Commodore Edition

kyan

PASCAL

A PROGRAMMING LANGUAGE

Tutorial Manual

kyan

PASCAL

A PROGRAMMING LANGUAGE

Tutorial Manual

1850 Union Street, Suite 183
San Francisco, CA 94123

LICENSE AGREEMENT

a. By purchasing Kyan Pascal you are granted a personal, non-transferable and non-exclusive license to use the documentation and Program on a single CPU under the terms stated in this Agreement. Title and ownership of the Program remain with Kyan Software;

b. you, your employee and/or agents are required to protect the confidentiality of the Program and documentation. You may not distribute or otherwise make the Program or documentation available to any third party;

c. you may not assign, sublicense or transfer this license and may not copy or reproduce the Program or documentation for any purpose, except you may copy the Program into machine readable or printed form for backup purposes in support of your use of the Program.

d. Any portion of this Program merged into or used in conjunction with another program will continue to be the property of Kyan Software. Kyan Software hereby grants you a non-exclusive license to merge or use portions of the Program in conjunction with your own programs <u>provided</u> <u>that</u>: you acknowledge Kyan Software's copyright and ownership of these portions in a prominent location on both the magnetic media and written documentation for your software.

e. You acknowledge that Kyan Software has a valuable proprietary interest in the Program and documentation and you are receiving only a LIMITED LICENSE TO USE the Program and related documentation and that Kyan Software retains title to the Program and documentation.

YOU MAY NOT USE, COPY, MODIFY, OR TRANSFER THE PROGRAM, OR DOCUMENTATION, OR ANY COPY, MODIFICATION OR MERGED PORTION, IN WHOLE OR IN PART, EXCEPT AS EXPRESSLY PROVIDED FOR IN THIS LICENSE. IF YOU DO, YOUR LICENSE IS AUTOMATICALLY TERMINATED.

TERM: This license is effective until terminated. You may terminate it at any time by destroying the Program and documentation with all copies, modifications and merged portions in any form. It will also terminate upon conditions set forth elsewhere in this agreement or if you fail to comply with any term or condition of this Agreement. You agree upon such termination to destroy the Program and documentation together with all copies, modifications and merged portions in any form.

CONTENTS

kyan

Dear Friend:

Thank you for purchasing Kyan Pascal for the Commodore 64/128. The following manual is intended as a reference for both STANDARD Kyan Pascal and ADVANCED Kyan Pascal. Those chapters which apply only to STANDARD Kyan Pascal are marked (STANDARD); those chapters which apply only to ADVANCED Kyan Pascal are marked (ADVANCED). You will find that most chapters apply to both versions.

<u>STANDARD Kyan Pascal</u>

STANDARD Kyan Pascal offers a very user-friendly programming environment which is perfect for learning the Pascal language. It features a P-code compiler, fully-resident software, and other useful functions which enable you to quickly write, compile and debug Pascal programs. It is widely used in schools and universities to teach Pascal to beginning students.

<u>ADVANCED Kyan Pascal</u>

ADVANCED Kyan Pascal is actually two products in one. The disk contains STANDARD Kyan Pascal plus a second machine code compiler. It includes all the sophisticated Pascal functions needed to develop professional quality programs. The built-in assembler, linking/chaining functions, and other advanced features enable you to write very large Pascal programs which will run at the maximum speed possible on the Commodore 6502 microprocessor.

If you purchased the ADVANCED version and are new to Pascal, we suggest that you concentrate first on STANDARD Kyan Pascal. Then, as your proficiency in the language grows, you can move up to the powerful capabilities of the ADVANCED version. If you are an old hand at Pascal, you can move directly to the ADVANCED version. We believe you will find the Kyan software has the ability to satisfy all of your programming needs.

If you have questions or suggestions for improvements in the software, please let us know. We are always trying to improve our products and welcome input from our customers.

Thanks again for selecting Kyan Pascal.

Sincerely,

Thomas E. Eckmann

Thomas E. Eckmann
President

TEE/sk

kyan software 1850 union street #183 san francisco, california 94123 (415) 775-2923

INTRODUCTION

The history of PASCAL goes back to the late 1960s, when Niklaus Wirth, a professor of computer science, decided that a new approach -- a new language -- was needed for teaching programming. Wirth introduced PASCAL as a formal language in 1971.

The two principal aims of PASCAL are "..to make available a language suitable to teach programming as a systematic discipline based on certain fundamental concepts clearly and naturally reflected by the language.." and "to develop implementations of this language which are both reliable and efficient on presently available computers" (Jensen and Wirth, "Pascal User Manual and Report").

PASCAL has become a widely used language for both elementary and advanced programming. Its popularity derives from the clarity of programs written in it and the efficency with which it can be implemented within the computer.

Kyan PASCAL is especially efficient in this regard, because the run time code and the compiler itself are written in assembly language, the language of the microprocessor integrated circuit.

PASCAL is a self-documenting and self-structuring language. Top-down programming and modulization are natural outgrowths of its features. These features include, among others, user-defined functions and procedures of which modules are built.

Separation of the declaration section from the program body also enforces good programming technique. All the information on constants, types of variables, and names of variables and constants appears in a single section rather than being spread throughout the program.

Kyan PASCAL provides features that help the programmer to find the syntax errors that account for over 90% of the errors in programming. Over 30 error messages for syntax are in the compiler. These not only tell the programmer what types of errors were made but also on which lines the errors occurred.

One final reason for using PASCAL is its portability. PASCAL, one of today's most popular languages, is implemented on nearly every computer on the market. Kyan PASCAL is compatible with Standard PASCAL. Programs and program modules written in Kyan PASCAL will run on a multitude of computers: a programmer can develop software on a home computer, transport it to many other machines, and run the programs immediately.

References

Jensen, Kathleen, and Wirth, Niklaus: "Pascal User Manual and Report," Springer-Verlag, Berlin, 1974

Kaufman, Elliott B.: "Pascal, A Problem Solving Approach," Addison-Wesley Inc., Reading, Mass., 1982

Zaks, Rodney: "Introduction to PASCAL," Sybex Inc., Berkeley, California, 1981

EDITOR AND COMPILER INSTRUCTIONS
STANDARD KYAN PASCAL

To transfer Standard Kyan PASCAL from the disk to memory, type:

```
LOAD "PASCAL",8
```

followed by <RETURN>. Striking the <RETURN> key tells the computer that you have completed your entry. Unless instructed otherwise, you should always press <RETURN> when entering information into the computer.

Your Commodore 64* will search for the program (SEARCHING FOR PASCAL), then indicate LOADING when Kyan PASCAL is found. Completion of loading will take several minutes, at which point READY will be displayed:

```
READY
LOAD "PASCAL",8

SEARCHING FOR PASCAL
LOADING
READY
```

Type RUN to complete the loading task. The version number of your Kyan PASCAL will appear on the screen followed by instructions to get the help menu, and finally the prompt symbol (>).

The prompt symbol (>) is a signal from the program, in this case from Kyan PASCAL, that you are to enter something.

```
READY
LOAD "PASCAL",8

SEARCHING FOR PASCAL
LOADING
READY
RUN

KYAN PASCAL VERSION 2.0
COPYRIGHT 1984 BY KYAN SOFTWARE
ALL RIGHTS RESERVED
TYPE "R HELP" TO GET A HELP MENU

>
```

You must remove your Kyan PASCAL disk before creating or saving programs. In general, the time to remove the disk is immediately after loading. You must then insert a blank disk on which your programs will be saved.

A sample program, called "PAINT," is included on the Kyan PASCAL disk. If this is your first time using Kyan PASCAL, leave the disk in the drive so that you can use the sample program as you familiarize yourself with the editor commands.

* Trademark of Commodore Inc.

Type R HELP and <RETURN>. A menu of commands that you will use will be displayed:

```
BUILD    B [FILE]
COMPILE  C [FILE]
EDIT     E [FILE]
FILE     F [FILE]
PRINT    P [FILE]
RUN      R [FILE]
SCRATCH  -S0:FILE
RENAME   -R0:NEWFILENAME=OLDFILENAME
COPY     -C0:NEWFILE=FILE
MERGE    -C0:NEWFILE=FILE1,FILE2,FILE3

>
```

Always press <RETURN> after entering a command.

The Default File

To make PAINT the default file, enter the following:

```
>F PAINT
```

PAINT is now the default file because it was the last file name entered with the F command. Whenever B, C, E, P or R is entered without a file name, the operation will be performed on PAINT.

The computer will wait for you to choose the next step.

The File Directory

To list the files on your disk, enter D. Among the files on your Kyan PASCAL disk are:

```
HELP      HELP$
```

HELP is an example of a PASCAL source code file similar to one that you might create. HELP$ was created by the compiler (discussed later) and is called an object code file. The dollar sign was added by the compiler as a means of distinguishing between the two files. However, a file without a dollar sign is not necessarily source code.

Editing

Programs are stored in separate files. Each program added to the disk must have its own file name consisting of 8 or fewer letters and/or characters. Spaces are not permitted within a file name.

To enter a new program or change an existing one, enter E or E XXXXXXXX. If XXXXXXXX is the name of an existing file, the program will appear on the screen; otherwise, the screen will be blank.

The entering or changing of a program is called editing. Suppose you wish to edit PAINT.

```
>F PAINT

>E
```

As soon as <RETURN> is pressed, the program will appear:

```
PROGRAM PAINT;

TYPE A = ARRAY[1..25,1..40] OF CHAR;
VAR
 V,C : ^A;
 I,J : INTEGER;

BEGIN
  ASSIGN(V,1024);
  ASSIGN(C,-10240);
  FOR I := 2 TO 17 DO
    FOR J := 1 TO 40 DO BEGIN
      V^[I,J] := CHR(160);
      C^[I,J] := CHR(I)
  END
END.



F1 DONE    F3 GOLD    F5 TEXT>    F7 PAGE>
F2 SAVE    F4 UNDO    F6 <TEXT    F8 <PAGE
```

Kyan PASCAL includes an easily learned, full-screen, insert mode editor. Anywhere you move the cursor, a letter, a number, a space, or even a new line break may be added. This greatly facilitates editing your program. (Holding down the up-down <CRSR> key scrolls the screen.)

<INST/DEL> operates (as in Commodore BASIC) by deleting the character, space, or line break just to the left of the cursor. In addition, the combination of <SHIFT> and <INST/DEL> deletes coincident to the cursor.

The choice of function key F3, F5, F6, F7, or F8 allows you to go to any line in the program. F5 takes you to the end of your program; F6 takes you to the beginning of your program. F7 and F8 move you forward or backward 20 lines, respectively. When F3 is pressed, the following instruction appears:

GO TO:
G LINE-NUMBER

SUBSTITUTE:
S/OLD-STRING/NEW-STRING[/?]

FIND FORWARDS:
F [/STRING]

FIND BACKWARDS:
B [/STRING]

The go-to, substitute and find commands are two-stage commands. For example, to go to any line in the program, press F3. When the above instructions appear, enter G followed by the line number you want to go to. G followed by zero will take you to the last line.

To find the occurrence of any string in the program, press F3, then either F or B followed the the string which is to be found. For example, if "look out" is to be found, enter:

F3
F/look out

The cursor will go to the first occurrence of "look out" ahead of the present cursor position. To find the second occurrence, enter F3 F without the string. This process can be repeated until all occurrences of "look out" have been found. If you enter F3 B, the occurrence of the string in back of the present cursor position will be found.

Maximum string size is 40 characters.

To substitute a new string for any string in the program, enter F3 followed by:

S /old string/new string

The old string consists of the words or lines the way they are before the substitution. The new string consists of the new words or lines that will take their place. To replace the string "first" with the string "# 1," enter:

F3
S/first/# 1

The slashes are called delimiters. One delimiter is used before the old string and one delimiter is used before the new string. Any character may be used for a delimiter, but you must choose one that is not used in the strings. Otherwise, the editor will not find the correct end or beginning of the string.

Add a third delimiter and a "?" after the new string if you want the editor to stop before making a substitution:

F3
S/first/# 1/?

If you answer Y when the string if found, the substitution will be made. If you answer N, the substitution will not be made. In either case, the editor will automatically go on to find the next occurrence of the old string. You may answer Q at any time to quit the search and replace process.

Editing PAINT, an Example

Use the cursor <CRSR> keys to go to the end of the line "FOR J := 1 TO 40 DO BEGIN." Then add a period to the end of the line.

You have now entered a change in the program PAINT and are ready to save the new version.

Saving a File

Because the Kyan PASCAL disk is write-protected, insert a formatted blank disk before attempting to save the edited version of PAINT.

To save your program, you can press the function key F1 or F2. F1 differs from from F2 in that F2 returns you to the editor immediately after saving the program, whereas F1 requires that you enter E to return to the editor.

If you press F4 (<SHIFT> F3), the changes produced by editing will not be saved, and the previous version of the program will remain unaltered. F4 is called the undo key.

Now press F1 to save PAINT.

Compiling a File

Before a PASCAL program can be run, it must be entered into the computer's memory, compiled and translated into machine language.

To compile the program, enter C. If there are errors, they will be counted and listed, and the program cannot be run. If there are no errors, the error messages will not be displayed.

