, its copies of the parent's files are not open. The child process must open any files needed before attempting to use them.

The child process must terminate with a SHUTDOWN.

The child process initiated is a duplicate of the current program, and execution begins at the current position within the new program.

FORK – continued

Example:

```
DEVNULL   INIT      "/dev/null"
.
MAIN ...........
          CALL      FORKSUB   ;Record retrieval routine
          GOTO      MAIN
.
FORKSUB   FORK      DEVNULL,DEVNULL
          GOTO      FORKERR IF ZERO     ;ZERO = failed
          RETURN    IF NOT OVER         ;NOT OVER = Parent
.
CHILD     SPLOPEN   "PRTFILE","Q"  ;OPEN the spool file
          PRINT     .............  ;Actual PRINT routine
          SPLCLOSE
          SHUTDOWN                 ;terminate CHILD
```

ROLLOUT

The ROLLOUT instruction suspends program execution allowing one or more system functions may be performed. Program execution is reinstated at the next instruction using the appropriate rollback method for the runtime being used. The instruction uses the following format:

```
     ROLLOUT  {command}[,{name}]
```

The optional {name} parameter is provided for language compatibility only and is ignored by the runtime.

The current program status, including all variable pointers, file pointers and Flags, is written into a file for subsequent restoration upon the return from the ROLLOUT.

If {command} is provided, and it is not a Null String, it is placed in the key ahead buffer for execution once the operating system takes control.

If the command fails to execute, ROLLOUT automatically attempts to execute the command using the current COMSPEC keyword if available.

Under PLBWIN, ROLLOUT functions as an EXECUTE and no rollback is necessary. When the task is finished, the current program begins executing again at the instruction following the ROLLOUT.

To return from a ROLLOUT under Linux OR PLBCON, PLB must be executed using the `-r' option either manually or as the last command in the shell script file.

EXECUTE

The EXECUTE instruction suspends program execution allowing another system function or program to be performed. Upon completion of the specified task, program execution resumes with the instruction following the EXECUTE statement.

```
     EXECUTE  {command}[,{os}[,{keyword}]]
```

The required {command} is a previously defined Character String Variable, a string Literal, or ARRAY element containing the command line to be executed.

The optional {os} parameter is provided for language compatibility only and is ignored by the runtime. A warning message will be displayed if the parameter is specified.

The optional {keyword} parameter must be one of the following values:

Keyword	**The operation of the verb …**
FOREGROUND	does not change.
BACKGROUND	is the same as the BATCH verb. Note that BATCH is not supported for all OS environments.

The currently executing program is suspended with all variables, pointers, file conditions/pointers and Flags intact. The specified command line is executed and upon its completion, the program resumes at the next instruction following the EXECUTE.

The OVER flag is set if the program is unable to perform the EXECUTE.

SHUTDOWN

The SHUTDOWN instruction provides a method for terminating program execution and returning control to the operating system. Optionally, it may pass a command line to the operating system (such as a utility name, batch file, or other command) for execution.

```
     SHUTDOWN     [{command}]
```

The program currently executing is terminated and control is returned to the operating system.

If {command} is provided and it is not a Null String, it is placed in the key ahead buffer for execution once the operating system takes control. The command line's execution status is only limited by the operating system's criteria for executable command lines.

If the command fails to execute, SHUTDOWN automatically attempts to execute the command using the current COMSPEC keyword if available.

When a SHUTDOWN is performed in a GUI environment with a command to execute, a new process is created for the command. If a '!' character starts the command line, the process is started in an iconized state.

EXERCISE 15

Write a program that performs the following:

- Shutdown and execute a "dir /p" command line.
- Shutdown and execute the same command in an iconized state.
- Shutdown and execute a null command line.
- Use EXECUTE to start a task running lab exercise 1.

CHAPTER NINE

INTERACTIVE IO INSTRUCTIONS

INTERACTIVE IO INSTRUCTIONS

The instructions detailed in this chapter accept information from the keyboard and to display information on the user's screen. Additional instructions provide for the saving and restoring of screen states and attributes.

When an interactive input or output instruction terminates, the cursor is normally repositioned with a line feed and a carriage return (column 1 of the next line). If the cursor is located on the bottom line of the screen and the instruction was not terminated with a semi-colon, the screen may roll up one line unless the *NOROLL control is in effect.

If an input or output instruction is terminated with a semi-colon (;), the line feed and carriage return are suppressed and the cursor remains after the last byte output.

Formatting of data items is influenced by both the data type and any list controls.

All Function Flags are cleared (FALSE) upon entering a KEYIN instruction. If a Function Key is struck, or a mouse button pressed, the appropriate Function Flag is set (TRUE) and the remainder of the KEYIN bypassed.

Under most circumstances, the Condition Flags remain unchanged.

BEEP

The BEEP instruction sends an ASCII "ring bell" character to the terminal to sound an audible beep, similar to the *B list control.

In Windows, an appropriate will be played.

```
     BEEP
```

DISPLAY

The DISPLAY instruction displays information on the user's screen.

```
     DISPLAY  {data}[;]
```

Items within {data} are separated with commas. The list items may be continued on additional lines by ending each continued line with a colon (:).

All string literals are displayed on the screen exactly as provided.

Binary, Decimal, Octal and Hex characters in literals are sent to the terminal as specified.

The contents of character string variables and numeric variables are displayed on the user's screen whenever they appear in the list of a DISPLAY instruction.

Example:

```
CVAR     INIT     "Sunbelt Computer Software"
NVAR     FORM     "8.2"
.
         DISPLAY  *HD,CVAR," - Version ",NVAR-
```

DISPLAY – continued

Unless modified by a list control, these variable types are displayed as follows:

Character String Variables

- The display begins with the first physical character and continues through the Length Pointer.
- Blanks are displayed for each character between the Length Pointer and the Physical Length.
- If the Form Pointer is zero, blanks are displayed for each character position to the Physical Length.
- If an item within the list is a character string ARRAY and no specific array element has been designated, every array element is displayed.

Numeric Variables

- The display begins with the first physical character and continues through the Physical Length of the variable.
- If an item within the list is a numeric ARRAY and no specific array element has been designated, every array element is displayed.
- INTEGER variables of length one, two, three and four are converted and displayed as FORMs of length three, five, eight, and ten respectively.

KEYIN

The KEYIN instruction accepts input from the keyboard and may also display data on the user's screen prior to accepting the input.

```
        KEYIN     {data}[;]
```

Individual items within {data} are separated with commas. The list items may be continued on additional lines by ending each continued line with a colon (:).

A line feed, Function/Special key, system interrupt, or trapped key terminates a KEYIN instruction.

If the *RV list control is used, several condition flags are set based on conditions encountered during the KEYIN:

- The EOS flag is set if a null entry was made with *RV in effect.
- The LESS flag is set if a timeout occurred with *T in effect.
- The OVER flag is set if the KEYIN was terminated with a Function/Special key, System Interrupt or Trapped Key

All string literals are displayed on the screen exactly as provided.

KEYIN – continued

Variables within the KEYIN list retain information entered from the keyboard. Each character entered is echoed to the screen, unless specified to the contrary. The cursor is bumped one position to the right for each character entered, up to the maximum field size. A null entry indicates the enter key ([E]) was pressed without entering any data. Unless modified by a list control, variables are accepted as follows:

Character String Variables

- Any key code from the keyboard that is not already defined as a function key, cursor key or other special key may be entered into character string variable.
- The number of characters entered cannot exceed the Physical Length.
- If a Null String is entered, both the Form Pointer and Length Pointer is set to zero.
- If data is entered, the Form Pointer is set to one (1) and the Length Pointer points to the last character entered.
- If the enter key is pressed, processing continues with the next item in the list.

Using a DIM 10 variable, the results of various KEYIN instructions are as follows:

```
------ INPUT -----   -- RESULTS -------------
                     PL   FP   LP   Contents
[E] (null entry)     10   0    0    ••••••••••
ABCDE[E]             10   1    5    ABCDE•••••
ABCDEFGH[E]          10   1    8    ABCDEFGH••
```

KEYIN - continued

Numeric Variables

Only valid numeric data is accepted for entry into a numeric variable. Upon pressing the enter key, the entered data is formatted to match the variable's data definition.

- A minus sign is accepted if it is the first character entered and there remains room for at least one digit before the decimal point.
- A decimal point is accepted only if one is specified in the data definition.
- Only one decimal point is accepted.
- The number of digits that may be entered both preceding and following any decimal point is restricted to the specified data definition.
- A null entry generates a value of zero.
- Once the enter key is pressed, processing continues with the next item in the list.

Using a FORM 3.1 variable, the results of various KEYIN instructions are as follows:

```
---- INPUT ----      --- RESULTS ---
PL   Contents
[E] (null entry)     5    •••.0
0 [E]                5    •••.0
5 [E]                5    ••5.0
25.2 [E]             5    •25.2
-5.8 [E]             5    •-5.8
```

KEYIN/DISPLAY LIST CONTROLS

*+	*COLOROFF	*GROWT	*OP	*SETSWTB
*-	*CON	*GROWTB	*OPNLIN	*SHRINKB
*ALLOFF	*DELCHR	*H	*P	*SHRINKL
*B	*DELLIN	*HA	*PL	*SHRINKLR
*BGCOLOR	*DSPMODE	*HD	*POFF	*SHRINKR
*BLANKOFF	*DSPNW	*HOFF	*PON	*SHRINKSW
*BLANKON	*DSPW	*HON	*R	*SHRINKT
*BLINKOFF	*EF	*HU	*RD	*SHRINKTB
*BLINKON	*EL	*IN	*RESETSW	*SL
*BOLDOFF	*EP	*INSCHR	*RESTSCR	*UC
*BORDER	*ERASESW	*INSLIN	*RESTSW	*ULOFF
*C	*ES	*IT	*ROLL	*ULON
*CL	*FGCOLOR	*JC	*RPTCHAR	*V
*CLICK	*GOFF	*L	*RPTDOWN	*VA
*CLICKOFF	*GON	*LC	*S0-*S9	*W
*CLICKON	*GROWB	*LL	*SAVESW	*ZF
*CLSLIN	*GROWL	*MONO	*SCRLEFT	*ZFOFF
*COFF	*GROWLR	*N	*SCRRIGHT	*ZFON
*{color}	*GROWR	*NOROLL	*SETSWALL	
*COLOR	*GROWSW	*NP	*SETSWLR	

KEYIN ONLY LIST CONTROLS

List controls that are only available when using the KEYIN instruction are:

*ABSOFF	*ESON
*ABSON	*INSERT
*CRTOFF	*INSOFF
*CRTON	*INSON
*DE	*JL
*DPI	*JR
*DPR	*KCOFF
*DV	*KCON
*DVEDIT	*KEYMODE
*DVRV	*MAXCHAR
*EDIT	*MINCHAR
*EOFF	*RV
*EON	*T
*ESCHAR	*WRAPOFF
*ESOFF	*WRAPON

GRAPHIC LIST CONTROLS

The following list controls generate special graphic characters.

The list controls are also valid as arguments to *RPTCHAR, *RPTDOWN, *SCRRIGHT, and *SCRLEFT.

Control	Output
*HLN	Horizontal single line (ANSI)
*HLNDD	Horizontal double line
*VLN	Vertical single line (ANSI)
*VLNDD	Vertical double line
*RTA	Right arrow (ANSI)
*LFA	(ANSI)
*DNA	(ANSI)
*UPA	Up arrow (ANSI)
*ULC	Upper left corner single line (ANSI)
*DTK	Down tick single line (ANSI)
*URC	Upper right corner single line (ANSI)
*RTK	Right tick single line (ANSI)
*CRS	Cross single line (ANSI)
*LTK	Left tick single line (ANSI)
*LLC	Lower left corner single line (ANSI)
*UTK	Up tick single line (ANSI)
*LRC	Lower right corner single line (ANSI)

GRAPHIC LIST CONTROLS – continued

*ULCDD	Upper left corner double line
*DTKDD	Down tick double line
*URCDD	Upper right corner double line
*RTKDD	Right tick double line
*CRSDD	Cross double line
*LTKDD	Left tick double line
*LLCDD	Lower left corner double line
*UTKDD	Up tick double line
*LRCDD	Lower right corner double line
*ULCSD	Upper left corner single horizontal/double vertical line
*DTKSD	Down tick single horizontal/double vertical line
*URCSD	Upper right corner single horizontal/double vertical line
*RTKSD	Right tick single horizontal/double vertical line
*CRSSD	Cross single horizontal/double vertical line
*LTKSD	Left tick single horizontal/double vertical line
*LLCSD	Lower left corner single horizontal/double vertical line
*UTKSD	Up tick single horizontal/double vertical line
*LRCSD	Lower right corner single horizontal/double vertical line
*ULCDS	Upper left corner double horizontal/single vertical line
*DTKDS	Down tick double horizontal/single vertical line
*URCDS	Upper right corner double horizontal/single vertical line
*RTKDS	Right tick double horizontal/single vertical line
*CRSDS	Cross double horizontal/single vertical line
*LTKDS	Left tick double horizontal/single vertical line
*LLCDS	Lower left corner double horizontal/single vertical line
*UTKDS	Up tick double horizontal/single vertical line
*LRCDS	Lower right corner double horizontal/single vertical line

EXERCISE 16

Write a data entry program that

1. The file should contain the following fields:

Type	Size	Description
N	7	Number
AN	30	Name
AN	30	Address
AN	20	City
AN	2	State
N	5	Zip code
N	7.2	Balance

2. Prompt the user for a value for each field and ensure that it is supplied.

3. Allow program termination using the Escape key at any point.

4. Pressing the up key during field entry should allow backing up to the previous field in a circular fashion.

5. Draw a graphic box around the input section.

6. Once the last field has been input, display the values in a separate area of the screen and then repeat the input process.

CHAPTER TEN

DISK IO INSTRUCTIONS

DISK IO

Through disk input/output instructions, information or data is stored and retrieved as logical records within files on the disk.

To understand how disk IO is done in PL/B, some basic concepts must first be discussed:

- File Structures
- File Access Methods
- File Access Modes
- File Locking
- Character Variables
- Numeric Variables
- Partial IO
- List Controls

FILE STRUCTURES

Information regarding each file is kept in a file definition variable.

```
DATAFILE  FILE
ISAMFILE  IFILE
AAMFILE   AFILE
```

Three types of files are available in PL/B:

Type	Description
FILE	One physical text file.
IFILE	One physical text file and an ISAM key file. The ISAM file contains the key information and associated logical record pointer for each record in the physical text file.
AFILE	One physical text and an AAM key file. The AAM file contains the hashed key information and associated logical record pointer for each record in the physical.

FILE ACCESS METHODS

PL/B supports four file access methods:

- Sequential Access Method
- Random Access Method
- Indexed Sequential Access Method
- Associative Access Method

SEQUENTIAL FILE ACCESS METHOD

Sequential access is the simplest method of processing records within a data file. Records within the data file are directly processed sequentially forward or backward.

Information is processed one logical record at a time, in sequential order.

Each logical record is terminated with an End Of Record (EOR) mark. The next record starts immediately after the EOR mark for the preceding record.

The EOR is automatically written after each WRITE operation unless partial I/O is being performed.

Sequential records may be fixed or variable in length.

Records written using the sequential method are not space compressed unless the *+ list control has been specified.

The space compression technique utilized is the standard technique incorporated within the operating system so that the files are fully accessible by any program or utility compatible with the operating system in use.

Sequential file processing using the -1 and -2 methods indicates standard sequential processing of the file in forward order.

Sequential files may also be processed using the -3 method (position to the End Of File prior to processing the record) for both READs and WRITES.

In addition, there is a -4 method for reading a file sequentially backwards from the current file pointer position. The -4 method is only supported on sequential READ instructions (not WRITE).

RANDOM FILE ACCESS METHOD

The random or direct access method allows the user to process logical records using sector positioning and tabbing within each sector.

By allocating a single record per sector, it is possible to position directly to any record in the file rather than having to process sequentially through the file until the desired record is reached.

Since sector numbering begins with zero (0) and increments accordingly, the first record in the file (record 1) is located at sector zero.

If only one record is placed per sector, sector zero would contain record 1, sector 1 would contain record 2, sector 2 would contain record 3 and so on.

Random records may be fixed or variable in length. However, since the positioning point of reference is based upon a fixed length sector rather than End Of Record (EOR) terminators, caution should be exercised when writing variable length records since they may end up being less than a sector or spanning more than one sector.

Records written using the random method are not space compressed unless the *+ list control has been specified.

INDEXED SEQUENTIAL ACCESS METHOD

Indexed sequential access method (ISAM) files use a key file (ISI file) to determine which logical record in the physical text file is associated with the key data supplied.

The ISI file acts as the control file for the processing of logical records within the physical text file.

The key data is usually built from actual data within the logical record to be processed.

An ISAM file may permit or disallow duplicate keys within the ISI control file.

- If duplicate ISAM keys are not to be allowed, each logical record in the physical text file must have a unique ISAM key pointing to it.

- If duplicate ISAM keys are allowed, caution must be taken to insure the correct record is being processed, since the same key information may be duplicated and reference a different logical record in the physical text file.

A physical text file that has an associated ISI key file may be read physically sequential or random, key sequentially (forward or backwards) or through a specific key. However, logical records may only be written, updated or deleted in the physical text file through use of a specific key.

An attempt to retrieve a logical record using a key that has not been previously written to the ISI file fails and positions the ISI file pointer to where the key information would have been placed had it been written. The OVER Condition Flag is set to TRUE.

INDEXED SEQUENTIAL ACCESS METHOD - continued

If fixed length records are used, the sector size must be equal to or greater than the declared fixed record length.

Any records written that are other than the declared fixed record length size are padded or truncated as appropriate.

Deleted record space is automatically re-used on ISAM files with fixed record lengths.

A physical text file may have more than one key file (ISAM and/or AAM) associated with it. Caution should be exercised when processing physical text files with more than one key file associated with it to insure adjustments are made to each affected key file when any processing is performed.

ASSOCIATIVE ACCESS METHOD

Associative access method (AAM) files allow the user to process logical records based on collective key information.

Similar to ISAM, this method uses a control file (AAM file) to determine which logical record is to be processed based upon the key information specified.

Unlike ISAM, AAM does not require a complete or specific key. However, the key information must have been extracted from data written within the logical record.

A physical text file that has an associated AAM key file may be read physically sequential or random, key sequentially (forward or backwards) or through a specific key. However, logical records may only be written, updated or deleted in the physical text file through the AAM key file.

An attempt to retrieve a logical record using a key that has not been previously written to the AAM file fails with the OVER Condition Flag set to TRUE.

If fixed length records are used, the sector size must be equal to or greater than the declared fixed record length. Any records written that are other than the declared fixed record length size is padded or truncated as appropriate.

In addition, deleted record space is automatically re-used on AAM files with fixed record lengths.

A physical text file may have more than one key file (AAM and/or ISAM) associated with it.

Caution should be exercised when processing physical text files with more than one key file associated with it to insure adjustments are made to each affected key file when any processing is performed.

ASSOCIATIVE ACCESS METHOD - continued

Up to 95 different key fields may be specified during an AAM file's creation.

A search for matching logical records may be made using the following techniques:

Value	Match
X	Exact, key information must exactly match the specified key field (like ISAM).
F	Free, key information may be contained anywhere within the specified key field.
L	Left, key information must be in the left most bytes of the specified key field.
R	Right, key information must be in the right most bytes of the specified key field.

At least one non-null key specification is necessary for the first READ operation since an AAM file has been opened. After this criteria has been met, subsequent READ operations using all null key information rereads the previously retrieved logical record.

Upon successful retrieval of a record, subsequent records that also match the search criteria may be accessed using the READKG instruction. After at least one successful READKG instruction, the search may be resumed in reverse order using the READKGP instruction.

Logical records may be updated in place (key information is automatically updated).

FILE ACCESS MODES

Any file (FILE, IFILE, or AFILE) may be opened or prepared in one of four access modes; SHARE, SHARENF, EXCLUSIVE, or READ.

CREATE and PREPARE keywords are also supported and are treated as SHARE.

Files may also be accessed in modes different than they were originally created or may be accessed in more than one mode at a time within the rules for each mode described in these sections:

- EXCLUSIVE File Access
- READ (Only) File Access
- SHARE File Access
- SHARENF File Access

EXCLUSIVE FILE ACCESS

The EXCLUSIVE access mode designates limited accessibility to the file.

An attempt to OPEN a file in EXCLUSIVE mode that is currently being used in SHARE or SHARENF mode or is write-protected generates a trappable I/O error.

A system open on the file is only performed upon execution of an OPEN or PREPARE instruction. The file handle remains open until the file is closed or the program terminates.

The data is not automatically flushed to disk after each instruction nor is the system directory being continuously updated. Data remains in memory until the internal buffer is full or a CLOSE, OPEN, or PREP instruction is executed for this same logical file name.

Data files opened in EXCLUSIVE mode are not protected against premature termination of the program since the system directory is not being continuously updated.

A file opened in EXCLUSIVE mode is locked and unavailable for SHARE, SHARENF, or EXCLUSIVE mode access by other PL/B programs. Files opened in EXCLUSIVE mode may only be accessed by other PL/B programs that use READ mode for the file.

Throughput from files opened in EXCLUSIVE mode is faster than those opened in SHARE or SHARENF mode.

EXCLUSIVE mode is solely recommended for inquiry only files or files that just one user is allowed to update.

A single program may open a file multiple times in EXCLUSIVE mode without giving an error.

READ FILE ACCESS

The READ access mode designates the file is used for inquiry (read only) purposes.

READ mode accesses files that were opened in either EXCLUSIVE, SHARE, or SHARENF mode by another process or are write-protected.

An attempt to perform an output instruction to a file opened in READ mode by the program generates an I/O error.

A system open on the file is only performed upon execution of an OPEN or PREPARE instruction. The file handle remains open until the file is closed or the program terminates.

Throughput from files opened in READ mode is faster than those opened in SHARE or SHARENF mode.

READ mode is useful for inquiry only purposes.

SHARE FILE ACCESS

The SHARE access mode is the default mode for all file PREPARE/OPEN instructions and designates that the file is shareable.

An attempt to OPEN a file already opened in EXCLUSIVE mode by another program or a file that is write-protected generates a trappable I/O error.

SHARE mode files offer the maximum data integrity of the four access modes.

Any file opened in SHARE mode has its contents and directory entry continuously updated for each transaction that takes place.

Files accessed in SHARE mode are protected from premature termination of the program due to errors or system failure.

Due to the efforts required to insure the aforementioned, overall throughput using SHARE mode is somewhat less than SHARENF, EXCLUSIVE, or READ mode files.

It remains the responsibility of the programmer to design the application for correct multi-user access (through usage of FILEPI etc.)

SHARE or SHARENF mode must be used if a physical text file is updated by more than one user simultaneously or through more than one key file within the program.

SHARENF FILE ACCESS

The SHARENF access mode is optional for all file PREPARE/OPEN instructions and designates that the file is shareable.

An attempt to OPEN a file already opened in EXCLUSIVE mode by another program or a file that is write-protected generates a trappable I/O error.

Any file opened in SHARENF mode does NOT automatically flush the new end of file position to the directory for each WRITE to the file.

A FLUSH, FLUSHEOF, or CLOSE instruction must be executed in order to get the correct file size updated in the directory.

SHARENF mode is faster than SHARE mode when the data files are being extended.

Since the directory contents are NOT continuously updated, premature system failure could cause loss of data when SHARENF is used.

It remains the responsibility of the programmer to design the application for correct multi-user access (through usage of FILEPI, etc.)

SHARE or SHARENF mode must be used if a physical text file is updated by more than one user simultaneously or through more than one key file within the program.

FILE AND RECORD LOCKING

When a file is being shared by multiple programs, some means must be employed prevent the programs from interfering with each other.

PL/B provides the ability to lock files and records so they cannot be accessed by other programs. These methods are called file locking and record locking.

File and record locking may not be active simultaneously.

FILE LOCKING

File locking is a method that grants one program exclusive access to entire files and their indexes for a short period.

File locking is implemented with a FILEPI statement in PL/B.

```
     FILEPI   {count};{file1][,{file2}]…
```

When a file or group of files is locked with a FILEPI statement, the user is granted exclusive access to that group of files for some number of PL/B statements.

The number of statements for which the files are to be locked is specified in the FILEPI statement.

Once the specified number of statements is executed, the files become unlocked automatically.

Not all instructions apply against the count in the FILEPI statement.

The lock may be programmatically canceled by specifying zero as the operand in the FILEPI statement.

When files are locked due to a FILEPI statement, another FILEPI statement may not be executed. This means that the program must lock all of the files that will be locked for a particular operation at once.

RECORD LOCKING

Record locking is a method that allows individual records in a file to be locked. This allows multiple users to access a file simultaneously while allowing the individual users to protect the data they are working on from other users.

Because users compete for records and not whole files, systems that use record locking will usually have better performance than those that use file locking.

Record locking is not limited to short periods. A program may lock a record and keep it locked as long as necessary.

Three types of record locking are supported.

RECORD LOCKING MODES

Two record locking modes are available with PL/B:

1. Manual
2. Automatic

The desired mode is specified when a file is opened.

If a file is opened specifying the manual locking mode, special forms of the read-class instructions may be employed read a record and lock it.

If a file is opened specifying the automatic locking mode, any time a record is read, it is locked.

RECORD UNLOCKING MODES

Two unlocking modes are available with PL/B:

1. Multiple
2. Single

The desired mode is specified when a file is opened.

If a file is opened specifying the multiple unlocking mode, all locked records remain locked until specifically unlocked.

An UNLOCK instruction is provided to unlock all locked records in a file.

Once multiple records are locked, it is not possible to unlock only one record in a file.

With the single unlocking mode, only one record in a file may be locked at a time. When an operation is performed on a file and it has a locked record, that record is unlocked according to the following:

1. A read instruction unlocks any previously locked record before the read is attempted.
2. An update or delete operation unlocks the record after the operation is complete.
3. A write to a record that is not locked unlocks the currently locked record before the write starts.
4. A write to the locked record unlocks the record when the operation completes.

RECORD LOCK WAITING OPTIONS

If a record is locked by another user and an attempt is made to read or lock it, the time to wait for the record to become unlocked may be specified.

The available options are:

1. Wait forever for the record to be unlocked. Mutual exclusion may be a particular problem when the operation to wait forever is selected. In these cases, care should be taken to prevent mutual exclusion.

2. Wait for some period for the record to be unlocked. A period may be specified in the OPEN instruction. If the record is locked when the operation is attempted and stays locked for the period, the LESS flag is set after the operation.

3. Return immediately. The LESS flag is set after the operation.

CHARACTER VARIABLES

All Character String Variables are read or written as follows. This may be modified with the list controls.

Input

The information is read from the disk starting at the file's current position.

Data is moved into the variable beginning with the first physical position of the variable and continuing through the Physical Length.

The Form Pointer is set to one and the Length Pointer is set to the last character transferred unless the *LL control is used.

If the *LL control is used with a READ instruction, the Form Pointer is set to one (1) and the Length Pointer is set to the last non-blank character transferred.

If the End Of Record (EOR) is reached before the variable is full, the Length Pointer indicates to the last character transferred and the remainder of the variable is blank filled.

If the EOR is reached before the variable list is exhausted, the remaining character string variables are blank filled and set to null.

Output

The contents from the first physical character through the Length Pointer are transferred to disk with any positions following the Length Pointer through the Physical Length being blank filled, unless the *LL control is used.

If *LL is used, only the contents between the Form Pointer and the Length Pointer are written, without any blank filling.

The first character is written at the current position within the file.

NUMERIC VARIABLES

All Numeric Variables are read or written as follows. This may be modified with the list controls.

Input

The information is read from the disk starting at the file's current position.

Data is moved into the variable beginning with the first physical position of the variable and continuing through the Physical Length.

Other than valid numeric data, only leading spaces are converted to zeros. Any other deviation results in a FORMAT error (F01).

A blank field (all spaces) will result in a variable value of zero.

If the End Of Record (EOR) is reached before the variable is full, the remainder of the variable are filled with zeros.

If the EOR is reached before the list is exhausted, remaining variables are set to zero.

Output

Characters are transferred, beginning with the first physical character and continuing through the Physical Length.

The first character is written at the current position within the file.

PARTIAL IO

Partial I/O allows processing of a portion of a record followed by other intermediate processing before the remainder of the record is processed.

Partial I/O is possible through use of a semi- colon (;) as the terminator for an I/O instruction.

The semi-colon at the end of an I/O list instructs the program to leave the file pointers positioned immediately after the last character processed, rather than skipping to the beginning of the next data record.

Example:

A data file has two types of records (each in a different format) as follows:

```
Position:  1234567890123456789012345678901234567890
Type 1  :  1NAME           ADDRESS        CITY ST
Type 2  :  2 234.67  100.00   50.00   84.67     .00
```

By doing a partial read on the first position only, it is possible to determine which type of record is being read:

```
     READ      INFILE,SEQ;TYPE;
     GOTO      EOJ IF OVER
     IF        ( TYPE = 1 )
     READ      INFILE,SEQ;NAME,ADDRESS,CITY,STATE
     ELSEIF    ( TYPE = 2 )
     READ      INFILE,SEQ;TOTAL,CURR,OV30,OV60,OV90
     ENDIF
```

It is possible to perform multiple partial reads on the same record before continuing.