Enter C to compile the edited version of PAINT. Because of the change you have made, there will be an error.

```
>C

0010      FOR J := 1 TO 40 DO BEGIN.
                                   1
(1) ";" OR "END" EXPECTED

0001 ERRORS

>
```

The line with the error is displayed with its line number. The "1" underneath the line indicates where the error occurred. A description of the error type is also displayed.

Sometimes, a single error in a program (such as a missing semicolon after a VAR declaration) will generate multiple errors following it.

There may be too many errors to display at one time on the screen. To see them all, enter the edit mode, but with the file ERROR$:

```
>E ERROR$
```

You may now use the <CRSR> keys to examine each of the errors. The errors can also be printed by entering:

```
>P ERROR$
```

Go back to the edit mode and correct the error you introduced. (PAINT is still the default file.) Try compilation again. Now there should be no error message. The program is ready to run.

Running a File

Enter R after compilation. The default program will run and perform the tasks it was designed to do. (A program with no errors on compilation may still be faulty. For example, programming the area of a circle to be "r*r" instead of "3.14*r*r" is an error the compiler will not detect.)

Building a Stand-Alone File

A stand-alone file is a compiled PASCAL program that can be run without using the Kyan PASCAL disk. To build the stand-alone file, enter B after you have compiled your program:

```
>B  (PAINT is still the default file.)
```

Look at the file directory; it should now include PAINT, PAINT$, and PAINT%. PAINT is the source code, PAINT$ is the compiled file, and PAINT% is the stand-alone file. You may now make copies of PAINT% and run these using the Commodore 64 without Kyan PASCAL.

To run the stand-alone file from Commodore BASIC, type:

```
LOAD "FILENAME%",8
RUN
```

In the last example, you would type LOAD "PAINT%",8.

Printing a Program

After editing, enter P or P XXXXXXXX, where XXXXXXXX is the name of the file. If you have a printer, the default program or other program specified will be printed.

File Management

The Commodore 64 provides a set of file management commands that may be accessed without leaving the Kyan PASCAL environment. These commands may be used at any time except when a program is running or being edited.

To format a disk enter:

```
-NO:Name,ID
```

"Name,ID" is the name you give to your disk (Kyan PASCAL does not use this identifier), where ID is restricted to two characters.

To clear the disk of all files, enter:

```
-NO:Name
```

To delete a file called "First" and also to delete its compiled version, "First$," enter:

```
-SO:First
```

Sometimes a new program will be a variation of or an extension of an existing one. The first step in producing the new program would be to make a copy of the old one. For example, if "Factor" is to be a copy of the file named "First," enter:

```
-CO:Factor=First
```

To combine the files "Factor" and "Max" and call them "Makeup," enter:

```
-CO:Makeup=Max,Factor
```

To change the name of a file named "Smith" to "Jones," enter:

```
-RO:Jones=Smith
```

The validate command enables you to clean up your disk. It increases the room on the disk by consolidating unused spaces and deleting improperly closed files. To validate your disk, enter:

```
-V
```

If the disk drive loses its place, or for some other reason you wish to restart, i.e., to initialize it, enter:

```
-I
```

Editor Specifications

All names (identifiers, file names, function names, program names, etc.) are limited to 8 characters. The editor will accept more than 8 characters, but any characters beyond 8 will be ignored. Therefore, "WORK/OUT," which has 8 characters including the slash, is equivalent to "WORK/OUTSIDE."

The editor will accept upper- and lowercase letters within a program, but they will be equivalent during the running of the program. Therefore, "WashOut" and "WASHOUT" will be treated as identical.

However, upper- and lowercase letters are distinguished when used for file names.

To Change the Color of Characters

Sometimes it is desirable to change the color of the characters on the screen to make them brighter or clearer. To do this, press <CTRL> in combination with the selected color.

To Use Both Upper- and Lowercase Letters

To be able to type both upper- and lowercase letters, press the <SHIFT> and <C=> keys at the same time.

To Halt a Program While It Is Running

If a program is in an endless loop or if you simply want to stop it, e.g., to see at what line it is operating, press <RUN/STOP> in combination with <RESTORE>.

Saving the HELP File

Because "HELP" is a useful file, it can be copied onto formatted blank disks.

1. Type: E HELP <RETURN>
2. Remove: Kyan PASCAL disk
3. Insert: Formatted blank disk
4. Type the function key: F2

(F2 is the combination of <SHIFT> and F1.)

List of Editor and Compiler Commands

Here is the complete set of editor and compiler commands:

B Build stand-alone module
C Compile
D List directory of file (program) names
E Edit
F Name default file
R Run
P Print
F1 (Function key) Exit from editing with save
F2 (Function key) Remain in editing with save
F3 (Function key) followed by:
  G Continue editing on line ____
  S Substitute old string / new string
  F Find string forward
  B Find string backward
F4 (Function key) Undo editing
F5 (Function key) Continue editing at end
F6 (Function key) Continue editing at beginning
F7 (Function key) Continue editing 20 lines ahead
F8 (Function key) Continue editing 20 lines back
<INST/DEL> Deletes letter or keystroke to the left of the cursor
<INST/DEL>-<SHIFT> Deletes letter or keystroke coincident to the cursor
<CTRL>-color Changes color of the characters
<RUN/STOP>-<RESTORE> Stops program during run time
E ERROR$ Examine the error file after compilation
-S0: Scratch a file
-R0: Rename a file
-C0: Copy a file or merge several files
-N0: Format a disk or clear all files
-V: Validate a disk
-I: Reinitialize the disk drive

To transfer Advanced Kyan PASCAL from the disk to memory, type:

```
LOAD "E",8
```

followed by <RETURN>. Striking the <RETURN> key tells the computer that you have completed your entry. Unless instructed otherwise, you should always press <RETURN> when entering information into the computer.

Your Commodore 64 (Trademark of Commodore Inc.) will search for the program (SEARCHING FOR E), then indicate LOADING when the Kyan program is found. Completion of loading will take several minutes, at which point READY will be displayed:

```
READY
LOAD "E",8

SEARCHING FOR E
LOADING
READY
```

Type RUN to complete this task. The version number of your Kyan PASCAL editor will appear on the screen followed by the prompt (FILE NAME?).

```
READY
LOAD "E",8

SEARCHING FOR E
LOADING
READY
RUN

EDITOR (1.0)
COPYRIGHT 1985 BY KYAN SOFTWARE

FILE NAME?
```

A sample program, called "PAINT," is included on the Kyan PASCAL disk. If this is your first time using Kyan PASCAL, leave the disk in the drive so that you can use the sample program as you familiarize yourself with the editor commands.

Editing

The entering or changing of a program is called editing. Suppose you wish to edit the sample program, PAINT. Enter the file name PAINT after the prompt:

```
FILE NAME? PAINT
```

Programs are stored in separate files. Each program added to the disk must have its own file name consisting of 8 or fewer letters and/or characters. Spaces are not permitted within a file name.

As soon as <RETURN> is pressed, the program will appear:

```
PROGRAM PAINT;

TYPE A = ARRAY[1..25,1..40] OF CHAR;
VAR
 V,C : ^A;
 I,J : INTEGER;

BEGIN
  ASSIGN(V,1024);
  ASSIGN(C,-10240);
  FOR I := 2 TO 17 DO
    FOR J := 1 TO 40 DO BEGIN
      V^[I,J] := CHR(160);
      C^[I,J] := CHR(I)
  END
END.


F1 DONE     F3 GOLD     F5 TEXT>     F7 PAGE>
F2 SAVE     F4 UNDO     F6 <TEXT     F8 <PAGE
```

Kyan PASCAL includes an easily learned, full-screen, insert mode editor. Anywhere you move the cursor, a letter, a number, a space, or even a new line break may be added. This greatly facilitates editing your program. (Holding down the up-down <CRSR> key scrolls the screen.)

<INST/DEL> operates (as in Commodore BASIC) by deleting the character, space, or line break just to the left of the cursor. In addition, the combination of <SHIFT> and <INST/DEL> deletes coincident to the cursor.

The choice of function key F3, F5, F6, F7, or F8 allows you to go to any line in the program. F5 takes you to the end of your program; F6 takes you to the beginning of your program. F7 and F8 move you forward or backward 20 lines, respectively. When F3 is pressed, the following instructions appear:

```
GO TO:
G LINE-NUMBER

SUBSTITUTE:
S/OLD-STRING/NEW-STRING[/?]

FIND FORWARDS:
F [/STRING]

FIND BACKWARDS:
B [/STRING]
```

INSERT FILE:
I FILE NAME

CHANGE FILE NAME:
P [FILE NAME]

The go-to, substitute and find commands are two-stage commands. For example, to go to any line in the program, press F3. **When** the above instructions appear, enter G followed by the line number you want to go to. G followed by zero will take you to the last line.

To find the occurrence of any string in the program, press F3, then either F or B followed the the string which is to be found. For example, if "look out" is to be found, enter:

F3
F/look out

The cursor will go to the first occurrence of "look out" ahead of the present cursor position. To find the second occurrence, enter F3 F without the string. This process can be repeated until all occurrences of "look out" have been found. If you enter F3 B, the occurrence of the string in back of the present cursor position will be found.

Maximum string size is 40 characters.

To substitute a new string for any string in the program, enter F3 followed by:

S /old string/new string

The old string consists of the words or lines the way they are before the substitution. The new string consists of the new words or lines that will take their place. To replace the string "first" with the string "# 1," enter:

F3
S/first/# 1

The slashes are called delimiters. One delimiter is used before the old string and one delimiter is used before the new string. Any character may be used for a delimiter, but you must choose one that is not used in the strings. Otherwise, the editor will not find the correct end or beginning of the string.

Add a third delimiter and a "?" after the new string if you want the editor to stop before making a substitution:

F3
S/first/# 1/?

If you answer Y when the string if found, the substitution will be made. If you answer N, the substitution will not be made. In either case, the editor will automatically go on to find the next occurrence of the old string. You may answer Q at any time to quit the search and replace process.

Sometimes your PASCAL program will incorporate substantial parts of other programs. These may be brought into the version you are editing by using the I command. To include another file in the one you are editing press the key F3 then enter I and the name of the file to be included:

I File Name

If you wish either to change the name of the file you are editing, or just to display its name press the F3 key and then enter P followed by the new file name:

P File Name

If P is entered without a file name, the current file name is displayed.

Editor: Block Move Commands

You may take any section of the program, and move it as a block. To mark a block: 1) Move the cursor to the character or space that is at the beginning and type the Control and O keys together, <CTRL>-O. Notice how the entire block is displayed in inverse video as you move the cursor. 2) Go to the last character or space in the block. Type <CTRL>-O to mark the end of the block.

Then entire block will seem to disappear. Actually, it is saved in memory so that it can be moved to any location you choose.

Move the cursor to the position where you wish the block to be inserted. Type <CTRL>-P and the block will be "pasted" in the new position. These commands are sometimes called cut and paste. As many copies as you wish may be pasted.

The block move commands are:

first <CTRL>-O	marks the start of block
second <CTRL>-O	marks end of block and puts block into memory (cut)
<CTRL>-P	insert the block (paste)

Editing PAINT, an Example

Use the cursor <CRSR> keys to go to the end of the line "FOR J := 1 TO 40 DO BEGIN." Then add a period to the end of the line.

You have now entered a change in the program PAINT and are ready to save the new version.

Saving a File

Insert a formatted blank disk before attempting to save the edited version of PAINT.

To save your program, you can press the function key F1 or F2. F1 differs from from F2 in that F2 returns you to the editor immediately after saving the program, whereas F1 requires that you enter E to return to the editor.

If you press F4 (<SHIFT> F3), the changes produced by editing will not be saved, and the previous version of the program will remain unaltered. F4 is called the undo key.

Now press F1 to save PAINT.

The File Directory

To list the files on your disk, you must leave the editor as you did when you saved the file PAINT above. All the files on the disk will be displayed when you enter:

```
LOAD "$",8
```

When the directory program is loaded enter LIST to display the files.

Compiling a File

Before a PASCAL program can be run, it must be entered into the computer's memory, compiled and translated into machine language.

To compile the program, enter:

```
LOAD "C",8
```

Type RUN followed by <RETURN> after the program has been loaded and the the prompt: READY is displayed. The compiler program will ask which file you wish to compile. Enter the name of the file. If there are errors, they will be counted and listed, and the program cannot be run.

Try to compile the edited version of PAINT. Because of the change you have made, there will be an error.

```
LOAD "C",8

SEARCHING FOR "C"
LOADING
READY
RUN

FILE NAME? PAINT
LISTING (Y/N)? Y
HARDCOPY (Y/N)? Y

0010      FOR J := 1 TO 40 DO BEGIN.
                                   1
(1) ";" OR "END" EXPECTED

0001 ERRORS

>
```

The line with the error is displayed with its line number. The "1" underneath the line indicates where the error occurred. A description of the error type is also displayed.

Sometimes, a single error in a program (such as a missing semicolon after a VAR declaration) will generate multiple errors following it.

Notice the two prompts that ask whether to create an Error "Listing" and/or "Hardcopy." In the above example the list of errors will be sent both to the screen and the printer. To list the errors only on the screen answer Y to "Listing" and N to "Hardcopy."

Go back to the edit mode and correct the error you introduced. Try compilation again. Now there should be no error message. The program is ready to run.

Running a File

After a file has been successfully compiled a new file will be created than can be run. Furthermore, this file may be run without the Kyan PASCAL disk in the system if the Kyan library file "L" is also on the disk.