If the End Of Record (EOR) is reached during a partial read, all variables are processed as defined in the previous sections.

To continue processing the next logical record, a non-partial read must be performed to complete the reading of the record and position the file pointer at the next record.

PARTIAL IO – continued

It is also possible to write part of a data record, do additional processing and then write the rest of the record.

If the write list is terminated with a semi-colon (;), the program does not automatically write the End Of Record. Instead, it remains positioned so additional writes to the data record are possible. This could be done as follows:

```
Position:  1234567890123456789012345678901234567890
Type 1  :  1NAME           ADDRESS         CITY ST
Type 2  :  2 234.67  100.00   50.00   84.67
```

Using this format, the following example performs a partial write to the first position of the file, followed by additional processing that determines what the remainder of the record should contain. The rest of the record is then written without partial I/O.

```
     WRITE    OUTFILE,SEQ;TYPE;
     IF       (TYPE = 1)
     WRITE    OUTFILE,SEQ;NAME,ADDRESS
     ELSE
     WRITE    OUTFILE,SEQ;TOTAL,CURR,OV30
     ENDIF
```

As with the read, it is also possible to perform multiple partial write operations before continuing with processing.

Any EOR must be explicitly written or a non-partial write must be performed.

It is not possible to perform partial output on IFILE and AFILE definitions.

END OF RECORD / END OF FILE MARKS

The End Of File (EOF) and End Of Record (EOR) marks used by this PL/B implementation are both in compliance and compatible with the operating system standard on which it is used.

End Of File

No End Of File (EOF) terminator is actually used.

The directory entry for the file is updated to point at the last byte in the file and the file is then truncated to that point.

End Of Record

The End Of Record (EOR) mark is defined by the standard for the operating system in use.

Under DOS/Windows, an EOF is a Carriage Return and Line Feed (Hex 0D and 0A).

Under Linux, the EOF is a Line Feed (Hex 0A).

This technique simplifies the functionality of the files with other programs and utilities.

CHAPTER ELEVEN

SEQUENTIAL DISK ACCESS

SEQUENTIAL FILE ACCESS METHOD

Sequential access is the simplest method of processing records within a data file. Records within the data file are directly processed sequentially forward or backward.

Basic Instructions used in sequential file accessing are:

Verb	**Purpose**
FILE	Define a file variable.
PREPARE	Create a new file or open an existing file
OPEN	Open an existing file
CLOSE	Close an open file and optionally delete it
READ	Retrieve data from a file
WRITE	Output data to a file
FPOSIT	Retrieve the current file position
REPOSIT	Set the file current file position

FILE

The FILE instructions define files that are to be used for either sequential or random file I/O.

```
MYFILE    FILE
```

A FILE must have been defined with a FILE Definition instruction before being referenced in a file I/O instruction.

Optional Parameters include:

BUFFER={nnnn}
This parameter establishes a buffer for I/O operations equal to {nnnn} bytes. When performing sequential I/O, this buffer is used strictly for read and write caching. When performing random I/O, the BUFFER is the actual sector length, including the EOR character(s).

COMP or **COMPRESSED**
Allows records within the file to be space compressed and of variable length. The VAR parameter is assumed in this case. Space compression is performed using the operating system standards.

FIX={nnnn} or **FIXED={nnnn}**
When performing both random I/O and sequential I/O, the byte count is automatically (by the compiler) incremented by one (Linux) or two (MS-DOS/Windows) to accommodate the EOR character(s) up to the defined record length. For WRITEs, records longer than the defined length will be truncated and shorter records will be padded with spaces to the defined record length.

FILE – continued

FORMAT=MSDOS or **FORMAT=UNX**
This parameter sets the end of record indicator within the file. The MSDOS type terminates records with a carriage return followed by a line feed for an end of record mark. The UNX type terminates records with a single line feed.

UNCOMP or **UNCOMPRESSED**
This parameter does not permit records to be space compressed. However, variable length records may still be created if the VARIABLE parameter is also present.

VAR={nnnn} or **VARIABLE={nnnn}**
This parameter functions as does the BUFFER parameter for FILE definitions.

If none of the above parameters are given, BUFFER=256 is assumed. This establishes a buffer and sector length of 256 bytes for all I/O operations.

PREPARE

A file must either be created or opened prior to any attempt to process data against it. The PREPARE instruction (PREP) instruction either opens an existing file or creates a new file if it does not exist.

```
     PREPARE  {file},{txtname}[,{mode}]
```

{txtname} is the name of the physical text file opened. If a character string variable is specified, only the Logical String is used.

If the file already exists, a PREPARE acts the same as an OPEN instruction.

The screen definition file determines whether an extension is required and what the default extension is.

If a specific drive and/or directory path is not provided, the current directory is checked for the file. If unsuccessful, the directories that have been established in the PLB_PATH environment variable or SEARCHPATH instruction are searched.

If still unable to locate the file, it is created in the directory specified by the PLB_PREP environment variable or the current directory.

The optional {mode} parameter, if used, must be the last parameter on the instruction.

The {mode} may be EXCLUSIVE, READ, SHARE, and SHARENF. If not specified, a MODE of SHARE is assumed.

If the same File Definition Label ({file}) is used for more than one PREPARE instruction within the same program, the currently open file is CLOSEd before an attempt is made to initialize the new file.

When a CLOSE is the first I/O instruction following the PREPARE, the file is deleted from the disk. This may also be accomplished with the ERASE verb.

OPEN

A file must either be created or opened prior to any attempt to process data against it. The OPEN instruction initializes an existing file for later access by the program.

```
     OPEN{file},{name}[,{mode}
```

{name} is the name of the physical text file to be opened. If a character string variable is specified, only the Logical String is used.

The screen definition file determines whether an extension is required and what the default extension is.

If a specific drive and/or directory path is not provided, the current directory is checked for the file. If unsuccessful, the directories that have been established in the PLB_PATH environment variable or SEARCHPATH instruction are searched.

Attempting to OPEN a file that does not exist or using a mode (SHARE, SHARENF, EXCLUSIVE, and READ) that is either incompatible with the privileges set for the file or with a mode the file is already open in, results in a trappable I/O error.

The optional {mode} parameter may be EXCLUSIVE, READ, SHARE, and SHARENF. If not specified, a MODE of SHARE is assumed.

If the same File Definition Label ({file}) is used for more than one OPEN instruction within the same program, the currently open file is CLOSEd before an attempt is made to initialize the new file.

CLOSE

The CLOSE instruction flushes all data to the disk and updates the disk directory entry for the file with the latest file attributes.

```
     CLOSE    {file}[,{mode}]
```

The CLOSE instruction does not automatically set the End Of File (EOF) mark. To set the End of File mark, use the WEOF instruction.

If CLOSE is the first I/O instruction after a PREPARE of a FILE declaration, the file named in the PREPARE statement is deleted from the disk.

The optional {mode} parameter may be DELETE to close the file and delete it if it was opened in EXCLUSIVE mode.

Example:

```
FILE1    FILE
.
         TRAP     PREPFILE IF IO
         OPEN     FILE1,"DATA01.TXT"
         TRAPCLR  IO
         ...
         CLOSE    FILE1
         STOP
.
PREPFILE
         PREPARE  FILE1,"DATA01.TXT"
         RETURN
```

READ

The READ instruction retrieves information from disk and transfers that information into data variables.

```
     READ{file},{record};{list}[;]
```

If {record} is a negative number, it indicates sequential file processing as follows:

Value	Action
-1	Sequential forward processing
-2	Sequential forward processing
-3	Position to End Of File (EOF) prior to processing
-4	Sequential backward processing

{list} may be any combination of character string and/or numeric variables (including arrayed items), character string and/or numeric literals and/or supported list controls separated by commas.

If an attempt is made to read past the End Of File (EOF) indicator (using -1 or -2) or if an attempt is made to read before the beginning of the file (using -4), the OVER Condition Flag is set (TRUE) and the LESS Condition Flag is cleared (FALSE). All numeric variables in the list are set to zero (0) and all character string variables in the list are blank-filled and CLEARed.

If a semi-colon (;) terminates the list, the rules relevant to partial reads apply.

READ LIST CONTROLS

The supported list controls for READ are as follows:

Control	Function
*ABSON	Enable absolute transfer of file data
*ABSOFF	Disable *ABSON
*CDFOFF	Disable *CDFON
*CDFON	Enable Comma Delimited Field support
*EDIOFF	Disable *EDION
*EDION	Enable Electronic Data Information support
*LC	Disable *UC (default)
*LL	Set the Length Pointer to last non-blank character
*{n}	Tab to the {n}'th character offset in the given record
*PL	Disable *LL (default)
*UC	Convert lower-case data to upper-case

READ EXAMPLE

The following example will read and display all the records in a file.

```
CUSTNO    FORM      7
NAME      DIM       30
SEQ       FORM      "-1"
FILE1     FILE
.
          TRAP      NOFILE IF IO
          OPEN      FILE1,"DATA01.TXT"
          TRAPCLR   IO
.
          LOOP
          READ      FILE1,SEQ;CUSTNO,NAME
          UNTIL     OVER
          DISPLAY   CUSTNO,":",NAME
          REPEAT
.
          CLOSE     FILE1
          STOP
.
NOFILE
          DISPLAY   *B,*HD,"DATA01.TXT Not Found."
          STOP
```

WRITE

The WRITE instruction transfers information from program variables into a disk file.

```
     WRITE    {file},{record};{list}[;]
```

If {record} is a negative number, it indicates sequential file processing as follows:

Value	**Action**
-1	Sequential forward processing
-2	Sequential forward processing
-3	Position to End Of File (EOF) prior to processing
-4	Sequential backward processing

{list} may be any combination of character string and/or numeric variables (including arrayed items), character string and/or numeric literals, control values and/or supported list controls separated by commas

If a semi-colon (;) terminates the list, the rules relevant to partial writes apply.

Space compression is disabled unless explicitly enabled through *+.

Any short records written to the file are padded with blanks to the record length.

WRITE LIST CONTROLS

The supported list controls are as follows:

Control	Function
*+	Enable space compression
*-	Disable space compression
*CDFOFF	Disable *CDFON
*CDFON	Enable Comma Delimited Field support
*EDIOFF	Disable *EDION
*EDION	Enable Electronic Data Information support
*LC	Data is written in the case in which it was stored
*LL	Forces output of only the Logical String
*MP	Minus overpunch negative numeric variables
*PL	Disable *LL (default)
*UC	Convert lower-case characters to upper-case
*ZF	Zero-fill numeric variables
*ZS	Blank-fill zero numeric variables

WRITE EXAMPLE

The following example program will write one hundred records to a file sequentially.

```
CUSTNO    FORM      7
NAME      DIM       30
FILE1     FILE
SEQ       FORM      "-1"
.
          TRAP      NOPREP IF IO
          PREP      FILE1,"DATA01.TXT"
          TRAPCLR   IO
.
          LOOP
          INCR      CUSTNO
          PACK      NAME WITH "CUSTOMER",CUSTNO
          SQUEEZE   NAME,NAME
          WRITE     FILE1,SEQ;CUSTNO,NAME
          COMPARE   "100",CUSTNO
          REPEAT    WHILE NOT EQUAL
.
          CLOSE     FILE1
          STOP
.
NOPREP
          DISPLAY   *B,*HD,"Unable to create DATA01.TXT"
          STOP
```

FPOSIT

The FPOSIT instruction retrieves the current position of the file pointer.

```
     FPOSIT   {file},{offset}
```

The current byte position is returned in {offset}.

The result of this instruction is compatible with the REPOSIT instruction.

The current file position remains unchanged.

If the file position is zero, the ZERO (or EQUAL) condition flag is set (TRUE).

If any variable is not large enough to hold the appropriate position, the result is indeterminate and the OVER condition flag is set (TRUE).

REPOSIT

The REPOSIT instruction repositions the file's pointer to the specified value. This instruction is normally used in conjunction with the FPOSIT instruction to save and restore a file's pointer.

```
     REPOSIT  {file}{sep}{offset}
```

The value contained in {offset} is where the file pointer is set.

If it equals zero (0), the file pointer is set to the beginning of the file.

If the specified location is at the End Of File (EOF), the OVER Condition Flag is set (TRUE).

If it is past the EOF, the ZERO Condition Flag is set (TRUE).

FPOSIT / REPOSIT EXAMPLE

```
FILE1     FILE
POS       FORM      7
DATA      DIM       20
SEQ       FORM      "-1"
ZERO      FORM      "0"
INDEX     FORM      1
.
          OPEN      FILE1,"DATA01.TXT"
*
.Read the 5 records
.
          FOR       INDEX,"1","5"
          READ      FILE1,SEQ;DATA
          DISPLAY   DATA
          REPEAT
*
.Note our file position
.
          FPOSIT    FILE1,POS
*
.Position to and read the first record
.
          REPOSIT   FILE1,ZERO
          READ      FILE1,SEQ;DATA
          DISPLAY   DATA
*
.Restore our saved position
.
          REPOSIT   FILE1,POS
*
.Read and display records 6 through 10
.
          FOR       INDEX,"1","5"
          READ      FILE1,SEQ;DATA
          DISPLAY   DATA
          REPEAT
.
          CLOSE     FILE1
          STOP
```

EXERCISE 17

Write a program that creates a customer data file (custdata.txt).

1. The file should contain the following fields:

Type	Size	Description
N	7	Number
AN	30	Name
AN	30	Address
AN	20	City
AN	2	State
N	5	Zip code
N	7.2	Balance

2. Create ten records with dummy data of your own design.

3. The number and name fields should be unique.

EXERCISE 18

1. Create a copy of Exercise 16.
2. Add logic to open the file created in Exercise 17 as the program begins.
3. Include appropriate logic to handle open failures.
4. Replace the code that displays the information when the data entry is complete with code that writes the data to the end of the file.
5. Upon exit, list the contents of the file on the screen before closing it.

CHAPTER TWELVE

ISAM DISK ACCESS

INDEX SEQUENTIAL ACCESS METHOD

Index sequential access is a most common method of accessing data files in PL/B Records within the data file are located using a key. The access method also allows reading forward or backward in key sequential order.

Basic Instructions used in ISAM file accessing are:

Verb	**Purpose**
IFILE	Define a index sequential file variable.
PREPARE	Create a new file or open an existing file and its key file.
OPEN	Open an existing file and its key file
CLOSE	Close an open file and optionally delete it
READ	Retrieve data from a file
READKS	Read the next record key sequentially.
READKP	Read the previous record key sequentially
UPDATE	Modify the current data record
WRITE	Output data to a file
INSERT	Add a new key for the current data record.
DELETE	Delete a record from a disk file.
DELETEK	Delete an ISAM key.

IFILE

The IFILE instruction defines an index sequential file. These files may be read by index keys directly or key sequentially forward and backwards. Individual records may be read, written, updated or deleted. IFILEs may also be read sequentially and randomly.

```
MYIFILE IFILE    [{parm}]
```

An IFILE label must have been defined before being referenced in an instruction.

IFILE definitions reference two physical files on the system. These files perform particular functions within the ISAM implementation as follows:

Type	Use
TXT	The data file containing all of the text records.
ISI	The index to the records in the TXT file

{parm} is the optional parameters as follows:

BUFFER={nnnn}
This parameter establishes a maximum record length (not including the EOR character(s)) of {nnnn} bytes. In addition (unless the COMPRESSED option is given), the records in the file are fixed length and are not space compressed. Any short records written to the file are padded with blanks to the record length.

COMP or **COMPRESSED**
This parameter allows records within the file to be space compressed and of variable length. The VAR parameter is assumed in this case.

DUP or **DUPLICATES**
This parameter permits records with duplicate keys to be written to the file. This is the default parameter.

IFILE – continued

FORMAT=MSDOS or **FORMAT=UNX**
This parameter sets the end of record indicator within the file. The MSDOS type terminates records with a carriage return followed by a line feed for an end of record mark. The UNX type terminates records with a single line feed.

NODUP or **NODUPLICATES**
This parameter disallows records with duplicate keys to be written to the file. If an attempt is made to write a duplicate key, an I/O error occurs.

UNCOMP or **UNCOMPRESSED**
This parameter does not permit records to be space compressed unless the VAR parameter is also present.

UPPER or **UPPERCASE**
This parameter treats all key data as upper case. When an I/O instruction occurs, the key information given is converted to upper case. The actual data written to the text file is not modified.

VAR={nnnn} or **VARIABLE={nnnn}**
This parameter allows records written to the text file to be of variable length with a maximum length of {nnnn} bytes.

WEOF
This parameter forces all write operations to be output at the End Of File. Deleted record space is not reused. This option is assumed if the COMP or VAR parameters are given. The WEOF parameter may result in faster I/O.

Most parameters specified on the IFILE definition are overridden at execution by the actual characteristics of the file. The exception to this rule is the actual (maximum if variable length or space compressed records) record length specified with BUFFER, FIXED or VARIABLE parameters.

Care should be used if performing UPDATEs against variable length or space-compressed records. UPDATEs are not recommended with space compressed data files.

PREPARE

A file must either be created or opened prior to any attempt to process data against it. The PREPARE instruction (PREP) instruction either opens an existing file and key file or creates a new file and key file if the file does not exist.

```
     PREPARE {ifile},{txtname},{isiname}:
               {keyspecs},{maxreclen}[,{mode}]
```

{txtname} is the name of the physical text file (txt) prepared and {isiname} is the name of the ISAM key file (isi).

The current screen definition file in use determines the text and ISAM file default extensions.

If a specific drive and/or directory path is not provided, the current directory is checked for the file. If unsuccessful, the directories that have been established in the PLB_PATH environment variable or SEARCHPATH instruction are searched. If still unable to locate the file, it is be created in the current directory specified by the PLB_PREP environment variable or the current directory if PLB_PREP is not present in the environment table.

The file name (including drive and/or path designation) must be in the valid operating system format and in the correct case for case sensitive systems like Linux.

PREPARE – continued

If {isiname} exists, its ISAM tree is cleared. {txtname}, if it exists, is also cleared and all data records overwritten.

The {keyspecs} parameter can be specified either as a numeric variable or literal or as a string variable or literal.

- When a numeric variable or literal value is detected, the value identifies the maximum key length for the ISAM file being created.
- When the {keyspecs} is specified as a string variable or literal, the string must contain key specifications as used by the SUNINDEX utility.

{maxreclen} must be a valid numeric variable, literal or constant in the range of one (1) to 32,767 inclusive that defines the maximum record size written to the IFILE.

If not specified, a {mode} of SHARE is assumed.

If the same File Definition Label {ifile} is used for more than one PREPARE instruction within the same program, the currently open file is CLOSEd before an attempt is made to initialize the new file.

OPEN

A file must either be created or opened prior to any attempt to process data against it. The OPEN instruction initializes an existing file for later access by the program.

```
     OPEN     {ifile},{name}[,{mode}
```

{name} is the name of the ISAM key file to be opened.

The current screen definition file determines the text and ISAM file extension used.

If a specific drive and/or directory path is not provided, the current directory is checked for the file. If unsuccessful, the directories that have been established in the PLB_PATH environment variable or SEARCHPATH instruction are searched.

The file name (including drive and/or path designation) must be in the valid operating system format and in the correct case for case sensitive systems like Linux.

If the same File Definition Label ({ifile}) is used for more than one OPEN instruction within the same program, the currently open file are CLOSEd before an attempt is made to initialize the new file.

The name of the physical text file is retrieved from the {name} file. If a specific drive or path was not incorporated as part of the physical text file name when the ISAM key file was created, a search of all the Logged On Drives is made for the physical text file using the guidelines previously explained.

The maximum key and record lengths are also retrieved from the {name} file.

CLOSE

The CLOSE instruction flushes all data to the disk and updates the disk directory entry for the file with the latest file attributes.

```
     CLOSE    {ifile}[,{mode}]
```

If CLOSE is the first I/O instruction after a PREPARE of a IFILE declaration, the file is simply closed

The optional {mode} parameter may be DELETE to close the file and delete it if it was opened in EXCLUSIVE mode.

Example:

```
IFILE1   IFILE
.
         TRAP     PREPFILE IF IO
         OPEN     IFILE1,"DATA01.ISI"
         TRAPCLR  IO
         ...
         CLOSE    IFILE1
         STOP
.
PREPFILE
         PREPARE  IFILE,"DATA01.TXT","DATA01.ISI":
                  "14","155",EXCLUSIVE
         RETURN
```

READ

The READ instruction retrieves information from disk and transfers that information into data variables. The statement format is as follows:

```
     READ     {ifile},{key};{list}[;]
```

If {key} is a numeric variable equal to or greater than zero, it designates a sector in the file for random or direct access.

If it is a negative numeric variable equal to -1, -2, -3 or -4, sequential file processing is indicated.

{list} may be optionally excluded if two semi-colons (;;) are specified after {key}. This technique only positions the file pointer to the beginning of the record/sector without actually reading any data.

When reading with a NULL key, the last read ISAM record is re-read.

Attempting to read a nonexistent ISAM record sets the OVER Condition Flag (TRUE) and clears the LESS Condition Flag (FALSE). All variables are left in their original. The pointer in the ISAM key file is then positioned where the record should have been so that a READKS (Read Key Sequential) instruction would retrieve the next sequential key record in the ISAM tree.

If a semi-colon (;) terminates the list, the rules relevant to partial reads apply (see Partial I/O).

The supported list controls defined for sequential READ apply.

READ EXAMPLE

The following example will read and display one record.

```
CUSTNO    FORM      7
NAME      DIM       30
KEY       INIT      "     30"
IFILE1    IFILE
.
          TRAP      NOFILE IF IO
          OPEN      IFILE1,"DATA01.ISI",READ
          TRAPCLR   IO
.
          READ      IFILE1,KEY;CUSTNO,NAME
          IF        OVER
          DISPLAY   *B,*HD,"Record Not Found."
          ELSE
          DISPLAY   CUSTNO,":",NAME
          ENDIF
.
          CLOSE     IFILE1
          STOP
.
NOFILE
          DISPLAY   *B,*HD,"DATA01.ISI Not Found."
          STOP
```

READKS / READKP

The READKS and READKP instructions read ISAM text files in ascending or descending key sequence without the necessity of specifying each key. The instructions use the following formats:

```
     READKS   {ifile};{list}[;]
     READKP   {ifile};{list}[;]
```

The OVER flag indicates that the end or beginning of file was encountered.

If a READKS is performed prior to any other ISAM read instruction, it begins with the lowest logical key in the ISAM key file.

If a semi-colon (;) terminates the list, the rules relevant to partial reads apply (see Partial I/O).

The supported list controls defined for sequential READ apply.

READKS / READKP EXAMPLE

The following example will read and display all the records in a file key sequentially from first to last and then from last to first.

```
CUSTNO    FORM      7
NAME      DIM       30
IFILE1    IFILE
.
          TRAP      NOFILE IF IO
          OPEN      IFILE1,"DATA01.ISI",READ
          TRAPCLR   IO
*
.Forward
.
          READKS    IFILE1;CUSTNO,NAME
          UNTIL     OVER
          DISPLAY   CUSTNO,":",NAME
          REPEAT
*
.Backward
.
          READ      IFILE,"9999999";;
.
          READKP    IFILE1,KEY;CUSTNO,NAME
          UNTIL     OVER
          DISPLAY   CUSTNO,":",NAME
          REPEAT
.
          CLOSE     IFILE1
          STOP
.
NOFILE
          DISPLAY   *B,*HD,"DATA01.ISI Not Found."
          STOP
```

UPDATE

The UPDATE instruction allows a record to be partially or completely updated in place.

```
     UPDATE   {ifile};{list}[;]
```

This instruction modifies the last ISAM record read. A valid ISAM read or write instruction must have been successfully completed prior to the UPDATE instruction or an I/O error occurs.

The `key' information in the ISAM key file remains unchanged, even though the key data within the text file may have been modified.

Note that if the file is space compressed or contains variable length records and the physical length of the updated data is greater than the physical length of the record on disk, the data is truncated and the OVER flag is set. In a space compressed file, if a field is changed that affects the number of spaces, it is highly probable that the physical length also changes. In no event is the actual record on disk increased in physical length.

Using a semicolon (;) as a terminator for partial I/O has no effect since the UPDATE instruction does not write an End Of Record (EOR) mark (an implied partial write). For the same reason, a sequential READ (-1 or -2) after an UPDATE begins immediately after the last updated field.

The ISAM key file pointer is not destroyed by an UPDATE instruction. Therefore, it is possible to perform the following sequence of events.

```
        READ     IFILE,KEY;VARIABLES...
        GOTO     ERROR IF OVER
        ...
UPDATEX UPDATE   IFILE;VARIABLES...
        READKS   IFILE;VARIABLES...
        GOTO     OVER1 IF OVER
        ...
        GOTO     UPDATEX
```

UPDATE – continued

The supported list controls are as follows:

Control	Function
*CDFOFF	Disable *CDFON
*CDFON	Enable Comma Delimited Field support
*EDIOFF	Disable *EDION
*EDION	Enable Electronic Data Information support
*LL	Forces output of only the Logical String
*MP	Minus overpunch negative numeric variables
*{n}	Tab to the {n}'th character offset in the given record
*PL	Disable *LL (default)
*TAB={n}	Tab to the {n}'th character offset in the given record
*ZF	Zero-fill numeric variables
*ZS	Blank-fill zero numeric variables

Example:

```
MYFILE   IFILE
KEY      INIT     "SMITH"
DATA1    INIT     "ABC"
DATA2    INIT     "XYZ"
.
         OPEN     MYFILE,"MyISAM01"
         READ     MYFILE,KEY;;
         IF       OVER
         DISPLAY  *B,*HD,"Record not found.";
         ELSE
         UPDATE   MYFILE;DATA1,*120,DATA2
         ENDIF
.
         CLOSE    MYFILE
```

WRITE

The WRITE instruction transfers information from program variables into a disk file.

```
     WRITE    {ifile}[,{key}];{list}[;]
```

The Logical String of {key} is the index key associated with this record.

If {key} is not specified, the key is constructed using the data in {list}.

Duplicate ISAM keys are allowed unless the `N' option was used during the most recent indexing of the file.

If a semi-colon (;) terminates the list, the rules applying to partial writes apply (see Partial I/O).

Any records written to a fixed length record file that are shorter than the declared record size are padded with blanks and the LESS flag is set.

Any records written to a fixed length record file that are longer than the declared record size or to a variable length record file that are longer than the maximum record size are truncated and the OVER flag is set.

WRITE – continued

The supported list controls are as follows:

Control	Function
*+	Enable space compression
*-	Disable space compression
*CDFOFF	Disable *CDFON
*CDFON	Enable Comma Delimited Field support
*EDIOFF	Disable *EDION
*EDION	Enable Electronic Data Information support
*LC	Data is written in the case in which it was stored
*LL	Forces output of only the Logical String
*MP	Minus overpunch negative numeric variables
*PL	Disable *LL (default)
*UC	Lower-case characters are converted to upper-case
*ZF	Zero-fill numeric variables
*ZS	Blank-fill zero numeric variables

Example:

```
MYFILE    IFILE
CUSTNO    FORM      "     32"
NAME      INIT      "THOMAS EDWARDS"
.
          OPEN      MYFILE,"MyISAM01"
          WRITE     MYFILE;CUSTNO,NAME
          CLOSE     MYFILE
          STOP
```

INSERT

The INSERT instruction allows ISAM record keys to be inserted into an alternate key file. It must immediately follow a successful READ, WRITE, or DELETEK instruction.

```
     INSERT   {ifile}[,{key}]
```

The Logical String of {key} is inserted into the ISAM key file. If {key} is a Null String, the key is extracted from the record data.

An INSERT should be performed against each secondary index after a WRITE to the primary text file. Otherwise, the secondary index may be unusable.

Since an ISAM WRITE automatically inserts the key into the primary index, this instruction should not be used for ISAM files with only one ISAM key file.

INSERT only works immediately after a READ, WRITE or DELETEK to the primary text file. Any other instruction type may cause the INSERT to perform incorrectly.

An attempt to insert a duplicate key into a file indexed with the `N' (NODUP) option generates an error.

INSERT – continued

Example:

```
MYFILE   IFILE
MYFILE2  IFILE
.
CUSTNO   FORM     "     32"
NAME     INIT     "THOMAS EDWARDS"
.
         OPEN     MYFILE,"MyISAM01"
         OPEN     MYFILE2,"MyISAM02"
.
         WRITE    MYFILE;CUSTNO,NAME
         INSERT   MYFILE2
.
         CLOSE    MYFILE
         CLOSE    MYFILE2
.
         STOP
```

DELETE / DELETEK

The DELETE instruction allows records to be physically deleted from the text file and for its key to be physically deleted from the ISAM file.

```
     DELETE   {ifile},[{key}]
     DELETEK  {ifile},{key}
```

The Logical String of {key} retrieves and then deletes both the text record and associated ISAM key.

If the specified key is not found, the OVER Condition Flag is set to TRUE. If {key} is a Null String or not specified, the last ISAM record read in {ifile} is deleted.

Deleted record space on fixed record length files is automatically reused by subsequent WRITE instructions (unless the WEOF parameter was specified for the file).

A DELETE or DELETEK must be performed against each alternate or secondary, index file that requires the key to be removed.

When deleting from a physical text file having both an ISAM and an AAM key file, the AAM DELETE should be performed first since it requires that the text record be present for the key information to be deleted.