All executable PASCAL files have the suffix "%." For example to run PAINT enter:

```
LOAD "PAINT%",8
```

When PAINT% has been loaded from the floppy disk enter RUN:

```
LOAD "PAINT%",8

SEARCHING FOR PAINT%
LOADING
READY
RUN
```

The program will run and perform the tasks it was designed to do. (A program with no errors on compilation may still be faulty. For example, programming the area of a circle to be "r*r" instead of "3.14*r*r" is an error the compiler will not detect.)

It is important to remember that the file "L" must be on the same floppy disk as the executable file "PAINT%" otherwise the program cannot be run. Notice that when "RUN" is entered, the disk light turns on, indicating that part of "L" is being loaded into memory. See the section of this chapter on copying files to learn how to copy the files from one disk to another.

Printing a Program

Any PASCAL source code file may be printed using PRINT%. For example, to print the program PAINT enter the following:

```
LOAD "PRINT%",8
```

When the program prompts, enter the file name of the program to be printed:

```
SEARCHING FOR PRINT%
LOADING
READY RUN
FILE NAME? PAINT
```

File Management

The Commodore 64 provides a set of file management commands that may be accessed at any time the computer is in the BASIC environment. Thus when a program is running or being edited the following commands may not be used.

In all of the following commands the disk files must be opened, the command entered, and the files closed. For example, enter:

```
OPEN 15,8,15
```

The disk drive light will go on and when the files are opened it will reply "READY." Next, enter the specific disk drive command. For example to format a disk enter:

```
PRINT #15,"NO:Name,ID"
```

"Name,ID" is the name you give to your disk (Kyan PASCAL does not use this identifier), where ID is restricted to two characters.

When the computer again indicates "READY" close the disk drive files:

```
CLOSE 15
```

To clear the disk of all files, without formatting enter:

```
PRINT#15,"NO:Name"
```

The following commands will be illustrated as they appear completed on the screen. To delete a file called "First" and also to delete its compiled version, "First$," enter:

```
OPEN 15,8,15
READY
PRINT#15,"SO:First"
READY
CLOSE 15
READY
```

Sometimes a new program will be a variation of or an extension of an existing one. The first step in producing the new program would be to make a copy of the old one. For example, if "Factor" is to be a copy of the file named "First," enter:

```
OPEN 15,8,15
READY
PRINT#15,"CO:Factor=First"
READY
CLOSE 15
READY
```

To combine the files "Factor" and "Max" and call them "Makeup," enter:

```
OPEN 15,8,15
READY
PRINT#15,"CO:Makeup=Max,Factor"
READY
CLOSE 15
READY
```

To change the name of a file named "Smith" to "Jones," enter:

```
OPEN 15,8,15
READY
PRINT#15,"RO:Jones=Smith"
READY
CLOSE 15
READY
```

The validate command enables you to clean up your disk. It increases the room on the disk by consolidating unused spaces and deleting improperly closed files. To validate your disk, enter:

```
OPEN 15,8,15
READY
PRINT#15,"V"
READY
CLOSE 15
READY
```

If the disk drive loses its place, or for some other reason you wish to restart, i.e., to initialize it, enter:

```
OPEN 15,8,15
READY
PRINT#15,"I"
READY
CLOSE 15
READY
```

Copying Files from Disk to Disk

To run the file copying utility first load the program from the Kyan PASCAL disk to memory. You must load the program from Commodore Basic, which is indicated by the prompt READY. This is the same procedure as loading the editor. When the program has loaded type RUN:

```
READY
LOAD "COPY", 8

SEARCHING FOR COPY
LOADING
READY
RUN

COPY 1.0
COPYRIGHT 1985 BY KYAN SOFTWARE

THIS PROGRAM DUPLICATES PRG FILES ONLY

FILE NAME?
```

When the computer prompts for file name as above enter the name of the file to be copied in quotes. The copy program will then prompt for the source disk (If you wish to copy "L" it is the disk with "L":) and the destination disk (Where "L" is to go). Press <RETURN> after inserting each disk:

```
FILE NAME? L

INSERT SOURCE DISK

INSERT DESTINATION DISK
READY
```

The copy program copies PASCAL programs and the Kyan PASCAL library (L) only.

To Change the Color of Characters

Sometimes it is desirable to change the color of the characters on the screen to make them brighter or clearer. To do this, press <CTRL> in combination with the selected color.

To Use Both Upper- and Lowercase Letters

To be able to type both upper- and lowercase letters, press the <SHIFT> and <C=> keys at the same time.

To Halt a Program While It Is Running

If a program is in an endless loop or if you simply want to stop it, e.g., to see at what line it is operating, press <RUN/STOP> in combination with <RESTORE>.

List of Commands

Load "E",8	Load the editor program
Load "C",8	Load the compiler program
Load "PRINT%",8	Load the print program
Load "$",8	Load the list directory program

Here is the complete set of editor commands:

<INST/DEL> Deletes letter or keystroke to the left of the cursor
<INST/DEL>-<SHIFT> Deletes letter or keystroke coincident to the cursor
<CTRL>-O start/stop block
<CTRL>-P insert block
F1 (Function key) Exit from editing with save
F2 (Function key) Remain in editing with save
F3 (Function key) followed by:
- G Continue editing on line ____
- S Substitute old string / new string
- F Find string forward
- B Find string backward
- I Include File
- P Rename File

F4 (Function key) Undo editing
F5 (Function key) Continue editing at end
F6 (Function key) Continue editing at beginning
F7 (Function key) Continue editing 20 lines ahead
F8 (Function key) Continue editing 20 lines back

Here is a list of file management commands useful while in BASIC mode:

SO: Scratch a file
RO: Rename a file
CO: Copy a file or merge several files
NO: Format a disk or clear all files
V: Validate a disk
I: Reinitialize the disk drive

Other commands:

<CTRL>-color Changes color of the characters
<RUN/STOP>-<RESTORE> Stops program during run time

PART I

SAMPLE PROGRAMS

EGO PROGRAM

The first program shows how to print a message.

```
PROGRAM Ego(Output);

BEGIN
  Writeln;
  Writeln;
  Writeln('My name is Sam Smith.')
END.
```

This program will put the message "My name is Sam Smith" on the screen or printer.

Program Statement and Reserved Words

The name of the program is Ego. It appears after the word PROGRAM. To end the statement which names the program we use a semicolon (;). If we did not use the semicolon, the computer might think that the next statement, "BEGIN," was part of the program name.

PASCAL has a precise vocabulary. Part of this vocabulary consists of words that cannot be used by the programmer as names within his or her program. PROGRAM and BEGIN are two such "reserved" words. It would be illegal to use the word "program" for the name of the program. Reserved words will be written in capital letters when they appear in programs in this manual.

As a general rule, do not use any of the vocabulary of PASCAL for the name of anything within the program. In addition to reserved words, this includes predefined words such as Integer, Read, and others whose meaning is consistent from one implementation of PASCAL to another. In this manual, all predefined words will be written with only the starting letter capitalized (except EOF and EOLN, which are acronyms for "end of file" and "end of line").

Of course, comments (*which appear between parentheses and asterisks like this*) and literals ('which appear between parentheses and single quotes like this') are not restricted.

Declaration and Program Body

Every PASCAL program has two main parts: the <u>declaration</u> and the <u>program body.</u>

The above program begins with a statement of the name of the program. Some programs also include lists of constants and variables. The naming of the program, constants and variables constitutes the declaration part of the program.

After the declaration is the portion of the program where computations, input, and output can occur. It is denoted by the word BEGIN and is called the program body. The word END followed by a period lets the computer know where the program body ends.

The indentation of statements in Ego and other programs in this manual is intended to help clarify the program structure; it is not recognized by the compiler.

Analysis of Ego

The first statement declares the name of the program, which is Ego.

The next line, BEGIN, tells the computer the following statements are part of the program body.

The third and fourth statements (Writeln, short for "write line") create two blank lines on the screen before the message.

The fifth statement causes the message to appear on the screen or on the printer:

    My name is Sam Smith.

CONSTRUCTION PROGRAM

The second program we are going to run will calculate the cost of constructing an apartment building, given the hours worked, the rate of pay, and the cost of materials.

```
PROGRAM Construction(Input,Output);
(*Dollar units are thousands*)

CONST
  Material = 325.0;

VAR
  Hours, Rate, Labor, Total : Real;

BEGIN
  Writeln  ('Enter hours worked and rate of pay');
  Readln (Hours, Rate);
  Labor := Hours * Rate;
  Total := Labor + Material;
  Writeln ('Labor = $', Labor : 8:3, ' Total = $', Total : 8:3)
END.
```

Analysis of Construction

The objective of the program is to calculate the Labor cost and Total cost for the construction project. The calculation of these costs will depend on the Hours worked and the Rate of pay during those hours.

The first part of the program gives the names of the program, the constants, and the variables. In PASCAL, user-defined names are called identifiers.

The fixed cost of the materials is given by the identifier Material and is $325,000.

Notice how the variables are listed after the reserved word VAR. "Real," although not a reserved word, is predefined and specifies that all the variables that precede it are Real numbers.

The first statement in the program body writes the following line on the screen:

"Enter hours worked and rate of pay"

The second statement reads the values for Hours and Rate which the user enters on the keyboard. Once these values are known, the Labor and Total costs can be calculated by the third and fourth statements in the program body.

The final statement in the program body writes the Labor and Total costs on the screen.

Algorithm

Step 1: Get the values of hours worked (Hours) and rate of pay (Rate).

Step 2: Multiply Hours times Rate to get the cost of the labor (Labor).

Step 3: Add the constant 325000 (Material) to Labor to get the total cost (Total).

Step 4: Output the Labor and Total costs.

Identifiers

An identifier is a name. It can be the name of a PASCAL program or program subsection, or it can be the name of some quantity that is used in a PASCAL program. Just as in algebra we can define a constant, C = 5, in PASCAL we can say:

```
CONST
  C = 5
```

The rules for constructing an identifier are: (1) it must start with a letter (A - Z or a - z), and (2) any combination of letters and numbers may follow. Although more than 8 characters may be used, only the first 8 will distinguish one identifier from another. The compiler does not distinguish between upper- and lowercase letters.

Write and Read Commands

Write, Writeln, Read, and Readln (short for "read line") commands pass information to and from the computer. Read and Readln enter data from the keyboard into the computer; Write and Writeln send data to the screen or printer.

The terms Input and Output should appear in parentheses after the program name to tell the compiler that data will be transfered into and out off memory.

Input and Output and Printing the Output - STANDARD KYAN PASCAL ONLY

After the program "Construction" was named, the two standard PASCAL terms, Input and Output, appeared in parentheses. Technically the compiler sees these terms as identifing a files:

```
PROGRAM Average(Input,Output);
```

Files allow information to go to and from places outside the directly addressable memory space of the computer. In this implemenation of PASCAL, information input at the keyboard goes into the Input file, and information output to the CRT goes into the Output file.

Another useful output is the non-standard "Printer." If some of the output is to go to the printer as well as to the CRT screen, "Printer" must be included in the declaration section of the program as if it were a file of characters, viz. Text:

```
VAR   Printer : Text;
```

Every Write and Writeln statement to be printed also includes "Printer":

```
Writeln (Printer,  'Labor = $', Labor:8:3,
                   'Total = $', Total:8:3);
```

Input and Output and Printing the Output - ADVANCED KYAN PASCAL ONLY

After the program "Construction" was named, the two standard PASCAL terms, Input and Output, appeared in parenthese. Technically the compiler sees these terms as identifing a file:

```
PROGRAM Average(Input,Output);
```

Files allow information to go to and from places outside the directly addressable memory space of the computer. In this implemenation of PASCAL, information input at the keyboard goes into the Input file, and information output to the CRT goes into the Output file.

It is also useful to be able to define the output as the printer. To do this the non-standard Kyan PASCAL procedures, PRON and PROFF, are used to redirect the Output to the printer or back to the screen. These procedures are in the file PR.I which must be included in the declaration part of the program. For example in the preceeding program:

```
PROGRAM Construction(Input,Output);

CONST  Material = 325.0;

VAR  Hours, Rate, Labor, Total : Real;

#i PR.I              (*include procedure to redirect output*)

BEGIN
  Writeln  ('Enter hours worked and rate of pay');
  Readln (Hours, Rate);
  Labor := Hours * Rate;
  Total := Labor + Material;
  PRON;                    (* redirect Output to printer *)
  Writeln ('Labor = $', Labor : 8:3, ' Total = $', Total : 8:3)
  PROFF;                   (* redirect Output to screen *)
END.
```

Readln

When data is read from the keyboard using Readln, more than one variable may be input as in:

```
Readln (Hours, Rate);
```

Data entered at the keyboard must include spaces or <RETURN> to distinguish the variables. In the examples below, Hours would get the value 10000 and Rate would be set to 14.20:

Example A: 10000 14.20 <RETURN>

Example B: 10000 <RETURN> 14.20 <RETURN>

CONST

Use of constants, CONST, makes programs easier to read and maintain. Suppose next year the cost of materials rises to $330,000. Also suppose that we had not used the constant, Material, and instead had said the total cost was:

```
Total := Labor + 325.00
```

In order to change the materials cost we would have to reanalyze the program, because in many programs a constant appears more than once. We would have to find every occurrence of 325,000. Then, we would have to make sure each time that it wasn't some other constant, such as Taxes.