The DELETEK instruction allows ISAM keys to be deleted without deleting the text record. This instruction is primarily used when maintaining an alternate index.

EXERCISE 19

1. Use SUNINDEX (Appendix B) to create two index files on the customer data file created previously.

   - The first index (custdata.isi) should be on the customer number (1-7).

   - The second index (custdat2.isi) should be on customer name and number (8-37,1-7).

   - Copy Exercise 16 to Exercise 19.

   - Modify the program to open both index files

   - Modify the write logic to add new records via the index files. Be sure to maintain the secondary index.

   - Modify the display logic to list the records via the primary index (custdata.isi) anytime the F2 key is pressed.

   - Modify the display logic to list the records via the secondary index (custdata2.isi) anytime the F3 key is pressed.

   - Modify the program to simply close the files and exit when the escape key is pressed.

EXERCISE 20 – optional

1. Begin with Exercise 19.
2. After displaying the input form, display a prompt and accept a command. The prompt should read
3. "(A)dd, (L)ist, (D)elete, (N)ext, (P)revious, or (Q)uit: "
4. An "A" command should perform the existing record addition logic.
5. A "L" command should list the records using the secondary index.
6. The "D" command should delete the current record.
7. The "N" command should retrieve the next record. If there are no more records, an appropriate message should be displayed.
8. The "P" command should retrieve the previous record. If there are no more records, an appropriate message should be displayed.

CHAPTER THIRTEEN

AAM DISK ACCESS

ASSOCIATIVE ACCESS METHOD

Associative access method (AAM) files allow the user to process logical records based on collective key information.

Similar to ISAM, this method uses a control file (AAM file) to determine which logical record is to be processed based upon the key information specified.

Unlike ISAM, AAM does not require a complete or specific key. However, the key information must have been extracted from data written within the logical record.

Basic Instructions used in AAM file accessing are:

Verb	**Purpose**
AFILE	Define an associative access file variable.
PREPARE	Create a new file or open an existing file and its key file.
OPEN	Open an existing file and its key file.
CLOSE	Close an open file and optionally delete it.
READ	Retrieve data from a file.
READKG	Read the next record key sequentially.
READKGP	Read the previous record key sequentially.
UPDATE	Modify the current data record.
WRITE	Output data to a file.
INSERT	Add a new key for the current data record.
DELETE	Delete a record from a disk file.

AFILE

The AFILE instruction defines an Associative Access Method (AAM) file. AAM is a method of retrieving data records using one or more free form key specifications.

```
MYAFILE   AFILE     [{parm}]
```

An AFILE label must have been defined before being referenced in an instruction.

AFILE definitions reference two physical files on the system. These files perform particular functions within the AAM implementation as follows:

Type	**Use**
TXT	The data file containing all of the text records.
AAM	The index to the records in the TXT file

{parm} is the optional parameters as follows:

BUFFER={nnnn}
This parameter establishes a maximum record length (not including the EOR character(s)) of {nnnn} bytes. In addition (unless the COMPRESSED option is given), the records in the file are fixed length and are not space compressed. Any short records written to the file are padded with blanks to the record length.

COMP or **COMPRESSED**
This parameter allows records within the file to be space compressed and of variable length. The VAR parameter is assumed in this case.

AFILE – continued

FORMAT=MSDOS or **FORMAT=UNX**
This parameter sets the end of record indicator within the file. The MSDOS type terminates records with a carriage return followed by a line feed for an end of record mark. The UNX type terminates records with a single line feed.

UNCOMP or **UNCOMPRESSED**
This parameter does not permit records to be space compressed unless the VAR parameter is also present.

UPPER or **UPPERCASE**
This parameter treats all key data as upper case. When an I/O instruction occurs, the key information given is converted to upper case. The actual data written to the text file is not modified.

VAR={nnnn} or **VARIABLE={nnnn}**
This parameter allows records written to the text file to be of variable length with a maximum length of {nnnn} bytes.

WEOF
This parameter forces all write operations to be output at the End Of File. Deleted record space is not reused. This option is assumed if the COMP or VAR parameters are given. The WEOF parameter may result in faster I/O.

Most parameters specified on the AFILE definition are overridden at execution by the actual characteristics of the file. The exception to this rule is the actual (maximum if variable length or space compressed records) record length specified with BUFFER, FIXED or VARIABLE parameters.

Care should be used if performing UPDATEs against variable length or space-compressed records. UPDATEs are not recommended with space compressed data files.

PREPARE

A file must either be created or opened prior to any attempt to process data against it. The PREPARE instruction (PREP) instruction either opens an existing file and key file or creates a new file and key file if the file does not exist.

```
     PREPARE {afile},{txtname},{aamname}:
               {keyspecs},{maxreclen}[,{mode}]
```

{txtname} is the name of the physical text file (txt) prepared and {aamname} is the name of the AAM key file (isi).

The current screen definition file in use determines the text and AAM file default extensions.

If a specific drive and/or directory path is not provided, the current directory is checked for the file. If unsuccessful, the directories that have been established in the PLB_PATH environment variable or SEARCHPATH instruction are searched. If still unable to locate the file, it is be created in the current directory specified by the PLB_PREP environment variable or the current directory if PLB_PREP is not present in the environment table.

The file name (including drive and/or path designation) must be in the valid operating system format and in the correct case for case sensitive systems like Linux.

PREPARE – continued

If {aamname} exists, its AAM tree is cleared. {txtname}, if it exists, is also cleared and all data records overwritten.

{maxreclen} must be a valid numeric variable, literal or constant in the range of one (1) to 32,767 inclusive that defines the maximum record size written to the IFILE.

If not specified, a {mode} of SHARE is assumed.

If the same File Definition Label {ifile} is used for more than one PREPARE instruction within the same program, the currently open file is CLOSEd before an attempt is made to initialize the new file.

{keyspecs} is a character string variable or literal containing up to 95 AAM key specifications and any options selected. If {keyspecs} is a character string, only the Logical String is used.

The AAM keys may overlap (1-10,5-15) or be wholly within another (1-10,3-9) and may be located anywhere within the record.

Example:

```
MYAFILE   AFILE
.
          PREP      MYAFILE,"EMPL","EMPL1":
                    "U,3-5,10-20","100"
          …
          CLOSE     MYAFILE
          STOP
```

OPEN

A file must either be created or opened prior to any attempt to process data against it. The OPEN instruction initializes an existing file for later access by the program.

```
     OPEN     {afile},{name}[,{mode}
```

{name} is the name of the AAM key file to be opened.

The current screen definition file determines the text and AAM file extension used.

If a specific drive and/or directory path is not provided, the current directory is checked for the file. If unsuccessful, the directories that have been established in the PLB_PATH environment variable or SEARCHPATH instruction are searched.

The file name (including drive and/or path designation) must be in the valid operating system format and in the correct case for case sensitive systems like Linux.

If the same AFILE Definition Label ({afile}) is used for more than one OPEN instruction within the same program, the currently open file are CLOSEd before an attempt is made to initialize the new file.

The name of the physical text file is retrieved from the {name} file. If a specific drive or path was not incorporated as part of the physical text file name when the AAM key file was created, a search of all the Logged On Drives is made for the physical text file using the guidelines previously explained.

The key information and record lengths are also retrieved from the {name} file.

CLOSE

The CLOSE instruction flushes all data to the disk and updates the disk directory entry for the file with the latest file attributes.

```
     CLOSE    {afile}[,{mode}]
```

If CLOSE is the first I/O instruction after a PREPARE of a AFILE declaration, the file is simply closed.

The optional {mode} parameter may be DELETE to close the file and delete it if it was opened in EXCLUSIVE mode.

Example:

```
AFILE1    AFILE
.
          TRAP      PREPFILE IF IO
          OPEN      AFILE1,"DATA01.AAM"
          TRAPCLR   IO
          …
          CLOSE     AFILE1
          STOP
.
PREPFILE
          PREPARE   AFILE,"DATA01.AAM","DATA01.TXT":
                    "U,3-5,10-20","100"
          RETURN
```

READ

The READ instruction retrieves information from disk and transfers that information into data variables. The statement format is as follows:

```
     READ     {afile},{{key},[...]};{list}[;]
```

If {key} is a numeric variable equal to or greater than zero, it designates a sector in the file for random or direct access.

If it is a negative numeric variable equal to -1, -2, -3 or -4, sequential file processing is indicated.

{list} may be optionally excluded if two semi-colons (;;) are specified after {key}. This technique only positions the file pointer to the beginning of the record/sector without actually reading any data.

The individual key specifications must be separated from other key specifications by commas.

Any Null String key specifications in {keys} are ignored. However, if all of the key specifications are null an attempt is made to re-read the last record retrieved by any AAM read instruction (READ, READKG, or READKGP). If a valid AAM read has not been previously performed, an I/O error occurs.

READ - continued

Each key is constructed as 'nnt{data}' with the format is as follows:

Value	Designates ...
nn	the key field number for the search. Each key specification or range is a unique key field number (i.e., using the key specifications '1-10,11-20,21-30' / 1-10 would be key field number 1, 11-20 would be key field number 2 and 21-30 key field number 3). nn must be two numeric digits and left zero-filled.
t	the type of search performed (i.e., X for eXact, R for Right, L for Left and F for Free - the search type character must be in upper-case).
{data}	the actual string of data to search for. The search string specified must conform to the requirements for the search type being used. The only exception to the rules is when the \`U' option has been specified during the AAM file creation (either through the PREPARE instruction or the AAMDEX utility). The \`U' option specifies that upper and lower-case characters are treated as equal during any searches.

READ – continued

The valid search types are as follows:

Type	Search
X	Exact, the search string must match all data within the key field specified. If the search string size is not equal to the key field size, it is treated as if it were truncated or blank-filled, as appropriate, to match the key field size. The wildcard character may be used as part of the search string if at least three contiguous non-blank, non-wildcard characters have been specified except if the field size is less than 3 characters.
R	Right, the search string must match the right most data in the key field. If the search string size is larger than the key field size, the search string is truncated and the search is treated as an Exact (X) search. The wildcard character maybe used as part of the search string if it is not the right most search character and at least one non-blank, non-wildcard character has been specified.
L	Left, the search string must match the left most data in the key field. If the search string size is larger than the key field size, the search string is truncated and the search is treated as an Exact (X) search. The wildcard character may be used as part of the search string if it is not the left most search character and at least one non-blank, non-wildcard character has been specified. If a blank is used as the leftmost character in the key, three non-blank, non-wildcard characters are required at some point following the blank or blanks. Otherwise, only one non-blank, non-wildcard character is required for a left search.
F	Free, the search string must be matched somewhere within the key field (a free floating check is made). This search type requires at least 3 bytes of search string data or an I/O error occurs. If the search string size is larger than the key field size, the search string is truncated and the search is treated as an Exact (X) search. The wildcard character maybe used as part of the search string if at least three contiguous non-blank, non-wildcard characters have been specified.

READ EXAMPLE

The following example will read and display one record.

```
CUSTNO    FORM      7
NAME      DIM       30
KEY       INIT      "01X     30"
AFILE1    AFILE
.
          TRAP      NOFILE IF IO
          OPEN      AFILE1,"DATA01.AAM",READ
          TRAPCLR   IO
.
          READ      AFILE1,KEY;CUSTNO,NAME
          IF        OVER
          DISPLAY   *B,*HD,"Record Not Found."
          ELSE
          DISPLAY   CUSTNO,":",NAME
          ENDIF
.
          CLOSE     AFILE1
          STOP
.
NOFILE
          DISPLAY   *B,*HD,"DATA01.AAM Not Found."
          STOP
```

READKG / READKGP

The READKG instruction allows retrieval of subsequent or previous records that met the key criteria for a prior AAM READ instruction.

```
     READKG    {afile};{list}[;]
     READKGP   {afile};{list}[;]
```

Once no further records meet the original AAM READ criteria, the OVER Condition Flag is set to TRUE and the LESS Condition Flag is set to FALSE. All variables are left in their original.

{list} may be optionally excluded if two semi-colons (;;) are specified after {afile}. This technique only positions the file pointer to the beginning of the record/sector without actually reading any data.

The FILE list controls are supported.

None of the key fields that were used in the READ instruction may be changed between the READ and the READKG instruction.

It is possible to interrupt out of a READKG instruction via a trapped function key. The current position in the file is maintained so that a READKG or READKGP may be issued if nothing else is done to the file. If you desire to continue reading from where you interrupted the AAM process, you should make the READKG or READKGP the last instruction in the trapped routine, just prior to the normal RETURN instruction.

READKG / READKGP EXAMPLE

The following example will read and display all the records in a file key sequentially from first to last and then from last to first.

```
CUSTNO    FORM      7
NAME      DIM       30
AFILE1    AFILE
.
          TRAP      NOFILE IF IO
          OPEN      AFILE1,"DATA01.AAM",READ
          TRAPCLR   IO
*
.Position the file
.
          READ      AFILE,"02LSMI";;
*
.Read Key Generically and Display
.
          LOOP
          READKG    AFILE1,KEY;CUSTNO,NAME
          UNTIL     OVER
          DISPLAY   CUSTNO,":",NAME
          REPEAT
.
          CLOSE     IFILE1
          STOP
.
NOFILE
          DISPLAY   *B,*HD,"DATA01.AAM Not Found."
          STOP
```

UPDATE

The UPDATE instruction allows a record to be partially or completely updated in place.

```
     UPDATE   {afile};{list}[;]
```

This instruction modifies the last AAM record read. A valid AAM read or write instruction must have been successfully completed prior to the UPDATE instruction or an I/O error occurs.

The `key' information in the AAM key file is updated along with the record.

Note that if the file is space compressed or contains variable length records and the physical length of the updated data is greater than the physical length of the record on disk, the data is truncated and the OVER flag is set. In a space compressed file, if a field is changed that effects the number of spaces, it is highly probable that the physical length also changes. In no event is the actual record on disk increased in physical length.

Using a semicolon (;) as a terminator for partial I/O has no effect since the UPDATE instruction does not write an End Of Record (EOR) mark (an implied partial write). For the same reason, a sequential READ (-1 or -2) after an UPDATE begins immediately after the last updated field.