It is much easier to go to the CONST declaration and change "Material = 325.00" to "Material = 330.00."

PROGRAM TO FIND THE AVERAGE

The following program finds the average of two numbers.

```
PROGRAM Average(Input,Output);
(*Computes the average of two numbers*)

VAR
  X1, X2, Average : Real;

BEGIN
  (*Read the two numbers*)
  Write ('First number = ');
  Readln (X1);
  Write ('Second number = ');
  Readln (X2);

  (*Compute Average*)
  Average := (X1 + X2)/2;

  (*Print Average*)
  Writeln ('Average = ', Average : 9:2)
END.
```

The following is a sample run of Average:

```
First number = 12
Second number = 8
Average = 10.00
```

In this book, data entered on the keyboard will be underlined.

Readln and Writeln

"Write ('First number =')" causes "First number =" to appear on the screen. The user then enters the first number, which in the above example is 12. If the program had used "Writeln ('First number ='),'' there would have been a <RETURN> and the user would have had to enter 12 on the line below the prompt.

"Readln (X1)" enters data from the keyboard into the computer. The entire line is read, up to and including the <RETURN>. However, the data that is assigned to X1 depends on what type X1 is. For example, suppose the data entered at the keyboard is "123 RALPH <RETURN>" and that X1 is Char type. Then X1 equals 1. The remaining characters and <RETURN> are lost. If X1 is of the type ARRAY[1..9] OF Char, then X1 equals 1,2,3, ,R,A,L,P,H; only the <RETURN> is lost. Finally, if X1 is of the type Integer, then X1 equals 123, and the remaining characters and the <RETURN> are lost. (Note: the preceding data types will be fully discussed in later sections of the manual.)

Suppose we wish to assign the input data 123 and RALPH be assigned to two variables, X1 and X2, respectively. Let X1 be of type Integer and X2 be an array [1..6] of type Char. Then "Read (X1)" followed by "Read (X2)" will accomplish this task. The Read statement differs from the Readln statement in that any input data not of the type of the variable in parentheses is left over for the next Read or Readln statement.

Because the remaining characters up to and including the <RETURN> are not cleared after "Read (X2)" above, the <RETURN> will be read as the first entry in the next Read or Readln statement. In most programs this is not desirable. This problem could be corrected by changing the statement to "Readln(X2)." An alternate method of assigning data to these variables would be "Readln (X1,X2)."

Real and Integer Data Types

Real numbers in PASCAL are positive or negative numbers represented in scientific (floating point) or decimal notation. Examples are 12.8, 3.456E+11, and -2.5555E+4. A number in decimal notation must have at least one digit before and one digit after the decimal point. Very large or very small numbers are best handled in scientific notation.

The statements following the declaration of the variable Z are equivalent:

```
VAR
  Z : Real;
BEGIN
  Z := -345.55;
  Z := -3.4555E+02
END.
```

In the program to find the average, X1 and X2 were declared to be of the type Real. The range of values that may be assigned to X1 or X2 is from ±9.9999999999E-99 to ±9.9999999999E+99.

Suppose we wish to declare an Integer variable, En. A declaration statement for En would be written as:

```
VAR
  En : Integer;
```

Integer numbers must be within the range -32768 to +32767.

If arithmetic expressions are formed by mixing Integer and Real types, the result will be expressed as a Real type.

When a Real or Integer number is written, the format specifies how many spaces are reserved for it and other details of how it will appear in print or on the screen. Notice in the Writeln statement that the size of the space reserved for the Real number is 9:2. This means that the number is to be printed in decimal notation. If the format were simply 9, the number would be printed in scientific notation. The format for Integers never needs to be larger than "5" because of the range limitation.

The format 9:2 reserves nine spaces total. This includes one space for the sign and one space for the decimal point. Finally, two spaces are reserved for the digits following the decimal.

If a number has fewer digits than the number of spaces reserved for it, the correct number will appear, but the compiler will fill in the extra spaces with blanks or zeroes. If a number in decimal format has more digits than the number of spaces reserved for it, a run-time error will occur. Programmers must think ahead when using the decimal format.

Run-time errors also occur when a number is out of range. For example, if X is an Integer that has the value -32800, an error will occur.

Real numbers are limited to 13 significant digits. Writing a format that reserves more than 13 spaces for a Real number will not make the number more accurate. The computer will present the correct number, with the digits beyond 13 filled in with blanks or zeroes. On the other hand, calling for fewer than 13 digits does not take advantage of all the accuracy available.

Trunc, Round and Maxint

The truncate function (Trunc) takes a decimal or floating point number and disposes of the non-Integer portion, leaving an Integer value. Round gives the integer value closest to the floating number by adding 0.5 before truncating. For example:

```
Trunc(5.9) = 5;           Round(5.9) = 6;
Trunc(75.3E-01) = 7;      Round(75.3E-01) = 7;
```

The maximum size of any Integer number is 32767 or -32767. Trunc or Round will cause an error if either operates on a Real number larger than ± 32767.

Maxint is the standard PASCAL constant whose value is the maximum Integer size. In this edition of PASCAL, Maxint = 32767. It will vary with different computers and compilers.

Arithmetic Operators

PASCAL uses the following arithmetic operators for Real and Integer data:

Add	+
Subtract	-
Multiply	*
Divide	/

Multiplication and division are performed before addition and subtraction. For example:

6 + 8/2 = 10, not 7.

SOCIAL SECURITY PROGRAM

The following program calculates the amount of social security tax to be deducted from each paycheck.

```
PROGRAM SocialSecurity(Input,Output);

CONST
  TaxRate = 0.075;
  TaxMaximum = 4275.0;

VAR
  Hours, Rate, TaxNow, TaxToDate : Real;

BEGIN
  (*Read hours, rate, and tax to date*)
  Writeln;
  Writeln;
  Write ('Hours worked = ');
  Readln (Hours);
  Write ('Hourly rate = $');
  Readln (Rate);
  Write ('Soc Sec tax paid to date = $');
  Readln (TaxToDate);

  (*Compute Soc Sec Tax for this period*)
  TaxNow := Hours*Rate*TaxRate;

  (*Test: IF TaxToDate + TaxNow is > Tax-
  Maximum THEN TaxNow must be recalculated*)
  IF TaxToDate + TaxNow > TaxMaximum THEN
    BEGIN
    TaxNow := TaxMaximum - TaxToDate;
    TaxToDate := TaxMaximum
    END (*IF true*)
  ELSE (*IF false*)
      TaxToDate :=TaxNow + TaxToDate;

  (*Write Results*)
  Writeln ('Soc Sec Tax This Pay Period = $',TaxNow :8:2);
  Writeln ('Soc Sec Tax To Date = $',TaxToDate :8:2)
  END.
```

Relational Operators

There are six relational operators that may be used to decide which of two branches will be taken within a program. One branch is taken if the relationship is true, the other if it is false. The six relational operators are:

=	equal to
<>	not equal to
<	less than
>	greater than
<=	less than or equal to
>=	greater than or equal to

In the program Social Security, IF the condition is true, THEN the tax for the present pay period, TaxNow, must be recalculated. Otherwise, the program skips the recalculation steps.

The IF-THEN Statement

Notice in the program above that there are two program steps following the IF statement. These are grouped between a BEGIN-END pair so that both will be performed when the IF statement is true. (Otherwise, only the first statement, TaxNow, would be associated with IF-true, and the second statement, TaxToDate, would be outside IF-THEN control and would be performed regardless.)

For program clarity, the comment (*IF true*) has been placed to signify the end of the program branch that will be executed if the condition is true.

Sometimes it is necessary to include some program statements for when the IF condition is False. These are added after the reserved word ELSE:

```
IF TaxToDate + TaxNow > TaxMaximum THEN
    BEGIN
    TaxNow := TaxMaximum - TaxToDate;
    TaxToDate := TaxMaximum
    END (*IF true*)
ELSE(*IF false*)
    TaxToDate := TaxNow + TaxToDate;
```

The statement following ELSE will only be executed if "TaxToDate + TaxNow > TaxMaximum" is false.

There is no semicolon after END (*IF true*) above. It is incorrect to terminate the statement preceding ELSE with a semicolon.

The Assignment Statement

Although the equal sign was listed above as a relational operator, the difference between equal (=) and the assignment operator (:=) might not be clear. If we examine an assignment statement from another program, the difference becomes clear:

```
AgeNow := Birthdays + AgeNow;
```

This statement is meant to recalculate the variable, AgeNow. The old value of AgeNow is on the right and the new value of AgeNow is on the left. In general, when the assignment symbol (:=) is used, the result is on the left.

The equal operator is used almost exclusively to determine which of two branches will be taken following a conditional statement. The only time the equal operator is used like an assignment statement is in a CONST declaration.

ALPHABETIZE PROGRAM

This program illustrates the use of data in the form of words. It finds the alphabetically first word on a list and counts the total words in the list. Since the size of the list is not known in advance, a signal word, stop, is used to indicate the end of the list.

```
PROGRAM FirstWord(Input,Output);
(*This  program selects the alphabetically first word and  counts
the total words tested*)

CONST
  Signal = '+';

TYPE
  String = ARRAY [1..15] OF Char;

VAR
  Word, LeastWord : String;
  LoopCount : Integer;

BEGIN
  (*Each time  through the loop, increment  the
   counter, LoopCount, and save the least word*)

  Write('Enter a word or "+": ');
  Readln(Word);
  LeastWord := Word;
  LoopCount := 0;
  WHILE Word[1] <> Signal DO
    BEGIN
      IF Word < LeastWord THEN
        LeastWord := Word;
      LoopCount := LoopCount + 1;
      Write('Enter a word or "+": ');
      Readln(Word)
    END; (*WHILE LOOP*)
  Writeln;
  Writeln;
  Writeln(LoopCount:5, ' words were entered.');
  Writeln(LeastWord, ' is alphabetically first.')
END.
```

<u>FirstWord Algorithm</u>

Step 1: Input the first word on the list to be alphabetized.

Step 2: Initialize variables: LeastWord = Word, LoopCount = 0

Step 3: Begin WHILE loop. Exit WHILE loop when Word = "+."

Step 3a: (WHILE loop) Input the next Word.

Step 3b: Increment LoopCount.

Step 3c: IF current Word is alphabetically first, LeastWord = Word.

Step 4: Output LeastWord and LoopCount.

String and Char Types

So far only two types of data have been discussed, Real and Integer. Another type, Char, is a predefined type that denotes a variable, constant, or other piece of data that is in the form of a single character.

Suppose we define a variable, Digit, to be of the type Char:

```
VAR
  Digit : Char;
```

This means Digit will always be a single "printable" character. It may be a letter, a number, or a symbol. In addition, it could be a space or a <RETURN>, but control characters such as "<CTRL> Q" are not allowed.

Although digits such as '1' (single quotes are used to denote Char values) may be of this type, they are not the same as Integers and ordinary arithmetic may not be performed on them.

Another type of data is called String. Any string of characters and spaces, such as "is alphabetically first," constitutes a string. When string data are entered on the keyboard, the end of the string is signaled by <RETURN>.

In Kyan PASCAL, the following statement declares a String:

```
String = ARRAY [1..15] OF Char
```

Since String is user defined, any number of characters may be specified, although 15 characters are used in "FirstWord." When a word with fewer than 15 letters is entered in this program, Readln will fill in the remaining places with blanks. If a word with more than 15 letters is entered, the extra letters will be ignored.

When String and Char values are assigned in a program statement, quotes are used:

```
VAR   Word : String;
    Letter : Char;
BEGIN
    Word := 'Help           ';
    Letter := 'A';
```

The number of characters in a String must be correct. Thus, there are 11 blanks in Word, which is defined as a 15-character String. Char is always a single character.

WHILE

The WHILE loop is repeated as long as the specified condition is true. If there is more than one statement in the loop, BEGIN and END must be used to mark the boudaries. Usually indentation is used to clarify the boundaries of the loop (although indentation has no significance to the compiler).

A program would never exit from the following loop, because Ex1 will never equal or exceed the test value:

```
PROGRAM Never;

CONST
  Alpha = 4.6; Pi = 3.14;

VAR
 Ex1,Ex2 : Real;

BEGIN
  Ex1 := Alpha;
    WHILE Ex1 < 5.432 DO
       BEGIN
         Ex1 := Ex1 - 1.00;
         Ex2 := Ex1*Pi
       END (*WHILE*)
END.
```

FACTORIAL PROGRAM

The following program calculates the factorial function of a given number. The factorial function is used quite frequently in analysis of probabilities.

```
PROGRAM CalcFact1(Input,Output);
(*This  program  computes n! where n = Integer*)
(*The result is an Integer*)

VAR
  Number,LoopCount,Factorial : Integer;

BEGIN
  Writeln;
  Writeln;
  Writeln('This program calculates the factorial'
  Writeln('of an Integer, N.');
  Write('Enter a value.  N = ');
  Readln(Number);

  Factorial := 1;
  FOR LoopCount := 1 TO Number DO
    BEGIN
      Factorial := Factorial*LoopCount
    END;(*FOR*)

  Writeln;Writeln;
  Writeln ('N! = ',Factorial : 6)
END.
```

<u>Analysis of Program</u>

If a number is equal to zero or one, its factorial is defined as one. In all other cases n! = 1*2*3*...*(n-1)*n.