The AAM key file pointer is not destroyed by an UPDATE instruction. Therefore, it is possible to perform the following sequence of events.

```
        READ     AFILE,KEY1,KEY2,KEY3;VARIABLES...
        GOTO     ERROR IF OVER
        ...
UPDATEX UPDATE   AFILE;VARIABLES...
        READKG   AFILE;VARIABLES...
        GOTO     OVER1 IF OVER
        ...
        GOTO     UPDATEX
```

UPDATE – continued

The supported list controls are as follows:

Control	Function
*CDFOFF	Disable *CDFON
*CDFON	Enable Comma Delimited Field support
*EDIOFF	Disable *EDION
*EDION	Enable Electronic Data Information support
*LL	Forces output of only the Logical String
*MP	Minus overpunch negative numeric variables
*{n}	Tab to the {n}'th character offset in the given record
*PL	Disable *LL (default)
*TAB={n}	Tab to the {n}'th character offset in the given record
*ZF	Zero-fill numeric variables
*ZS	Blank-fill zero numeric variables

Example:

```
MYFILE    AFILE
KEY       INIT      "02XJOHN SMITH"
DATA1     INIT      "ABC"
DATA2     INIT      "XYZ"
.
          OPEN      MYFILE,"MyAAM01"
.
          READ      MYFILE,KEY;;
          IF        OVER
          DISPLAY   *B,*HD,"Record not found.";
          ELSE
          UPDATE    MYFILE;*8,"SAM SMITH"
          ENDIF
.
CLOSE     MYFILE
          STOP
```

WRITE

The WRITE instruction transfers information from program variables into a disk file.

```
     WRITE    {afile};{list}[;]
```

The key information for the AAM key file is automatically extracted from the variables being written by the AAM WRITE instruction and need not be specified.

If a semi-colon (;) terminates the list, the rules applying to partial writes apply (see Partial I/O).

Any records written to a fixed length record file that are shorter than the declared record size are padded with blanks and the LESS flag is set.

Any records written to a fixed length record file that are longer than the declared record size or to a variable length record file that are longer than the maximum record size are truncated and the OVER flag is set.

WRITE – continued

The supported list controls are as follows:

Control	Function
*+	Enable space compression
*-	Disable space compression
*CDFOFF	Disable *CDFON
*CDFON	Enable Comma Delimited Field support
*EDIOFF	Disable *EDION
*EDION	Enable Electronic Data Information support
*LC	Data is written in the case in which it was stored
*LL	Forces output of only the Logical String
*MP	Minus overpunch negative numeric variables
*PL	Disable *LL (default)
*UC	Lower-case characters are converted to upper-case
*ZF	Zero-fill numeric variables
*ZS	Blank-fill zero numeric variables

Example:

```
MYFILE    AFILE
CUSTNO    FORM      "     32"
NAME      INIT      "THOMAS EDWARDS"
.
          OPEN      MYFILE,"MyAAM01"
          WRITE     MYFILE;CUSTNO,NAME
          CLOSE     MYFILE
          STOP
```

INSERT

The INSERT instruction allows AAM record keys to be inserted into an alternate key file. It must immediately follow a successful READ, WRITE, or DELETEK instruction.

```
     INSERT   {afile)
```

The INSERT instruction automatically extracts the key data from the last written record.

An INSERT should be performed against each secondary index after a WRITE to the primary text file. Otherwise, the secondary index may be unusable.

Since an AAM WRITE automatically inserts the key into the primary index, this instruction should not be used for files with only one AAM file.

INSERT only works immediately after a READ, WRITE or DELETEK to the primary text file. Any other instruction type may cause the INSERT to perform incorrectly.

INSERT – continued

Example:

```
MYFILE    IFILE
MYFILE2   AFILE
.
CUSTNO    FORM      "     32"
NAME      INIT      "THOMAS EDWARDS"
.
          OPEN      MYFILE,"MyISAM01"
          OPEN      MYFILE2,"MyAAM01"
.
          WRITE     MYFILE;CUSTNO,NAME
          INSERT    MYFILE2
.
          CLOSE     MYFILE
          CLOSE     MYFILE2
.
          STOP
```

DELETE

The DELETE instruction allows records to be physically deleted from the text file and for its key to be physically deleted from the ISAM file.

```
     DELETE   {afile}
```

The last AAM record accessed through an AAM READ instruction is the record to be deleted. If a successful AAM READ instruction was not previously performed, an I/O error occurs.

Both the entire text record, including logical End Of Record (EOR) mark and the key are deleted. If the text record has already been deleted, only the key is deleted.

Deleted record space on fixed record length files is automatically reused by subsequent WRITE instructions (unless the WEOF parameter was specified for the file).

A DELETE must be performed against each alternate or secondary, index file that requires the key to be removed.

When deleting from a physical text file having both an ISAM and an AAM key file, the AAM DELETE should be performed first since it requires that the text record be present for the key information to be deleted.

EXERCISE 21

1. Create an Aamdex for the customer file using the number and name fields.

2. Modify Exercise 20 to perform the following:

   - Additionally open the Aamdex file.
   - Add a Find command.
   - Find should prompt for a number and a name.
   - If a number is entered, an exact key should be constructed.
   - If a name is entered, a floating key should be constructed.
   - Use the constructed key to locate and display a matching record.
   - Ensure that the Next and Previous command still function as defined.
   - Optionally add commands to move key generically through the file
   - The Delete command should be modified to maintain all indexes and aamdexes.

CHAPTER FOURTEEN

PRINT OUTPUT

PRINT OUTPUT

The printer output instructions control the data flow to the printer.

Print output may be directly to an attached printer or to a spool file for later printing.

The PL/B instructions that perform print output are:

Verb	**Usage**
PRINT	Print information on the printer or to a print file.
RELEASE	Free a printer.
SPLOPEN	Open a printer spool file.
SPLCLOSE	Close a printer spool file

The formatting of data output by PRINT is handled in one of the following ways:

1. By the variable's data definition.
2. By using supported PRINT list controls.
3. By embedding ESCAPE sequences within the print instruction.

If a PRINT or PRTPAGE instruction is not terminated by a semi-colon (;), a Carriage Return (CR), and Line Feed (LF) is sent upon completion of the instruction.

If the instruction is terminated by a semi-colon, the print location remains one position beyond the last character printed.

PRINT

The PRINT instruction sends information to the default print device/file.

```
     PRINT    [{pfile}];{list}[;]
```

{pfile} is an optional PFILE variable that routes the output to an alternate device/file. The {pfile} must have been previously opened via the SPLOPEN command.

{list} contains any combination of variables, literals, or list control characters.

The optional semicolon inhibits the normal carriage return and line feed.

If a spool file/device is open, the PRINT output is directed to it. Otherwise, it is directed to the default print device/file name.

LITERALS

All characters within a pair of double quotes are sent to the default printer device/file exactly as they appear in the literal.

The first character in the literal is printed at the current print location.

The print location is incremented one position to the right for every character printed and remains one position to the right of the last character printed.

Control values (Ctrl) are sent as coded without translation or verification.

Example:

```
     PRINT     "Hello ";
     PRINT     "World"
```

VARIABLES

The first character printed is at the current print position.

The print position is bumped one position for each character printed. It remains one position to the right of the last character printed.

Unless modified by a list control, they are printed as follows:

Character String Variables

- The print begins with the first physical character and continues through Length Pointer (LP).
- Blanks are printed for each character between the Length Pointer and Physical Length (PL).
- If the Form Pointer (FP) is zero, blanks are printed for each character position to the Physical Length.
- If an item within the list is a character string Array and no specific array element has been designated, every array element is printed.

Numeric Variables

- The print begins with the first physical character and continues through the Physical Length.
- If an item within the list is a numeric array and no specific array element has been designated, every array element is printed.

LIST CONTROLS

List controls format the printing of the data. The following List Controls are supported:

Control	Meaning
*+	Blank suppression on
*-	Blank suppression off
*BLANKOFF	Terminate *BLANKON
*BLANKON	Blank-fill zero value numerics
*C	Carriage return
*F	Form feed
*JC	Justify center
*L	Line feed
*LL	Print through logical length
*{n}	Tab to specified column
*N	New line
*PL	Print through physical length
*RPTCHAR	Repeat character
*S0-*S9	User defined sequences
*TAB	Set horizontal position
*ZF	Zero fill
*ZFON	Zero fill on
*ZFOFF	Zero fill off

Example:

```
PRINT    *F,*N,*N,*55,"Sunbelt Computer Software":
         *90,DATE
```

RELEASE

The RELEASE instruction frees a printer device/file name that has been reserved for printing.

```
    RELEASE
```

The default printer device/file name that was locked by a non-spooled PRINT instruction is unlocked (or released) for access by other users.

If spooling is active, the standard Form Feed character (Hex 0C) is sent to the designated device/file name. The spooled device/file name is not unlocked by it.

PFILE

The PFILE instruction defines spool files used for alternate PRINT spooling.

```
PRINTFILE    PFILE
```

A PFILE must have been defined with a PFILE definition instruction before being referenced in a PRINT instruction.

A SPLOPEN instruction must have been performed using the given PFILE variable before performing any PRINT instructions using the PFILE variable.

Multiple PFILEs may be opened at one time allowing access to multiple printers.

SPLOPEN

The SPLOPEN instruction directs printed output to the disk or another specified device.

Spooled disk files are written in a standard operating system format and may be printed using any compatible list utilities.

```
     SPLOPEN  [{pfile}],{name}[,{options}]
```

{pfile} is a previously defined PFILE.

{name} is the name of the spool file/device name.

If {name} is a Null String, the default file name is used (DSPORT##.SPL for MS-DOS or PORT####.??? for Linux. The pound signs (#) are replaced with the user's port number and the question marks (?) are replaced by the default extension from the screen definition file.

The "Q" {option} designates the append mode for the file/device rather than its being overwritten.

Print spooling with no {pfile} remains active across program chaining until a SPLCLOSE is executed or the program returns control to the operating system.

Print spooling with a {pfile} that is not a Global variable does not remain active across program chaining.

SPLCLOSE

The SPLCLOSE instruction closes an existing spool file after updating its End Of File information in the directory.

```
    SPLCLOSE     [{pfile}]
```

A PRINT instruction that does not specify a {pfile} and is executed after a SPLCLOSE instruction directs the output to the default printer device/file name.

A PRINT instruction that specifies a {pfile} and is executed after a SPLCLOSE instruction for that {pfile} generates a Spool Error.

A SPLCLOSE instruction that is executed against a open spool file that is not open is ignored.

SPOOLING EXAMPLE

```
*
.Customer File
.
NAME      DIM       30
ADDRESS   DIM       30
CITYST    DIM       30
ZIPCODE   FORM      5
BALANCE   FORM      7.2
CUSTFILE  IFILE
*
.Work Variables
.
PRTFILE   PFILE
LINECNT   FORM      2    // Customers per page
*
.Ready the spool file
.
          SPLOPEN   PRTFILE,"REPORT.LST"
*
.Open the Customer File
.
          TRAP      NOFILE IO
          OPEN      CUSTFILE,"CUSTDATA"
          TRAPCLR   IO
          CALL      HEADING
*
.Read and Print
.
          LOOP
          READKS    IFILE;NAME,ADDRESS:
                    CITYST,ZIPCODE,BALANCE
          UNTIL     OVER
.
          PRINT     *N,*N,NAME,*N,ADDRESS,*N,CITY,*N:
                    ZIPCODE,*N,BALANCE
.
          DECR      COUNT
          CALL      HEADING IF ZERO
          REPEAT
```

SPOOLING EXAMPLE - CONTINUED

```
*
.Finish
.
          CLOSE
          SPLCLOSE  PFILE
.
          DISPLAY   *B,*HD,"Printing complete",*W;
          STOP
*
.Print the heading
.
HEADING
          PRINT     *F,*N,*N,*55,"Customer List",*N
          MOVE      "7",COUNT
          RETURN
*
.The customer file failed to open
.
NOFILE
          KEYIN     *B,*HD,"The customer file is ":
                    "not available. ",NAME
          STOP
```

EXERCISE 22

Using the previous exercise

1. Add a command to allow printing.
2. Spool the output to a user specified file.
3. Output the account number, name, and balance on a single line.
4. Provide a page heading with column headings.
5. Print the total the accounts balances.
6. Print the spooled report using the SUNLIST utility.

APPENDIX A

CHARACTER TRANSLATION TABLE

TRANSLATION TABLE

DEC	CTRL	OCTAL	HEX	PC	ASCII	EBCDIC	BINARY
0		000	00		NULL	NULL	0000 0000
1	(^A)	001	01	J	SOH	SOH	0000 0001
2	(^B)	002	02	Ä	STX	STX	0000 0010
3	(^C)	003	03	©	ETX	ETX	0000 0011
4	(^D)	004	04	¨	EOT	PF	0000 0100
5	(^E)	005	05	§	ENQ	HT	0000 0101
6	(^F)	006	06	ª	ACK	LC	0000 0110
7	(^G)	007	07	!	BEL	DEL	0000 0111
8	(^H)	010	08	Ï	BS		0000 1000
9	(^I)	011	09	"	HT	RLF	0000 1001
10	(^J)	012	0A	Ð	LF	SMM	0000 1010
11	(^K)	013	0B	¿	VT	VT	0000 1011
12	(^L)	014	0C	À	FF	FF	0000 1100
13	(^M)	015	0D	Ç	CR	CR	0000 1101
14	(^N)	016	0E	È	SO	SO	0000 1110
15	(^O)	017	0F	Á	SI	SI	0000 1111
16	(^P)	020	10	'	DLE	DLE	0001 0000
17	(^Q)	021	11	(	DC1/XON	DC1	0001 0001
18	(^R)	022	12	&	DC2	DC2	0001 0010
19	(^S)	023	13	Ë	DC3/XOFF	DC3/TM	0001 0011
20	(^T)	024	14	¶	DC4	RES	0001 0100
21	(^U)	025	15	§	NAK	NL	0001 0101
22	(^V)	026	16	É	SYN	BS	0001 0110
23	(^W)	027	17	Ì	ETB	IL	0001 0111
24	(^X)	030	18	-	CAN	CAN	0001 1000

TRANSLATION TABLE - continued

DEC	CTRL	OCTAL	HEX	PC	ASCII	EBCDIC	BINARY
25	(^Y)	031	19	¯	EM	EM	0001 1001
26	(^Z)	032	1A	®	SUB	CC	0001 1010
27	(^[)	033	1B	¬	ESC	CU1	0001 1011
28	(^\)	034	1C	Î	FS	IFS	0001 1100
29	(^])	035	1D	«	GS	IGS	0001 1101
30	(^^)	036	1E	)	RS	IRS	0001 1110
31	(^_)	037	1F	Ú	US	IUS	0001 1111
32		040	20	SP	SPACE	DS	0010 0000
33		041	21	!	!	SOS	0010 0001
34		042	22	"	"	FS	0010 0010
35		043	23	#	#		0010 0011
36		044	24	$	$	BYP	0010 0100
37		045	25	%	%	LF	0010 0101
38		046	26	&	&	ETB	0010 0110
39		047	27	'	'	ESC	0010 0111
40		050	28	(	(		0010 1000
41		051	29	)	)		0010 1001
42		052	2A	*	*	SM	0010 1010
43		053	2B	+	+	CU2	0010 1011
44		054	2C	,	,		0010 1100
45		055	2D	-	-	ENQ	0010 1101
46		056	2E	.	.	ACK	0010 1110
47		057	2F	/	/	BEL	0010 1111
48		060	30	0	0		0011 0000