1. Input N (Number).

2. Initialize N! (Factorial) = 1.

3. Begin FOR loop. Start with LoopCount = 1.
   Increment LoopCount until

   Loop Count = N (Number).

   For each pass through the loop, calculate a new value for Factorial:

   Factorial = Factorial*LoopCount.

4. Output N! (Factorial).

FOR Loops and Loop Control Variable

CalcFact1 uses the FOR loop, which increments a loop control variable from some initial value to some final value. Although the loop control variable is an Integer, in other uses of the FOR loop it might be an alphabetic character (Char).

The FOR loop may also decrement the loop control variable if written in the following form:

```
FOR LoopCount := Number DOWNTO 1 DO
```

BOOLEAN PROGRAM

```
PROGRAM DivLesn(Input,Output);

VAR X,W,Z : Integer;
    Ans : Char;
    Correct : Boolean;

BEGIN
 Ans := 'Y';
 WHILE Ans = 'Y' DO
  BEGIN
  Write('Enter an Integer ');Readln(X);
  Write('One of the factors is ');Readln(W);
  Write(X : 3, 'divided by',W : 3, 'is ');
  Readln(Z);
  Correct := (X MOD W = 0) AND (X DIV W = Z);
  IF Correct THEN
   BEGIN
   Write('Correct! Another? Enter Y or N ');
   Readln(Ans)  END(*IF THEN*)
  ELSE
   BEGIN
   Write('Incorrect. Try again? Enter Y or N ');
   Readln(Ans)  END (*IF ELSE*)
  END (*WHILE*)
END.
```

Boolean Data Type

Boolean is a predefined type. Boolean type expressions, variables, and constants are always in one of two states: they are either in the True state or the False state.

In the program above, the IF statements are executed only when Correct equals True. Correct is a Boolean variable which is true when both of the parenthetical statements following it are true (see DIV and MOD operators below).

The AND operator means that both equalities in parentheses must be True; otherwise, Correct will be false and the next two statements will be skipped.

DIV and MOD Operators

The DIV and MOD operators give the quotient and the remainder of a division problem when the divisor and dividend are both of the type Integer. The general form is:

```
Integer1 DIV Integer2   (* = quotient*)
Integer1 MOD Integer2   (* = remainder*)
```

For example, if Integer1 = 14 and Integer2 = 4, then 14 DIV 4 = 3 and 14 MOD 4 = 2.

Boolean Operators

Up to this point we have discussed only the manipulation of Real and Integer type data. This included the add, subtract, multiply and divide operators. There are also Boolean operators:

NOT
OR
AND

Boolean operators follow the rules of formal logic and can be diagramed in truth tables.

```
NOT:     False = NOT True
         True = NOT False
```

An example of NOT: A coin is flipped. If it is NOT heads (True), it is tails (False). If it is NOT tails (False), it is heads (True).

```
OR:      True =  True OR False
         True = False OR True
         True = True OR True
         False = False OR False
```

An example of OR: Two cars are racing. The race is over (True) whenever car A crosses the finish line OR car B crosses. Only one condition has to be True for the result to be True.

```
AND:     False = True AND False
         False = False AND True
         True = True AND True
         False = False AND False
```

An example of AND: The environment is clean (True) only when both the air AND water are clean. Both conditions have to be True for the result to be True. AND is also illustrated by the program DivLesn.

Operator Precedence

Operations within parentheses are performed first. For example: 4*(5+1) = 24, while (4*5)+1 = 21. If parentheses are nested, the operation within the innermost pair is done first: 3*(2+(6/2)) = 15.

However, it is not always necessary to use parentheses, because operator precedence is predefined: operations of higher precedence are performed before operations of lower precedence. If the levels are equal, it does not matter which is performed first.

The five levels of precedence in PASCAL are:

1st--Highest Precedence: ()

2nd--Level of Precedence: NOT

3rd--Level of Precedence: *, /, AND, DIV, MOD

4th--Level of Precedence: +, -, OR

5th--Lowest Precedence: =, <=, >=, >, <, <>

MULTI-DIGIT HEXADECIMAL CONVERSION

The following program converts a hexadecimal number into a decimal number.

```
 PROGRAM Hexadecimal(Input,Output);
  (*Hexadecimal to base ten*)

  TYPE
    YesNo = (Yes,No);

  VAR
    Digit, Signal : Char;
    Number, OldNumber : Integer;
    Answer : YesNo;
    Continue : Boolean;

  BEGIN
    OldNumber := 0;
    Write('Enter the most significant-far left-digit ');
    Readln(Digit);

    REPEAT
      CASE Digit OF
        '0' : Number := 0;
        '1' : Number := 1;
        '2' : Number := 2;
        '3' : Number := 3;
        '4' : Number := 4;
        '5' : Number := 5;
        '6' : Number := 6;
        '7' : Number := 7;
        '8' : Number := 8;
        '9' : Number := 9;
        'A' : Number := 10;
        'B' : Number := 11;
        'C' : Number := 12;
        'D' : Number := 13;
        'E' : Number := 14;
        'F' : Number := 15
      END (*CASE*);

      OldNumber := Number + OldNumber*16;
      (*The more significant digit (OldNumber) is a power of 16
        times greater than the next digit (Number) *)
      Writeln('Is there another digit');
      Write('after this one (Yes/No)? ');
      Readln(Signal);
      IF (Signal = 'Y') OR (Signal = 'y') THEN
        Answer := Yes
       ELSE
        Answer := No;
      IF Answer = Yes THEN
        BEGIN
         Continue := True;
         Write ('Enter the next digit ');
         Readln (Digit)
        END (*IF Answer true*)
       ELSE
```

```
        Continue := False;
    UNTIL NOT(Continue);

    Writeln;Writeln;
    Writeln('The decimal equivalent is ', OldNumber : 6)
    END.
```

Algorithm

1. Initialize OldNumber := 0
2. Input the most significant Digit
3. REPEAT
   3a. Convert Digit to decimal Number
   3b. OldNumber := Number + OldNumber*16
   3c. Is there another digit?
       3ca. IF NOT(Continue) = False, input the next most significant digit
4. UNTIL NOT(Continue) = True
5. Output base ten number (OldNumber)

REPEAT UNTIL

The REPEAT UNTIL loop is very much like the WHILE loop discussed earlier. The statements in the loop are repeated until the specified condition becomes True. (The WHILE loop continues until the condition becomes False.) It is important to note that the REPEAT UNTIL condition is tested at the end of the loop rather than at the beginning like the WHILE condition.

Scalar Types and Boolean Variables

In the program above, Hexadecimal, Answer is a scalar variable. Scalar variables are used when there is a short list of names, words, numbers, or other legal identifiers that the variable might be. A "scalar type," which is user defined, gives the possible values of a scalar variable. Listed below are two scalar types:

```
TYPE
 DaysWeek = (Mon,Tue,Wed,Thur,Fri,Sat,Sun);
 PayRate = (Regular,Overtime);
```

The scalar variables below may take on any of the values listed in the type declaration, but no others.

```
VAR
  Day : DaysWeek;
  Rate : PayRate;
```

The following declaration of PayNames is illegal because the values in a scalar type cannot be defined in terms of any other type. Because quotes are used, 'A' and 'B' are of the type Char, and 'Other' is a string. Without quotes they are simply identifiers, and are therefore acceptable. Characters or strings cannot be used, nor can integers or real numbers.

```
TYPE   PayNames : ('A', 'B', 'Other');
```

The only exception to this rule is explained below in the definition of a scalar type subrange.

A Boolean variable is much like a scalar variable where the type would be:

```
TYPE
 Boolean = (True, False);
```

In the program above, the variable Continue can be either True or False. Whether Continue is true or false is determined by the assignment statement where Continue is (:=) True when Answer is (=) Y or y.

Subrange Types

The subrange type is a form of the scalar type where only the first and last value or item within the range have to be specified. For example, if the variables Component, IC, and Resistance are to take on a range of values and each of the possible values is known from the beginning of the program, then they might be declared as follows:

```
TYPE
  CompType = (Resis,Cap,Trans,Diode,OpAmp,
       Rgltr,Osc,GateArray,Trnfr,Coil);
  ResRange = 1..100;
  ICrange =OpAmp..GateArray;

VAR
  Component : CompType;
  Resistance : ResRange;
  IC : ICrange;
```

Both ResRange and ICrange in this example are subrange types. (CompType is a scalar type.) ResRange is a subrange of the Integer type. ICrange is a subrange of CompType declared before it.

Although ResRange is an example of a subrange of the type Integer, scalar types of the type Integer are not permitted. This restriction precludes the inadvertent redefining of a predefined type.

CASE OF

Sometimes, especially in programs that use scalar type variables, a series of IF..THEN tests may need to be employed. To take the place of these tests, the CASE OF statement may be used. The following are equivalent:

```
CASE Digit OF
  '0' : Number := 0;
  '1' : Number := 1  END;

IF Digit = '0' THEN
  Number :=0 ELSE
IF Digit = '1' THEN
  Number :=1;
```

The Functions Ord, Pred, Succ, and Chr

Scalar type variables are declared in a particular order, or scale. Often the order of these items is of significance and can be used in a program. This is made possible by the functions Ord (order) and Pred (preceding), and Succ (succeeding). One example is the days of the week:

```
TYPE
 DaysWeek = (Sun,Mon,Tue,Wed,Thur,Fri,Sat);
```

The items in the list are called the values. Each item is an identifier (i.e., it must start with a letter followed only by letters or numbers).

The first value in the type Days is Sun. The seventh value is Sat. Thus both these statements are true:

```
Ord(Sun) = 0;
Ord(Sat) = 6;
```

The day succeeding Sun is Mon, and the day preceding Fri is Thur. Both these statements are true:

```
Succ(Sun) = Mon;
Pred(Fri) = Thur;
```

If two scalar types are declared, some of the items in the two lists will have the same ordinal value. For example, if the days of the week and the months of the year are declared, both Tue and Mar will have the ordinal value 2.

There is an ASCII character corresponding to every Integer from 1 to 128. The function Chr (Character) gives the ASCII character corresponding to an Integer specified in parentheses, e.g., "Chr(2)." This Integer may be the ordinal value of a scalar element. (However, Chr is not the inverse function of Ord.)

```
Chr(2) = STX;
```

STX is a nonprintable ASCII character used in some compilers to mark the start of a text file. The ordinal values corresponding to the characters 'A', 'B', '1', and '2' are shown below. The quotes around the characters denote that they are of the type Char and are not undefined variables or Integers.

```
Chr(65) = 'A';  Chr(66) = 'B';  Chr(49) = '1';  Chr(50) = '2';
```

PART II

PROGRAMMING TECHNIQUES

PROCEDURES

The following section explains a technique for breaking down long programs into simple and easy to understand modules called procedures. With a little rewriting, any procedure can be made into a program by itself.

Procedures may or may not communicate with the main program or other procedures. If they do, a list of parameters is generally declared. In the following example, the parameters are X1 and X2.

```
PROCEDURE ExchgVal(VAR X1,X2 : Real);
(*Values of X1 and X2 are exchanged*)

VAR
  Y : Real;

BEGIN
  Y := X1;
  X1 := X2;
  X2 := Y;
END;
```

Declaring and Executing PROCEDURES

The following outline lists the steps necessary in using the procedure ExchgVal in a program, Demo. The program is divided into three main sections:

The first section, the declaration part of the program, was discussed earlier.

The second section is the declaration of the procedure (or procedures).

The third section is the body of the program, where the procedure is actually used.

1. Declaration section of main program, Demo.
   - 1a. Declare program name.
     - 1b. If there were program constants or types to declare, they would be in this section.
     - 1c. Declare program variables, A, B.

2. Declare procedure, ExchgVal.
   - 2a. Declare procedure name and parameters.
   - 2b. If there were local constants or types to declare, they would be in this section.
   - 2c. Declare procedure local variable, Y.
   - 2d. Procedure body: the executable statements are declared here, but not executed.

3. Main program body.
   - 3a. Enter two numbers from keyboard: A, B.
   - 3b. Exchange A and B by executing procedure, ExchgVal.
   - 3c. Output numbers, A and B, to the screen.

```
PROGRAM Demo(Input,Output);
 (*Shows result of procedure ExchgVal*)

 VAR A, B : Real;

 PROCEDURE ExchgVal(VAR X1,X2 : Real);
 (*Values of X1 and X2 are exchanged*)
 VAR Y : Real;
 BEGIN
   Y := X1;
   X1 := X2;
   X2 := Y
 END(*Procedure ExchgVal*);

 BEGIN (*Demo*)
   Write ('Enter two numbers: ');
   Readln (A,B);
   ExchgVal (A,B);
   Writeln;
   Writeln ('Now first = ',A : 7:2, ' and second = ',B : 7:2)
 END.
```

Suppose the values to be exchanged are 5.8 and 11.15. The screen will show the following (user entries are underlined):

Enter two numbers: <u>5.8</u> <u>11.15</u>

Now first = 11.15 and second = 5.8

<u>Parameter Lists, Actual and Formal</u>

Because a procedure is a program within a program, there must be a way of getting data into and out of the procedure. In the above example, the variables X1, X2, A, and B provide this means. These variables are examples of parameters. Parameters may be variables, constants, and even other parameters.

When parameters are listed in parentheses after the procedure name in the declaration part of the program, as are X1 and X2, they are part of the <u>formal parameter list</u>.

```
PROCEDURE ExChgVal(VAR X1,X2 : Real);
```

When parameters such as A and B appear in parentheses after the procedure name in the body of the program, they are part of the <u>actual parameter list</u>.

```
ExChgVal(A,B);
```

Obviously, the formal parameters X1 and X2 are variables of the type Real, as are the actual parameters, A and B. Real numbers such as 4.3 and 6.7 may also have been used. Actual and formal parameters must match.

Although the formal parameter list is written within parentheses, it can be arranged to look more like the declaration section of a program. The following are identical:

```
PROCEDURE Calculate(A, B : Real; VAR X : Real; Y : Integer);

PROCEDURE Calculate(
  A, B : Real;

  VAR
   X : Real;
   Y : Integer);
```

Variable and Value Parameters

Notice that in the following formal parameter list, only some of the parameters are preceded by VAR. These are the variable parameters (i.e., X1, X2, Y). Variable parameters are used for both input to the procedure and output from the procedure. A value parameter, such as Z, is formal parameter that is not preceded by a declaration such as VAR, and can be used only to input data to the procedure:

```
PROCEDURE OtherVal(VAR X1, X2 : Real; Z : Real; VAR Y : Integer);
```

Although Z may change value during the execution of the procedure, the new value of Z is not communicated to the main program.

The following statements might occur within the body of the program when the procedure OtherVal is to be executed:

```
OtherVal(A, B, 5.0, D);
OtherVal(C, B, A/10.0, E);
```

Notice that arithmetic operators and values (such as Integers) can appear in a list of actual parameters if the corresponding parameter is a value parameter. An error is generated if the corresponding parameter is a variable one.

Correspondence Between Actual and Formal Parameters

The following rules must always be obeyed:

1) The number of actual parameters in each set of parentheses must be exactly the same as the number of formal parameters.

2) The parameter types must be consistent. Thus, the main program (which uses the procedure OtherVal) may declare:

```
VAR
  A,B,C : Real;
  D,E : Integer;
```

The names of the variables in a procedure may be the same as names used in other procedures or in the main program.

Functions

Functions are similar to procedures in that both use parameters, but different in that a function takes the values input (viz., the parameter values) and returns a single value which is identified by the function name. For example, the function Sqr(X) returns the value of X squared when given some value of X. Thus, when X equals 12, Sqr(X) equals 144.

A few of the most commonly used mathematical functions are included in Kyan PASCAL (X is a Real number or Integer):

```
Abs(X)    =  Absolute value of X
Sqr(X)    =  The square of X
Sqrt(X)   =  The square root of X
Sin(X)    =  The sine of X (X is in radians)
Cos(X)    =  The cosine of X (X is in radians)
Arctan(X) =  The arctangent of X (result is in radians)
Ln(X)     =  The natural logarithm of X
Exp(X)    =  e raised to the power X
```

Additional functions can be defined by the user.

Declaring Functions

A user-defined function is a simple procedure that uses only value parameters. The elements of a function are illustrated below. They include the function name, Cosine Law (CsLaw), the formal parameter list (A, B, Theta : Real), the result type (Real), the local declaration (VAR C : Real), and the function body (BEGIN...END).

```
PROGRAM Trig(Input,Output);
 VAR
  E,H1,W1,Ang1,AngX : Real;

   FUNCTION CsLaw (A, B, Theta : Real) : Real;
   (*Returns the length of side, C, opposite the angle Theta*)
   VAR
     C : Real;

   BEGIN
    C := A*A + B*B;
    CsLaw := C - 2.0*A*B*Cos(Theta)
   END;(*PROCEDURE*)

BEGIN
 Readln(H1,W1,Ang1,AngX);
 E := 1.0 + CsLaw(H1,W1,Ang1)*Sin(AngX)
END.(*PROGRAM Trig*)
```

Like value parameters in procedures, the parameters of a function do not change their values outside the function. The function returns only a single value, the result (CsLaw), whereas a procedure may return as many values as there are variable parameters listed.

When a function is used in a program, a separate statement to call it up is not required. For example, CsLaw can be called up by relational or arithmetic statements such as the following:

```
E := 1 + CsLaw(H1,W1,Ang1)*Sin(AngX);
```

A procedure, however, does require a separate statement [e.g., OtherVal(A,B,C,D);]. This is because the identifier of a function has some value, viz., the result, but the identifier of a procedure does not have a value.

The Function Odd

The function Odd(parameter) returns the value True when the parameter is odd or the value False when the parameter is even. It is important that the parameters used with Odd be of the type Integer.

For example, if the variable Number equals 3, then:

```
Odd(Number) = True
```

Thus, this function turns Integer data into Boolean data.

Global and Local Variables

When a variable is declared in the main program, it is called a global variable. When a variable is declared within a function or procedure, it is called a local variable. Parameters are neither local nor global variables, although they are used to pass values of global variables to and from the procedure.

```
PROGRAM Alpha(Input,Output);
  VAR A1    : Real;
      A3,A4 : Char;

  PROCEDURE Other (VAR AA1:Real; AA3:Char);
    VAR BB1:Integer;
    BEGIN
     A4 := 'Y';              (*A4 is global*)
     BB1 := 5;               (*BB1 is local*)
     AA1 := 15.3
    END;(*Procedure*)

BEGIN
  Other(A1,A3);          (*A1,A3 are parameters*)
  IF A4 = 'Y' THEN
  Writeln('A4 is global');
  IF A1 = 15.3 THEN
  Writeln('AA1 is a formal parameter')
END.(*Program Alpha*)
```

The statements in the body of a function or procedure manipulate a variety of variables and parameters. Variables must be appropriately defined in order for the program to function properly:

1) They can be declared in the global declaration section.
(VAR A1:Real; A3,A4:Char;)

A variable that has been declared in the main program may be used in a function or procedure in a global manner. The variable A4 is used in this way:

```
A4 := 'Y';
```

Every time the procedure Other is called, A4 is given the value 'Y' and the statement in the main program, A4 = 'Y', becomes true.

2) They can be declared in the local declaration section.
(VAR BB1 : Integer;)

A variable declared only in the procedure may be used. The variable BB1 is used locally in the program:

```
BB1 := 5;
```

Because BB1 was not declared in the main program, if the statement "BB1 = 5" were to appear in the main program, it would make no sense and the compiler would generate an error message.

3) They can be listed in the formal parameter section.
[Other (VAR AA1 : Real; AA3 : Char);]

Passing values through global variables is not recommended because it makes it difficult to keep track of incoming and outgoing data: it is better to use actual and formal parameters.

The following section extends the preceding definitions of global and local to more general cases where a variable is relatively global or relatively local. This occurs when there are several functions and procedures sharing variables.

Nesting of Functions and Procedures

Functions and procedures may be nested within other functions or procedures. The declaration section of a program is illustrated below with nested boxes to represent the concept called "scope." The innermost box, Phase1 is within the scope of both CsLaw and PhaseDis, while CsLaw is only within the scope of the main program, PhaseDis.

Because of the top-down structure of PASCAL, the procedures or functions declared first have greater scope than those declared later. Identifiers (of variables and types) in the outer boxes are global relative to the inner boxes. Identifiers that are declared in procedures of greater scope are global relative to procedures of lesser scope.

Thus, values of variables may be passed from a procedure of greater scope to one of lesser scope either by parameters or by global variables of the procedure of greater scope.

```
PROGRAM PhaseDis;
VAR Heightl, Widthl, Anglel, Angle2, Dist : Real;

  FUNCTION CsLaw(A,B,Theta : Real) : Real; VAR C : Real;

    PROCEDURE Phasel(Hl,Wl,Angl,AngX : Real; VAR D : Real);
    VAR E : Real;

    BEGIN
    E := 1 + CsLaw(Hl,Wl,Angl)*Sin(AngX);
    D := 1.22*C
    END;(*Phase Declaration*)

  BEGIN
   C := A*A + B*B;
   CsLaw := C - 2*A*B*Cos(Theta)
  END;(*CsLaw Declaration*)

BEGIN
 . .
END.(*PhaseDis*)
```

Notice how the scope of a variable is determined the moment it is declared and remains in effect until the end of the procedure, function or main program in which it was declared.

The scope of the variables can be represented more clearly by showing only the declaration sections of the program, functions, and procedures:

```
program: PhaseDis
 variables declared: Heightl, Widthl, Anglel, Angle2, Dist

  function: CsLaw
   variables declared
    (formal parameters): A,B,Theta
    (local variables): C

 procedure: Phasel
  variables declared
   (formal parameters): Hl, Wl, Angl, AngX, D
   (local variables): E
```

In this example, C is global to Phasel. The new value for C is passed to Phasel as soon as CsLaw is executed. Use of global variables in this way is not recommended. Values should be passed to and from functions and procedures only through parameters.

Compare the following version of the program PhaseDis to the previous one. The procedure Phasel is no longer nested within CsLaw. C is no longer global relative to Phasel because CsLaw no longer has greater scope than Phasel. The statement using C in Phasel had to be dropped, because it would no longer be syntactically correct.

It is possible, and often desirable, in a long program to reuse names in several places but with different meanings. As long as the scope of one definition of such a name does not not encompass another definition, there will be no conflict.

```
PROGRAM PhaseDis;
VAR Height1, Width1, Angle1, Angle2, Dist : Real;

  FUNCTION CsLaw(A,B,Theta : Real) : Real; VAR C : Real;
  BEGIN
  C := A*A + B*B;
  CsLaw := C - 2*A*B*Cos(Theta)
  END;(*CsLaw Declaration*)

  PROCEDURE Phase1(H1,W1,Ang1,AngX : Real);
  VAR E : Real;
  BEGIN
  E := 1 + CsLaw(H1,W1,Ang1)*Sin(AngX)
  END;(*Phase Declaration*)

BEGIN
  . .
END.(*PhaseDis*)
```

Global and Local Types

User-defined types such as scalar types may be local or global. The same rules of scope apply.

Forward References

Calling, i.e., executing, a procedure or function before it has been defined is called a forward reference. Whenever a forward reference is used in a PASCAL program, it must be declared as shown in the third line of the following program:

```
PROGRAM Compute(Input,Output);
VAR X : Integer; Y : Real;

  FUNCTION Factor(Z : Integer) : Integer; FORWARD;

     PROCEDURE Bisect(Alpha : Integer; Beta : Real);
     BEGIN
     Beta := Beta + Alpha*Factor(Alpha)
     END;(*PROCEDURE*)

FUNCTION Factor;
  CONST LargeNum = 12345;
  BEGIN
   Factor := LargeNum MOD Z
  END;(*FUNCTION*)

BEGIN
 Write('Enter an Integer ');Readln(X);
 Write('Enter a decimal number '); Readln(Y);
 Bisect(X,Y);
 Y := Factor(X)*Y;
 Writeln;Writeln('Answer is ',Y )
END.
```

The procedure Bisect is able to execute the function Factor because the latter is declared as a forward reference before Bisect is declared. Notice that the forward reference declaration includes the formal parameter list; later, when Factor is fully declared, the parameters and the FORWARD declaration are not repeated.

Unconditional Branch: GOTO

Although it is not ordinarily done, PASCAL statements may be labeled to allow unconditional branching, such as from a REPEAT UNTIL loop.

A label (i.e., statement number) in PASCAL is an Integer followed by a colon and placed before a statement in a program. The maximum size of a label is four digits. Labels must be declared just like variables and constants. The following statements might occur in a program with a forward jump:

```
PROGRAM Example(Input,Output);
LABEL 22, 35;
VAR  A : Integer;

BEGIN
      A := 0;
 22:  Writeln('A= ',A :4);
      A := A + 1;
      IF A < 5 THEN GOTO 22 ELSE GOTO 35;
      Writeln('Skip Me');
 35:  Writeln('The End')
END.
```

The unconditional jump, which may be either forward or backward in the program, is written as follows:

```
GOTO label;
```

Labels used in a function or procedure must be declared locally. GOTO jumps can be used to jump forward or backward within a function or procedure, or to leave a function or procedure to enter the main program, but cannot be used to jump from the main program to enter a function or procedure.

Most of the types of data that have been discussed so far are limited to single values. (Integer and Real both imply a single number; Char implies a single character; and Boolean is either the value True or False.)

However, some kinds of data are not conveniently divided into components. This is the case with words or strings, which were discussed previously. A string, such as "butter" is actually a collection of characters. This is the identifying characteristic of an array: an array is always a collection of one of the simpler data types.

A vector, such as the direction of a spaceship in flight, is another example of an array. The clearest and most correct way to handle such data is to put parentheses around the components (X, Y, Z) to clarify that they represent a single direction.

Arrays are declared in PASCAL as follows:

```
Array Type = ARRAY[Subscript Type] OF Element Type
```

1. Array types are always user defined.

2. The subscript type specifies the size of the array and assigns a number to each of the elements of the array. See examples below.

3. The element type may be any standard or user-defined type. All the elements in an array must be the same type.

The amount of memory space allocated for an array is determined by the subscript type. If an array of characters is not filled because the input is smaller than the array size, the remaining spaces are set to blanks. However, unused array spaces of other types are not determined.