TRANSLATION TABLE - continued

DEC	CTRL	OCTAL	HEX	PC	ASCII	EBCDIC	BINARY
49		061	31	1	1		0011 0001
50		062	32	2	2	SYN	0011 0010
51		063	33	3	3		0011 0011
52		064	34	4	4	PN	0011 0100
53		065	35	5	5	RS	0011 0101
54		066	36	6	6	UC	0011 0110
55		067	37	7	7	EOT	0011 0111
56		070	38	8	8		0011 1000
57		071	39	9	9		0011 1001
58		072	3A	:	:		0011 1010
59		073	3B	;	;	CU3	0011 1011
60		074	3C	<	<	DC4	0011 1100
61		075	3D	=	=	NAK	0011 1101
62		076	3E	>	>		0011 1110
63		077	3F	?	?	SUB	0011 1111
64		100	40	@	@	SPACE	0100 0000
65		101	41	A	A		0100 0001
66		102	42	B	B		0100 0010
67		103	43	C	C		0100 0011
68		104	44	D	D		0100 0100
69		105	45	E	E		0100 0101
70		106	46	F	F		0100 0110
71		107	47	G	G		0100 0111
72		110	48	H	H		0100 1000

TRANSLATION TABLE - continued

DEC	CTRL	OCTAL	HEX	PC	ASCII	EBCDIC	BINARY
73		111	49	I	I		0100 1001
74		112	4A	J	J		0100 1010
75		113	4B	K	K	.	0100 1011
76		114	4C	L	L	<	0100 1100
77		115	4D	M	M	(	0100 1101
78		116	4E	N	N	+	0100 1110
79		117	4F	O	O	|	0100 1111
80		120	50	P	P	&	0101 0000
81		121	51	Q	Q		0101 0001
82		122	52	R	R		0101 0010
83		123	53	S	S		0101 0011
84		124	54	T	T		0101 0100
85		125	55	U	U		0101 0101
86		126	56	V	V		0101 0110
87		127	57	W	W		0101 0111
88		130	58	X	X		0101 1000
89		131	59	Y	Y		0101 1001
90		132	5A	Z	Z	!	0101 1010
91		133	5B	[	[	$	0101 1011
92		134	5C	\	\	*	0101 1100
93		135	5D	]	]	)	0101 1101
94		136	5E	^	^	;	0101 1110
95		137	5F	_	_		0101 1111
96		140	60	`	`	-	0110 0000

TRANSLATION TABLE - continued

DEC	CTRL	OCTAL	HEX	PC	ASCII	EBCDIC	BINARY
97		141	61	a	a		0110 0001
98		142	62	b	b		0110 0010
99		143	63	c	c		0110 0011
100		144	64	d	d		0110 0100
101		145	65	e	e		0110 0101
102		146	66	f	f		0110 0110
103		147	67	g	g		0110 0111
104		150	68	h	h		0110 1000
105		151	69	i	i	|	0110 1001
106		152	6A	j	j	|	0110 1010
107		153	6B	k	k	,	0110 1011
108		154	6C	l	l	%	0110 1100
109		155	6D	m	m	_	0110 1101
110		156	6E	n	n	>	0110 1110
111		157	6F	o	o	?	0110 1111
112		160	70	p	p		0111 0000
113		161	71	q	q		0111 0001
114		162	72	r	r		0111 0010
115		163	73	s	s		0111 0011
116		164	74	t	t		0111 0100
117		165	75	u	u		0111 0101
118		166	76	v	v		0111 0110
119		167	77	w	w		0111 0111
120		170	78	x	x		0111 1000

TRANSLATION TABLE - continued

DEC	CTRL	OCTAL	HEX	PC	ASCII	EBCDIC	BINARY
121		171	79	y	y	/	0111 1001
122		172	7A	z	z	:	0111 1010
123		173	7B	{	{	#	0111 1011
124		174	7C	|	|	@	0111 1100
125		175	7D	}	}	'	0111 1101
126		176	7E	~	~	=	0111 1110
127		177	7F	DEL	DEL	"	0111 1111
128		200	80	Ç			1000 0000
129		201	81	ü		a	1000 0001
130		202	82	é		b	1000 0010
131		203	83	â		c	1000 0011
132		204	84	ä		d	1000 0100
133		205	85	à		e	1000 0101
134		206	86	å		f	1000 0110
135		207	87	ç		g	1000 0111
136		210	88	ê		h	1000 1000
137		211	89	ë		i	1000 1001
138		212	8A	è			1000 1010
139		213	8B	ï			1000 1011
140		214	8C	î			1000 1100
141		215	8D	ì			1000 1101
142		216	8E	Ä			1000 1110
143		217	8F	Å			1000 1111
144		220	90	É			1001 0000
145		221	91	æ		j	1001 0001
146		222	92	Æ		k	1001 0010

TRANSLATION TABLE - continued

DEC	CTRL	OCTAL	HEX	PC	ASCII	EBCDIC	BINARY
147		223	93	ô		l	1001 0011
148		224	94	ö		m	1001 0100
149		225	95	ò		n	1001 0101
150		226	96	û		o	1001 0110
151		227	97	ù		p	1001 0111
152		230	98	n		q	1001 1000
153		231	99	Ö		r	1001 1001
154		232	9A	Ü			1001 1010
155		233	9B	¢			1001 1011
156		234	9C	£			1001 1100
157		235	9D	¥			1001 1101
158		236	9E	&			1001 1110
159		237	9F	¦			1001 1111
160		240	A0	á			1010 0000
161		241	A1	í		~	1010 0001
162		242	A2	ó		s	1010 0010
163		243	A3	ú		t	1010 0011
164		244	A4	ñ		u	1010 0100
165		245	A5	Ñ		v	1010 0101
166		246	A6	ª		w	1010 0110
167		247	A7	º		x	1010 0111
168		250	A8	¿		y	1010 1000
169		251	A9	Í		z	1010 1001
170		252	AA	¬			1010 1010

TRANSLATION TABLE - continued

DEC	CTRL	OCTAL	HEX	PC	ASCII	EBCDIC	BINARY
171		253	AB	½			1010 1011
172		254	AC	¼			1010 1100
173		255	AD	¡			1010 1101
174		256	AE	«			1010 1110
175		257	AF	»			1010 1111
176		260	B0	¦			1011 0000
177		261	B1	¦			1011 0001
178		262	B2	¦			1011 0010
179		263	B3	¦			1011 0011
180		264	B4	¦			1011 0100
181		265	B5	¦			1011 0101
182		266	B6	¦			1011 0110
183		267	B7	+			1011 0111
184		270	B8	+			1011 1000
185		271	B9	¦			1011 1001
186		272	BA	¦			1011 1010
187		273	BB	+			1011 1011
188		274	BC	+			1011 1100
189		275	BD	+			1011 1101
190		276	BE	+			1011 1110
191		277	BF	+			1011 1111
192		300	C0	+			1100 0000
193		301	C1	-		A	1100 0001
194		302	C2	-		B	1100 0010

TRANSLATION TABLE - continued

DEC	CTRL	OCTAL	HEX	PC	ASCII	EBCDIC	BINARY
195		303	C3	+		C	1100 0011
196		304	C4	-		D	1100 0100
197		305	C5	+		E	1100 0101
198		306	C6	¦		F	1100 0110
199		307	C7	¦		G	1100 0111
200		310	C8	+		H	1100 1000
201		311	C9	+		I	1100 1001
202		312	CA	-			1100 1010
203		313	CB	-			1100 1011
204		314	CC	¦			1100 1100
205		315	CD	-			1100 1101
206		316	CE	+			1100 1110
207		317	CF	-			1100 1111
208		320	D0	-			1101 0000
209		321	D1	-		J	1101 0001
210		322	D2	-		K	1101 0010
211		323	D3	+		L	1101 0011
212		324	D4	+		M	1101 0100
213		325	D5	+		N	1101 0101
214		326	D6	+		O	1101 0110
215		327	D7	+		P	1101 0111
216		330	D8	+		Q	1101 1000
217		331	D9	+		R	1101 1001
218		332	DA	+			1101 1010

TRANSLATION TABLE - continued

DEC	CTRL	OCTAL	HEX	PC	ASCII	EBCDIC	BINARY
219		333	DB	¦			1101 1011
220		334	DC	_			1101 1100
221		335	DD	¦			1101 1101
222		336	DE	¦			1101 1110
223		337	DF	¯			1101 1111
224		340	E0	I			1110 0000
225		341	E1	ß			1110 0001
226		342	E2	,		S	1110 0010
227		343	E3	X		T	1110 0011
228		344	E4	;		U	1110 0100
229		345	E5	[		V	1110 0101
230		346	E6	T		W	1110 0110
231		347	E7	\		X	1110 0111
232		350	E8	>		Y	1110 1000
233		351	E9	1		Z	1110 1001
234		352	EA	A			1110 1010
235		353	EB	L			1110 1011
236		354	EC	¥			1110 1100
237		355	ED	f			1110 1101
238		356	EE	M			1110 1110
239		357	EF	Ç			1110 1111
240		360	F0	°		0	1111 0000
241		361	F1	±		1	1111 0001
242		362	F2	³		2	1111 0010

TRANSLATION TABLE - continued

DEC	CTRL	OCTAL	HEX	PC	ASCII	EBCDIC	BINARY
243		363	F3	£		3	1111 0011
244		364	F4	ó		4	1111 0100
245		365	F5	õ		5	1111 0101
246		366	F6	¸		6	1111 0110
247		367	F7	»		7	1111 0111
248		370	F8	0		8	1111 1000
249		371	F9	+		9	1111 1001
250		372	FA	,			1111 1010
251		373	FB	Ö			1111 1011
252		374	FC	'			1111 1100
253		375	FD	²			1111 1101
254		376	FE	#			1111 1110
255		377	FF				1111 1111

APPENDIX B

PL/B UTILITIES

PL/B UTILITIES

The PL/B Utilities allow manipulation of various types of files. These files include not only ASCII text files, but also Sunbelt's Indexed Sequential Access Method (ISAM) files, Associated Aamdex Method (AAM) files, object code files and others.

BLOKEDIT	Creates a new text file by merging data from other text files.
BUILD	Quick and easy way to create a small text file.
OBJMATCH	Compare two files byte for byte.
REFORMAT	Change the format of a text file.
SUNAAMDX	Creates an Associated Access Method file
SUNDUMP	Display and modify any type of file on disk.
SUNINDEX	Creates an Indexed Sequential Access Method file
SUNLIST	List a text file to the CRT, printer or to a print formatted file.
SUNSCAN	View a text file using the arrow keys and PgUp and PgDn keys.
SUNSORT	Rearrange the records within a text file into another sequence.
TXTMATCH	Comparison of two text files.

BLOKEDIT

The BLOKEDIT utility copies all or part(s) of one or more text files into a new text file.

Instructions may be entered from the keyboard or retrieved from a text file.

This utility is useful for rearranging source programs, extracting routines from various programs to create a new program or for rebuilding ASCII data files.

BLOKEDIT also provides the option to insert text from the keyboard.

The format for the BLOKEDIT command line is:

```
BLOKEDIT [{cmdfile}],{output} -[{opts}]
```

Instructions are entered to the BLOKEDIT utility while positioned on line 24 of the CRT or as retrieved from the {cmdfile}. Valid instructions are as follows:

Entry	Meaning
FILENAME	Search for specified input file.
n-m	Copy lines n through m from the input file and write these lines into the output file.
" (double quote)	Enter or exit text mode. Any lines entered between these are considered as input and are written to the output file. These lines start or terminate the text entry only if the "" is the first and only data on that line. Otherwise, it is considered an invalid command (if not in text entry mode) or is written to the output file.
* (asterisk)	End this BLOKEDIT. Write the EOF on the {output} file and return to the operating system.
. (period)	Comment line in command file.

BLOKEDIT - continued

If {cmdfile} is specified, it must be an ASCII text format file that contains the instructions for this particular BLOKEDIT procedure.

If {cmdfile} is not specified, all instructions for this particular BLOKEDIT procedure must be entered from the terminal keyboard.

The assumed file type (extension) on both files is TXT.

If {output} already exists and {option} was not specified, the following prompt is displayed:

```
OUTPUT FILE ALREADY EXISTS.  OVERWRITE IT (Y/N)?
```

Any response other than 'Y' aborts the BLOKEDIT utility.

A search of all search paths is performed in an attempt to locate the specified files. If {output} is not found, it is created in the current directory.

{option} is not required and if specified must be as follows:

Option	Meaning
C	Compressed output file.
N	Uncompressed output file (default).
O	Overwrite {output} if it already exists.
?	Display the BLOKEDIT help information.

BUILD

The BUILD utility creates ASCII text files or null files. The command line format for the BUILD utility is:

```
BUILD {output} -[{endchar}]
```

If {output} already exists, it is overwritten and no error message generated.

If no path specification is given for {output}, a path search is performed to determine if it already exists.

The assumed file type on {output} is TXT.

{endchar} signals the BUILD utility that this particular BUILD is complete. The character specified must be the one and only character on that line to terminate the command.

If {endchar} is not specified, a null entry terminates the BUILD.

OBJMATCH

The OBJMATCH utility compares two files and displays any differences between them.

```
OBJMATCH {file1},{file2}
```

{file1} and {file2} may include directory information.

Every byte of each file is compared.

If a mismatch is found, both the hexadecimal and ASCII interpretations of the data from both files is shown.

REFORMAT

The REFORMAT utility converts the internal disk format of text files. It also recovers lost disk space resulting from records deleted via AAM or ISAM. The command line for REFORMAT is:

```
REFORMAT [{input}],{output} [-{options}]
```

If {input} is not specified, the user is prompted for the input file(s). When all files have been entered, terminate the list by entering an asterisk (*).

If an extension is not given for either {input} or {output}, TXT is assumed.

REFORMAT first attempts to locate the {input} file(s) in the current path. If not found, a search of all search paths occurs.

{output} is overwritten if it exists.

The {input} and {output} files must be different files.

REFORMAT – continued

{options} are as follows:

Option	**Meaning**
B{n}	Block the records in {output} {n} records per sector. The sector size is assumed 256 bytes.
C	The {output} file will be space compressed using the standard tab character (Hex 09). This is the default option.
E={xx}	Set the output End of Record Type to be {xx} where {xx} is one of CR, LF, CRLF, or LFCR.
K	Kill the input file(s) after completion of the REFORMAT.
L{n}	{output} records are to be written {n} bytes long.
P	Pack as many records into each 256 byte sector without crossing sector boundaries. The number of records in each sector may then be variable.
R	The {output} file is record compressed, but not space compressed.
S	Segment records longer than the specified length. Valid only if the 'L' option is also specified.
T	Truncate records longer than the specified length. Valid only if the 'L' option is also specified.
Z	Display the run time statistics including records in and out and start and stop times.
?	Display the REFORMAT help screen.