Example Program:

```
PROGRAM Graphic;
TYPE
  String = ARRAY[1..15] OF Char;
  CoordnType = (X,Y,Z);
  VectorType = ARRAY[CoordnType] OF Real;
VAR
  Vector : VectorType;
  Word : String;
BEGIN
  Vector[X] := 3.0;
  Vector[Y] := 5.0;
  Vector[Z] := 4.0;
  Word := 'First Point    '
END.
```

The first array, String, may be used to handle words or phrases that have 15 characters, including blanks. Integers (1 to 15) identify the elements of the array.

The second array declares that each vector consists of 3 numbers. (Each one is a direction in three-dimensional space.) The elements of this array are identified not by Integers, but by CoordnType, a user-defined type. Any scalar type may index an array.

Arrays of Arrays and Multidimensional Arrays

If we wished to represent a paragraph that contained up to 50 words, we might define it as an array of String (i.e., an array of an array):

```
TYPE
  String = ARRAY[1..15] OF Char;
  Paragraph =  ARRAY[1..50] OF String;
```

Use of the array Paragraph could prove to be a wasteful programming technique because it reserves a lot of memory space for what might turn out to be a short paragraph.

The array Paragraph is an example of a multidimensional array. The array MatxType below is also multidimensional. MatxType is a two-dimensional array of numbers. (It is not necessary for the dimensions of the matrix to be the same size, although in this one they are, 3 elements each.)

Two ways of declaring MatxType are:

```
TYPE
  Row = ARRAY[1..3] OF Real;
  MatxType = ARRAY[1..3] OF ROW;
  (*Each element of MatxType is a row*)

TYPE
  MatxType = ARRAY[1..3, 1..3] OF Real;
  (*Subscripts are row number, column number*)
```

It is important to recognize which subscript refers to which dimension in such arrays. The significance of this is illustrated by the following example, in which a name, i.e., a string, is copied from a list:

```
TYPE
 String =  ARRAY[1..14] OF Char;
 TableType = ARRAY[1..100] OF String;
VAR
 Table : TableType;   Name : String;   I : Integer;
BEGIN
 FOR I := 1 TO 14 DO
  Table[2,I] := Name[I]
(*Name is written into the second row of table*)
END.
```

One way to remember which subscript is first is to rewrite the declaration of the array type. The first subscript type in the declaration below gives the first subscript, S, in Table[S,P]; the second subscript type gives the second subscript, P.

```
TYPE
 TableType = ARRAY[1..100] OF ARRAY[1..14] OF Char;
```

Adding Two Multidimensional Arrays

The following program adds two 3 X 3 matrices. To find the sum of two matrices, the corresponding elements (those with identical row and column subscripts) are added to form the elements of the sum matrix. In this program, the first matrix is entered in matrix form into the computer's memory. The elements of the second matrix are then added, one at a time, to the elements of the first matrix. Thus, the sum matrix is formed without the computer's ever having "seen" the second matrix.

```
PROGRAM AddMatrix(Input,Output);

TYPE
 MatxType = ARRAY[1..3,1..3] OF Real;

VAR
 Matrix : MatxType;
 SubSRow, SubSCol : Integer;
 (*Subscripts of the matrices*)
 AddEle : Real;
 (*Elements of 2nd Matrix*)

BEGIN
 FOR SubSRow := 1 TO 3 DO
  FOR SubSCol := 1 TO 3 DO
   BEGIN
   Write('Matrix1 element ',SubSRow : 3, SubSCol : 3, 'is ');
   Readln(Matrix1[SubSRow,SubSCol]);
   (*Inputs the elements of first Matrix*)
   END;(*FOR*)
FOR SubSRow := 1 TO 3 DO
 FOR SubSCol := 1 TO 3 DO
  BEGIN
  Write('Matrix2 element ',SubSRow : 3, SubSCol : 3, 'is ');
         Readln(AddEle);
  (*Inputs the elements of second Matrix*)
  Matrix[SubSRow,SubSCol] := AddEle + Matrix[SubSRow,SubSCol]
  END;(*FOR loops*)
Writeln;Writeln('The sum of the two matrices is:');
Writeln;
FOR SubSRow := 1 TO 3 DO
  BEGIN
  Writeln;
  FOR SubSCol := 1 TO 3 DO
  Write(Matrix[SubSRow,SubSCol] : 7:3)
  END;(*FOR*)
END.
```

The Array As a Parameter

In the procedure below (ShOrder), an array (SubArry) is used as a variable parameter:

```
PROCEDURE ShOrder(First, Last: Integer; VAR SubArry: NumbArray);
```

If SubArry were to be passed as a value parameter, VAR would be deleted. (But this would take twice as much memory space, because an extra copy of the array would be set up for use in the procedure.)

Individual elements of an array may also be passed as parameters, such as the third element of Vector, viz. Vector[Z]:

```
PROCEDURE CheckPoint (Vector[Z] : Real);
```

Program Example1

This program is used to order a small subset of a list of up to 150 numbers. Beyond six numbers in the subset, the procedure becomes inefficient.

The ordering of the subset is accomplished by the procedure ShOrder, which works as follows: pairs of elements in the subset are compared, starting with the first and second elements. If the first element is greater than the second, they are exchanged. This is repeated for the second and third elements, etc. As long as any exchanges have taken place anywhere in the list, this procedure will repeat again for the entire list. When no exchanges have taken place, the list is in order.

```
PROGRAM Example1(Input,Output);
CONST MaxNumbs = 150;
TYPE NumbArray = ARRAY[1..MaxNumbs] OF Real;
VAR First, Last, Subscript: Integer; BigArry: NumbArray;

PROCEDURE Exchg(VAR A,B: Real);
  VAR C: Real;
BEGIN (*Procedure Exchg*)
  C := A;
  A := B;
  B := C
END;(*Procedure Exchg*)

PROCEDURE ShOrder(First, Last: Integer; VAR SubArry: NumbArray);
(*Orders a list of numbers, subset of full list*)
VAR NumbIndex : Integer;
    Exchanged : Boolean;
BEGIN
  REPEAT
    Exchanged := False;
    FOR NumbIndex := First TO (Last-1) DO
    IF SubArry[NumbIndex] > SubArry[NumbIndex+1]
     THEN BEGIN (*Exchange if out of order*)
     Exchg(SubArry[NumbIndex],SubArry[NumbIndex+1]);
     Exchanged := True
     END;(*Exchg,THEN*)
  UNTIL Exchanged = False (*If one of the elements was exchanged,
  the test must be repeated until all elements are in order &
  Exchanged remains False*)
END;(*Procedure ShOrder*)

BEGIN(*Main Program*)
Writeln('Enter a list of numbers to be ordered.');
Writeln('After each number press the return key.')
Writeln('After last number enter 0 and press');
Writeln('return to stop.');
```

```
Subscript := 0;
REPEAT
 Subscript := Subscript +1;
 Write('Entry Number ', Subscript: 3, 'is');
 Readln(BigArry[Subscript])
UNTIL BigArry[Subscript] = 0.0;
Writeln('Between which "Entry Numbers" should');
Writeln('this list be ordered? First :');
Readln(First); Writeln('Last :');
Readln(Last);
ShOrder(First,Last,BigArry);
Writeln;
FOR Subscript := First TO Last DO
Writeln(BigArry[Subscript]: 7:3, ',Entry Number', Subscript: 3)
END.
```

PROGRAM Example2

In the following program, the procedure ShOrder is modified to sort a two-dimensional array. The new procedure ShAlph, is used to alphabetize a list of 6 words, each of which has no more than 15 letters. The procedure Exchg exchanges the position of two consecutive words in the array.

The two-dimensional word array used in this program can be visualized as follows:

```
help
program
difficult
easy
should
be
```

The first subscript gives the horizontal position of a letter; the second subscript gives the vertical position. Thus, the "r" in "program" would be subscripted (2,2), the "d" in "should" would be subscripted (6,5), etc.

```
PROGRAM Example2(Input,Output);

 CONST MaxLetters = 15; MaxWords = 6;
 TYPE String = ARRAY[1..MaxLetters] OF Char;
      WordArray = ARRAY[1..MaxWords] OF String;
 VAR WordMatrix: WordArray; WordIndex: Integer;

 PROCEDURE Exchg(VAR WordMatrix: WordArray; WordIndex: Integer);

 VAR C: String;
  BEGIN
   C := WordMatrix[WordIndex];
   WordMatrix[WordIndex] := WordMatrix[WordIndex+1];
   WordMatrix[WordIndex+1] := C
  END;(*Procedure Exchg*)

 PROCEDURE ShAlph(VAR WordMatrix: WordArray);
 (*Alphabetize word list <= 6 words*)
 VAR WordIndex: Integer; Exchanged: Boolean;
```

```
BEGIN
 REPEAT(*Until all words are in order*)
 Exchanged := False;(*All words are in order if none need to be
                    exchanged*)
  FOR WordIndex := 1 TO MaxWords - 1 DO IF
   WordMatrix[WordIndex] > WordMatrix[WordIndex + 1]
   THEN BEGIN (*FOR Loop, Test all in WordMatrix*)
    ExChg( WordMatrix, WordIndex);
    Exchanged := True;
   END;(*IF*)
  UNTIL Exchanged = False;
 END;(*Procedure ShAlph*)

BEGIN(*Main Program*)
Writeln('Enter six words, each with a maximum of');
Writeln('15 letters.  After each word press the');
Writeln('RETURN key. ');
WordIndex := 0;
REPEAT
 WordIndex := WordIndex + 1;
 Write('Word number ', WordIndex :3, 'is ');
 Readln(WordMatrix[WordIndex])
UNTIL WordIndex = MaxWords;

ShAlph(WordMatrix);
Writeln;
Writeln('Alphabetized Words: ');
FOR WordIndex := 1 TO MaxWords DO
 Writeln;
 Writeln(WordMatrix[WordIndex]);
END.
```

<u>End of Line</u>

The character that terminates a line of data on the keyboard is the end of line character, EOLN (<RETURN> key). The statements

```
Read (Letter);
IF EOLN THEN
```

may be used to control input from the keyboard, because the THEN statement will only be executed when a <RETURN> is entered. EOLN stays True until additional data are entered through a Read or Readln statement.

EOLN is used to control data entry below:

```
Writeln('Enter four words.  End each word');
Writeln('with the RETURN key ');
FOR WordIndex := 1 to 4 DO
BEGIN
LetterIndex := 0;
WHILE NOT EOLN DO
 BEGIN
 LetterIndex := LetterIndex + 1;
 Read(WordMatrix[WordIndex,LetterIndex])
 END;(*WHILE*)
```

```
Writeln('Preceding word had ', LetterIndex :3, 'letters.');
Readln
END;(*FOR*)
```

The above lines allow words to be entered, one letter at a time, into a list. Each EOLN signifies the end of a word, i.e., the end of a row in the array WordMatrix. If WordMatrix is declared to be of the size [1..4,1..15], when a word has fewer than 15 letters, the unused places will be filled with blanks. If a word is longer than 15 letters, the excess letters will not be saved.

Recursive Procedures and Functions

A procedure or function that calls itself is said to be recursive. In Progam Example2, it is possible to rewrite ShAlph to make it recursive. Typical of recursive procedures, ShAlph has fewer statements than before, but in the compiled machine code it will be longer.

```
PROCEDURE ShAlph (WordMatrix : Word Array);
VAR WordIndex : Integer;
BEGIN
 FOR WordIndex := 1 to MaxWords - 1 DO
 IF(WordMatrix[WordIndex}] > WordMatrix[WordIndex+1])
  THEN BEGIN
   ExChg(WordMatrix,WordIndex);
   ShAlph(WordMatrix)
 END
END;(*PROCEDURE*)
```

As rewritten, ShAlph tests the words from the first word to the last. If any of the words are out of alphabetical order, they are exchanged and ShAlph begins again. When ShAlph is called recursively, the index to the array is reset to the beginning.

There are two uses for recursion: 1) where logical decisions occur repetitively as above, and 2) when computing a function in the form of some repetitive function, such as N! = N*(N-1)*(N-2)* ...*[N-(N-1)].

Copying Arrays

If two arrays have the same subscript type and element type, the values of one may be copied to the other using a simple assignment statement. Notice that it is not necessary to specify the subscripts when copying.

```
VAR  Matrix1, Matrix2 : ARRAY[1..3,1..3] OF Real;

BEGIN   Matrix1 := Matrix2;
```

Values may be assigned to string array variables by using single quotes around the characters to be included. This is illustrated in the example below. Blanks are assigned because the string array size is larger than the word being put into it:

```
PROGRAM CopyArrays;
TYPE   String = ARRAY[1..16] OF Char;
VAR    Word1, Word2 : String;

BEGIN
 Word1 := 'Initial         ';
 Word2 := Word1;
 Word2[8] := 's';
 Writeln(Word2)
END.
```

In this program, Word2 is given the value "Initials" with eight blanks.