SUNAAMDX

The SUNAAMDX utility creates an Associated Access Method (AAM) key file using the specified key positions in the corresponding data file.

```
SUNAAMDX  {txt}[,{aam}[,{Lnnnn}]] -[{opts}]{keyspecs}
SUNAAMDX  {aam2 } -{I | R}
```

If no extension is specified for {txt}, TXT is assumed.

If {aam} is not specified, the file name in {txt} is used with the extension AAM. If {aam} is specified without an extension, AAM is used.

If neither {txt} nor {aam} are located in the current directory, the directories specified in the PLB_PATH environment variable are searched for the existence of both {txt} and {aam} files.

{Lnnnn} specifies the longest record length. This parameter is required when creating or indexing a null file or when indexing a variable record length or space compressed file and the first record is not the longest record in the file.

If {Lnnnn} is not specified on the command line, the record length is determined by the first record in the data file. If the input file is of fixed length records, all records in the file are assumed that length during the SUNAAMDX procedure.

SUNAAMDX – continued

{opts} supports the following options:

Option	Meaning
D={c}	The ASCII character specified is the wildcard or don't care character for READ operations ('?' is the default).
I	Display the command line that constructed the AAM file.
I{ininame}	When specified, the utility will first look for any operational keywords in {inifile}. If the keyword is not found, the UET is searched. The filename specified by {ininame} must be a fully qualified name. If a full path is not specified, the filename must exist in the current working directory.
N	Disable the record counter display.
P{nn}{=|#}'xxx'	The primary record selection criteria for partial indexing - 'xxx' may be up to eight (8) characters long.
Q	Quit without creating the output file if the input file does not exist.
R	Rebuild the index using stored command line.
S	Allow space-compressed records in the data file.
U	Treats Upper/Lower case characters equally in all READ operations.
V	Allow variable length records.
W	Write all new records at the end of the {txt} file.
Y	Do not place {txt} drive information into AAM header.
Z	Do not place {txt} drive or path information into AAM header.

SUNAAMDX – continued

The wildcard or don't care character defaults to a question mark but may be changed by using the D option.

P{nn}{=|#}’xxx’ is the primary record select option. It allows only records meeting the specified criteria to be indexed. The criteria match, up to eight bytes, starts at column 'nn' and the match is based on the data being either equal to (=) or not equal to (#), the specified characters (‘xxx’). Records meeting the criteria are indexed while the others are ignored.

Records in the data file need not be in fixed and/or non-space compressed format. If the 'S' option is given, the records may be both variable length and space compressed. If the 'S' option is given, the 'W' option is assumed.

Records written to the data file may be variable length and not be space compressed by using the 'V' option. As with the 'S' option, the 'W' option is assumed.

The 'W' option causes new records added to the file to be written at the end of file at all times. Deleted record space is not re-used and this space is not recovered until the file is reformatted.

It is possible to have all lower case letters treated as upper case during the search by specifying the 'U' option. This causes all lower case letters to be indexed as if they were upper case and treated as such during AAM I/O.

Up to 95 key specifications may be given and they may be continued to subsequent lines by terminating the line with a colon (:). Additional key specifications must then be given.

Individual key specifications must give a starting and ending record position, separated by a hyphen (-) unless a one byte key and must be separated by commas.

SUNAAMDX – continued

An 'X' immediately preceding an AAM key range (i.e., X1-10 or X10-20) indicates an excluded field. These fields may be used in conjunction with one or more valid AAM keys to retrieve records, but are not hashed into the AAM file. Their key position is relative to the other valid or excluded keys given in the keyspecs.

The data file is not re-written to discard any deleted record space. Only a new AAM file is created.

The order in which your fields are specified is critical during the READ operation. The first key specification is field number 1, the second is field number 2, etc.

Sub-fields are allowed. A key field may be wholly or partially contained within another field. Field 1 may be positions 1-10, while field 2 may be positions 3-5 and field 3 may be 7-14.

The information about the index file may be retrieved by specifying the index file name followed by the 'I' option. The command that was required to generate the file is displayed.

The index file may be re-indexed without knowing any of the key specs by specifying the index file name followed by the 'R' option.

SUNINDEX

The SUNINDEX utility creates an ISAM key file using the specified key positions in the associated data file. SUNIDXNT is a Windows NT or Windows 95 Version of SUNINDEX. SUNINDEX uses one of the following command lines:

```
SUNINDEX [{-tnnnn}] {txt}[,{isi}[,{Lnnnn}[,{work}]]]
-[{opts}]{keyspecs}
```

```
SUNINDEX {isi2} -[I | R]
```

If no extension is specified for {txt}, the extension TXT is assumed.

If the {txt} file does not exist, it is created if the {Lnnnn} specification is given.

If {isi} is not specified, the file name in {txt} is used with the extension ISI. If {isi} is specified without an extension, ISI is used.

If neither {txt} nor {isi} are located in the current directory, the directories specified in the PLB_PATH environment variable are searched for the existence of both {txt} and {isi} files.

{Lnnnn} specifies the longest record length. A maximum record length of 32k bytes is supported. This parameter is required when creating or indexing a null file or when indexing a variable record length or space compressed file and the first record is not the longest record in the file.

If {Lnnnn} is not specified on the command line, the record length is determined by the first record in the data file. If the input file is of fixed length records, all records in the file are assumed that length during the SUNINDEX procedure.

SUNINDEX – continued

In addition to the actual key specifications, {keyspecs} supports the following options:

Option	Meaning
D	Allow duplicate keys in the ISAM key file (default).
E	Create a file containing any duplicate keys found during indexing.
F={nn}	Records are fixed length binary with NO end of record bytes.
G={nn}	Records are fixed length binary with normal end of record bytes.
N	Do not allow duplicate keys in the ISAM key file. If any duplicate keys are found, the output file is deleted. Use the 'E' option to create an error file containing duplicate key information.
NO	The index file is created ignoring duplicate keys and not adding them into the file. For PLB OPEN operations, the IFILE is tagged as a NODUP file.

SUNINDEX – continued

Option	Meaning
Q	Quit without creating the output file if the input file does not exist.
S	Allow space-compressed records in the data file. The 'V' and 'W' options are assumed.
U	Convert lower case characters to upper case for the ISAM key file.
V	Allow variable length records. If the 'S' option is not given, these records cannot be space compressed. The 'W' option is assumed.
W	Write all new records at the end of the file. Assumed if the 'S' or 'V' options are specified.
X	Disable record counter display.
Y	Do not place {txt} drive information into ISI header.
Z	Do not place drive or path information into ISI header.

SUNINDEX – continued

Records in the data file need not be in fixed and/or non-space compressed format. If the 'S' option is given, the records may be both variable length and space compressed. If the 'S' option is given, the 'W' option is assumed.

Records written to the data file may be variable length and not be space compressed. The 'V' option is used if this is the case. As with the 'S' option, the 'W' option is assumed.

The 'W' option causes new records added to the file to be written at the end of file at all times. Deleted record space is not re-used and this space is not recovered until the file is reformatted.

It is possible to have all lower case letters converted to upper case for the ISAM key file by specifying the 'U' option as the first item in {keyspecs}. This does not affect the data in {txt}, only the key information that is written to the {isi} file.

Each part of key specification must give a starting and ending record position, separated by a hyphen (-) unless a one byte key and must be separated by commas.

The sum of the key specifications cannot exceed 99 positions.

The data file is NOT re-written to discard any deleted record space. Only a new ISAM key file is created.

The information about the ISAM file may be retrieved by specifying the ISAM file name followed by the 'i' option. The command that was required to generate the file is displayed.

The ISAM file may be re-indexed without knowing any of the key specs by specifying the ISAM file name followed by the 'r' option.

SUNLIST

The SUNLIST utility outputs the contents of a text file on the screen, the system printer, or to a print image disk file. The command line syntax for SUNLIST is:

```
SUNLIST {input}[,{start}[,{spool}]][-{options}
```

If a drive specification is not given on {input}, the current path is searched followed by all search paths.

If a file type is not given, it defaults to TXT with the following exceptions:

Option	Defaults to
A	AAM
F	LST
G	LST
I	ISI

Records may be of variable length and space compressed.

SUNLIST – continued

The allowable parameters for {start} are as follows:

Parameter	Indicates to start listing at ...
L{nnn}	logical record number {nnn} within the text file. This option works with all types of files supported.
R{nnn}	physical sector number {nnn} within the text file. A length of 256 bytes is used for the sector length. This only works with print files or standard text files.
P{nnn}	page number {nnn} within the print format file. This option is only applicable to print files.

SUNLIST – continued

Allowable {options} are as follows:

Option	Meaning
A	List via AAM file. A key specification must be given and enclosed within single quotes or no records are displayed. Up to five (5) AAM keys may be given and each must be preceded by the 'A'. Under Linux, the single quotes must each be preceded by a \.
B{prog}	Chain to program {prog} at the conclusion of the list program.
C{n}	Print {n} copies. Default is one.
D	Display on terminal (default option). When displaying a file on the terminal, the F1 key pauses the display and then continues it when it has been pressed a second time. The F2 or ESCAPE key terminates the display of the file.
E{x}	Replace {x} with an ESCAPE (Hex 1B) when printing the data file. This allows for control characters to be embedded within the data file for printer control. This option is only checked if the 'F' option is specified and is valid for all output devices except the screen.
F	List a print formatted file.
G	List an ANSI print formatted file.

SUNLIST – continued

Option	Meaning
H{n}	Pause for {n} number of seconds after the list is finished.
I	List via ISI file. All records are listed unless a beginning key or line number is entered. If a beginning key is specified, it must immediately follow the 'I' option and must be enclosed in single quotes. Under Linux, the single quotes must each be preceded by a \.
J	Force a leading form feed before printing the file.
K	Delete the file after completion. This option is not valid with ISI and AAM files.
L	List on the system printer.
N{n}	Set the number of lines per page. This determines the number of lines written before a page advance. This option is not valid with print format files.
P	Output to a print format file.
Q	Append to an existing print format file. If the file is not found, it is created.
R	List by page in reverse order. Read the file from end of file forward (useful for laser printers).
S{n}	Indicates the starting byte location within each logical record. This then becomes the first character printed or displayed.
T{n}	Tab to column {n} before printing each line.
V	Suppress final form feed.
W	Wrap option. List the entire record, even if it exceeds the output medium length.
X	List without line numbers.
Z{n}	Execute a Sunlist command upon completion of the list.

SUNSCAN

The SUNSCAN utility displays any part of a file using the arrow keys and page controls. SUNSCAN may also search for strings in the file and display a screen of information following the search string.

```
SUNSCAN {filename}[-{opt}]
```

The Left, Right, Up and Down arrow keys move the display in the appropriate direction. PgUp and PgDn display the next or previous screen. Home and End display either the top or bottom line while still maintaining the current column position.

The ESCAPE terminates the program.

{opt} supports the following option:

Option	Meaning
W{nn}	specifies the line width. If not specified, the line width is 250 bytes. It may be changed to any value between one and 65535.
S{"string"}	specifies an initial search string.

The 'S' key initiates a search. A window is opened allowing entry of the search string. A status line is displayed on the first line of the screen giving information concerning the progress of the search and the location in the file where the information was found when the search succeeds.

SUNSORT

The SUNSORT utility sequences a data file in a particular order. The command line syntax for SUNSORT is as follows:

```
SUNSORT [-L{log}] [{-tnnnn}]{input}[,{output}
        [,{tmppath}[,{sortseq}]]] -[{opts},]{keys}
```

If a path specification is not given on {input}, the current path is searched followed by all search paths.

When no file extension is given for {input} or {output}, the default text extension as specified in the screen definition file is used.

If neither {input} nor {output} are located in the current directory, the directories specified in the PLB_PATH environment variable are searched for the existence of both {input} and {output} files.

If {output} is not specified, the file is sorted to a temporary file and when the sort is finished the temporary file is renamed to the original file.

If a path specification is not given on {output}, the file is placed in the current path.

If the log file option is specified, all program messages are written to {log}.

SUNSORT – continued

{opts} refers to special options that affect the output of the sort.

Option	Meaning
B	Treat any space compression characters as binary characters and do not expand as a compression count.
F={nn}	Records are fixed length binary with NO end of record bytes.
G={nn}	Records are fixed length binary with normal end of record bytes.
K	Key tag file output. The output of a key tag sort is a file with record pointers to where the sorted records are located in the input file. The file is in a binary format with each pointer taking up 4 bytes. There are no carriage returns or line feeds in the file. The file may be read with PL/B by reading an INTEGER 4 field followed by a semicolon to keep the record position at the current location.
S{nn}{=|#}{c}	Selective output. If specified, only the records that match the specification are included in the output file. Since only a single character may be specified for this option, multiple S options select on multiple record positions. The {nn} specifies the record starting position, the {=|#} specifies either an equal or not equal condition test, and the {c} is the character to compare.

SUNSORT – continued

{keys} refers to the actual key information used in the sequencing of the file. Any key specification may be preceded by one of the following options. The options affect every key specification that follow until an alternate option is given.

Letter	**Meaning**
A	(default) All data within this keyspec range is sorted in ascending order.
D	(optional) All data within this keyspec range is sorted in descending order.
L	(default) All lower case data within this keyspec range is treated as lower case.
U	(optional) All lower case data within this keyspec range is treated as upper case.
N	The sort key is sorted as a signed numeric. When the minus sign character identifies a negative value, the sort output is in reverse order for negative numeric keys.

TXTMATCH

The TXTMATCH utility compares two text files for any inconsistencies between them. When a discrepancy is encountered, the comparison stops for your review. Each file is displayed separately on one-half of the screen display.

```
TXTMATCH {infile1},{infile2}[,{outfile}] -[{options}]
```

If the second file specification contains only a path, the input file name is assumed.

The program options control the handling of the log file:

Option	Meaning
A	Create the log file
Q	Append to the log file

When two files have been specified, the display of the files may be advanced individually or concurrently by use of the function keys.

Key	Meaning
F1	Read the first file and advance the display accordingly.
F2	Read the second file and advance the display accordingly.
F3	Read both files and advance the display accordingly.

TXTMATCH – continued

When three files have been specified, the display of the files and the output to the third file may be advanced and controlled by the use of the function keys. When the corresponding records in the first two files match, the record is automatically output to the third file. Only when the files do not match do the following function keys become active.

Key	Meaning
F1	Skip record in file 1.
F2	Skip record in file 2.
F3	Skip records in both file1 and file2.
F5	Write current record in file 1 to file 3 and read another record from file 1.
F6	Write current record in file 2 to file 3 and read another record from file 2.
F7	Write current record in file 1 to file 3 and read another record from both files.
F8	Write current record in file 2 to file 3 and read another record from both files.