RECORDS

Some kinds of data are most conveniently handled as a mixture of several types. An example of mixed type data is the date: "January 1, 1987" is a string of characters followed by two Integers. PASCAL allows the user to define mixed data types as records:

```
TYPE
 DateType = RECORD
   Month: ARRAY[1..10] OF Char;
   Day: Integer;
   Year: Integer
 END;(*DateType*)

VAR
  DateRec: DateType;
```

DateType is the identifier of a record type with three fields, and DateRec is the identifier of a variable of the type DateType. The general form of the declaration of a record and its fields is:

```
TYPE
  Identifier = RECORD
   field1: type1;
   field2: type2;
   field3: type3;
         etc.
  END;
```

The last field in a record does not need to be terminated by a semicolon. The three fields in the record DateType are Month, Day, and Year.

The statement below is one way to refer to a record variable. It uses the form "identifier.field."

```
Writeln (DateRec.Year: 5:0);
```

Another way to refer to record variables is to use the WITH statement:

```
WITH DateRec DO
  BEGIN
    Readln(Month);
    Readln(Day);
    Readln(Year)
  END; (*WITH DateRec*)
```

All three record variables are read using the WITH format.

Copying a Record

If two records are of the same type, it is possible to use a simple assignment to transfer the value of one to the other:

```
VAR
 DateRec1, DateRec2: DateType;

BEGIN
 DateRec1 := DateRec2;
```

This copies all the fields of DateRec2 into DateRec1 without having to list them. In this case, the Boolean comparison below would have the value True.

```
DateRec2 = DateRec1
```

Program Absolute

Using data in the form of a record, it is easy to write a program to calculate the absolute value of a complex number. The formula for the absolute value is the same as that for the distance from a point to the origin.

```
PROGRAM Absolute(Input,Output);
 (*Finds the absolute value of a complex number*)

TYPE
 ComplexType = RECORD
  RealPart : Real;
  ImagPart : Real
 END(*ComplexType Record*);

VAR
 ComplexNum : ComplexType;     Abs : Real;

BEGIN WITH ComplexNum DO
 BEGIN
  Write('The real part = ');  Readln(RealPart);
  Write('The imaginary part = ');  Readln(ImagPart);
  Abs := Sqrt(Sqr(RealPart) + Sqr(ImagPart));
  Writeln;  Writeln(('Absolute Value = ', Abs: 10:2)
 END(*With*)
END.
```

Program ElapsedTime

In the program that follows, the approximate time elapsed since January 1, 1980 is computed. All months are assumed to be of equal length, 30 days, and all years are 365 days long.

```
PROGRAM ElapsedTime(Input,Output);
(*Since Starting Time*)

 CONST
  StartDay = 1;
  StartMonth = 1;
  StartYear = 1980;

 TYPE
  DateType = RECORD
   Day : 1..31;
   Month : 0..12;
   Year : Integer
 END;(*DateType Record*)
VAR
 B: Integer;
 DateRec: DateType;
 InMonth: ARRAY[1..3] OF Char;

BEGIN
 Write ('MONTH (upper case, first 3 lett) = ');
 Readln (InMonth);
 WITH DateRec DO
 BEGIN
  Write ('DAY = ');  Readln (Day);
  Write ('YEAR = ');  Readln (Year)
 END;(*WITH reads DateRec*)
 DateRec.Month := 0;
 IF InMonth='JAN' THEN DateRec.Month := 1;
 IF InMonth='FEB' THEN DateRec.Month := 2;
 IF InMonth='MAR' THEN DateRec.Month := 3;
 IF InMonth='APR' THEN DateRec.Month := 4;
 IF InMonth='MAY' THEN DateRec.Month := 5;
 IF InMonth='JUN' THEN DateRec.Month := 6;
 IF InMonth='JUL' THEN DateRec.Month := 7;
 IF InMonth='AUG' THEN DateRec.Month := 8;
 IF InMonth='SEP' THEN DateRec.Month := 9;
 IF InMonth='OCT' THEN DateRec.Month := 10;
 IF InMonth='NOV' THEN DateRec.Month := 11;
 IF InMonth='DEC' THEN DateRec.Month := 12;
B := (DateRec.Day - StartDay)+ 30*(DateRec.Month - StartMonth) +
     365*(DateRec.Year - StartYear);
IF DateRec.Month = 0 THEN
 Writeln('Format error in Month') ELSE
 Writeln('Days since Starting Time = ' B: 8)
END.
```

In this program the record type contains three sets of Integers. Only a subset of the type Integers is used for Month and Day. In the range of values for Month, 0 is included to check that a three-letter abbreviation is input correctly. Error-checking statements are always part of a professionally written program.

The record field DateRec.Month could have been the scalar type (JAN,FEB..DEC), but this would not have enabled a direct comparison with the input, which is in the form of strings: 'JAN', 'FEB'..'DEC'. This is because 'JAN', a string, is not equal to JAN, an identifier.

Arrays of Records

Suppose the quality control department of a company wished to calculate the failure rate of each of thek parts in a gearbox or some other machine. Although 5000 of these machines were built, only 500 of them have come back for service. The first step in such a program would be to declare an appropriate array of records.

To construct an array of records, the following format is used:

```
TYPE
 Array Type = ARRAY[Subscript Range] OF Record Type;

VAR
 Array Variable : Array Type;
```

A particular element in such an array can be specified as follows:

```
Array Variable[Subscript].Field
```

In the program below, "Failures," the array is a set of records consisting of the serial number (Serial Num), the gear number(Gear), the date of failure (FailDate) and the date the gearbox was put into service (StartDate).

"Failures" is used to calculate the time between when the unit was placed in service and when a part failed. SurviveTime is a function similar to the program ElapsedTime, which was previously discussed.

```
PROGRAM Failures(Input,Output);
CONST
  GearCount = 50; (*50 parts in gearbox*)
  FailCount = 500;
  MachCount = 5000;

TYPE
  DateType = RECORD
   Month: 0..12; Day: 0..31; Year: Integer
  END;(*RECORD*)

  FailType = RECORD
   SerialNum: Integer;  Gear: Integer;
   FailDate: DateType;  StartDate: DateType
  END;(*RECORD*)

VAR
  Failure: ARRAY[1..FailCount] OF FailType;

FUNCTION
 SurviveTime( VAR FailDate, StartDate: DateType): Integer;
 BEGIN
  SurviveTime := (FailDate.Day-StartDate.Day) +
                 30*(FailDate.Month-StartDate.Month) +
                 365*(FailDate.Year-StartDate.Year)
 END;(*FUNCTION*)
```

```
BEGIN
 Writeln('The first gearbox to fail lasted ',
    SurviveTime(Failure[1].FailDate, Failure[1].StartDate): 5);
 Writeln('It was serial #', Failure[1].SerialNum: 9)
END.(*PROGRAM*)
```

The first Writeln statement specifies two fields (FailDate and StartDate) of the first record of the array Failure:

```
SurviveTime(Failure[1].FailDate, Failure[1].StartDate)
```

This statement calls the function SurviveTime, which then calculates the time to failure of the first machine.

WITH

Some programs that use records can be made more compact by using the WITH statement to access the fields of the array:

```
PROGRAM Entry;

 TYPE InputType = RECORD
 Money: Real; Name: ARRAY[1..15] OF Char;
 MonDate, DayDate,YearDate: Integer
 END;(*RECORD*)

 VAR InputVar: InputType;

BEGIN WITH InputVar DO BEGIN
 Money := 25.50;   Name := 'Full Moon Inc. ';
 MonDate := 4;  DayDate := 30;  YearDate := 1952
 END;(*WITH*)
Writeln(InputVar.Name, ' gave', InputVar.Money: 6:2, ' on',
        InputVar.MonDate: 3, '/', InputVar.DayDate: 3, '/',
        InputVar.YearDate: 3)
END.
```

The WITH statement is used with record variables, such as InputVar in the program above. It allows the fields of a record to be accessed without repeating the name of the record.

Variant Records

In many applications of the record type, there are two or more records that have most, but not all, their fields in common. The variant record is constructed for such cases.

For example, an auto repair shop owner wishes to keep a record of each repair in order to bill his customers later. His customers are either individuals or companies. In both cases, he wants to know the labor and parts used as well as invoice number and customer's name and address. In the case of companies, he also wants to know their requisition number. The two records are nearly the same:

```
TYPE              Invoice1 = RECORD
 InvoiceNum, Labor, Parts: Integer;
 CusName,CusAddr: String;
 SocSec: String
 END;(*RECORD*)

TYPE              Invoice2 = RECORD
 InvoiceNum, Labor, Parts: Integer;
 CusName,CusAddr: String;
 ReqNum: Integer
 END;(*RECORD*)
```

For convenience, these may be combined into a single variant record, named Invoice, by use of the CASE statement:

```
TYPE
  Invoice = RECORD
  InvoiceNum,Labor,Parts: Integer;
  CusName,CusAddr: String;
  CASE Custmr: Integer OF
    1 : (ReqNum: Integer);
    2 : (SocSec: String)
  END;(*RECORD*)
```

Suppose a variable of type Invoice is named Bill:

```
VAR  Bill: Invoice;
```

Then, to refer to the billing number of either an individual or company, the auto shop would use "Bill.InvoiceNum." To refer to the requisition number of a company, "Bill.ReqNum" would be used. The latter number does not exist for individuals.

SETS

Another type of data that may be declared in PASCAL is the set type. Sets may have up to 256 members. The general format for the declaration of a set type is:

```
Identifier = SET OF Base Type;
```

The base type must be a scalar type (but not Real).

For example, if the numbers from 10 to 25 are the base type, the prime and nonprime numbers from 10 to 25 are two possible set variables:

```
PROGRAM ExSet1(Input,Output);
 TYPE
  NumType = SET OF 10..25;
 VAR
  Prime, NotPrime: NumType;
  N: Integer;

BEGIN
  Prime := [11,13,17,19,23];
  NotPrime := [10,12,14,15,16,18,20,21,22,24,25];
  Write('Enter a number between 10 and 25 '); Readln(N);
  IF N IN Prime THEN Writeln('That is a prime') ELSE
  IF N IN NotPrime THEN Writeln('Not a prime') ELSE
  Writeln('That is not between 10 and 25)
 END.
```

Notice that the declaration does not specify what numbers constitute the set variables, only that they must be some set of Integers between 10 and 25. The numbers constituting the variables are assigned in the program as shown.

There are many similarities between a set type and a scalar type. In fact, the scalar type Numbers has the same range of values for its elements as the elements (members) of the set type NumType:

```
TYPE
 Numbers = 10..25;
 NumType = SET OF Numbers;
VAR
 Prime : NumType;
 NotPrime : NumType;
 Prim : Numbers;
 NotPrim : Numbers;
```

Numbers differs from NumType in that the elements in a scalar type are ordered, whereas they are not ordered in a set. (This allows the declaration of subranges of scalar types such as <u>TYPE FirstQt : Jan..Mar</u>, a subrange of <u>TYPE Year</u>.)

The differences between Prime and Prim are twofold:

1. Scalar variables such as Prim can have only one value at a time, whereas set variables can include 0 to 256 values.

2. The <u>set operations</u> may be applied to Prime but not to Prim.

For example, if all the prime Integers and all the non-prime Integers from 10 to 25 were to be listed, the list would be exactly all the Integers from 10 to 25. This is equivalent to the PASCAL statement below, where FullSet has been declared to be of the type NumType:

```
FullSet := Prime + NotPrime
```

Operations on Sets

There are three basic operations on sets: Union, Intersection, and Difference. Consider the following program:

```
PROGRAM Assign;
TYPE
  Numbers = SET OF 1..9;
VAR
  Prime : Numbers;
  Odd : Numbers;
  Test : Numbers;
BEGIN
  Prime := [1,2,3,5,7];
  Odd := [1,3,5,7,9];
```

The two sets Prime and Odd may be combined in three ways:

```
Test := Prime + Odd;(*Union = [1,2,3,4,5,7,9]*)
Test := Prime * Odd;(*Intersection =[1,3,5,7]*)
Test := Prime - Odd;(*Difference =[2]*)
```

In addition to the three basic set operators, there are seven set relational operators. These result in either a True or a False output and are exactly parallel to the arithmetic relational operators previously discussed.

```
Equality       Set1  =   Set2
Inequality     Set1  <>  Set2
Subset         Set1  <=  Set2
Superset       Set1  >=  Set2
SetMembership  Set1  IN  Set2
```

The set membership operator is True if the element is a member of SET1.

It is not necessary to declare a set type to use sets. The following program uses the set of failing grades : [F,NP].

```
PROGRAM Finals(Input,Output);
CONST ClassSize = 30;
TYPE
 GradeType = (A,B,C,D,F,P,NP,I);
 StuGrade = RECORD
   StudentID : Integer;
   Grade : GradeType
 END;(*RECORD*)
VAR
 ClassGrades : ARRAY[1..ClassSize] OF StuGrade;
 N : Integer;
 LettGra : ARRAY[1..2] OF CHAR;
BEGIN
 FOR N := 1 TO ClassSize DO
 BEGIN Write('Student ',ClassGrades[N].StudentID,'Enter final grade= ');
 Readln(LettGra);
 IF LettGra = 'F ' THEN ClassGrades[N].Grade;=F;
 IF LettGra = 'NP' THEN ClassGrades[N].Grade;=NP;
 IF ClassGrades[N].Grade IN [F,NP] THEN
   Writeln('Too Bad') ELSE Writeln('Good!')
 END (*FOR*)
END.
```

Using Sets to Examine the Members of an Array

In the program below, the set operator "IN" is used to examine the members of an array that contains all of the test scores of a student in a mathematics class. This process is repeated for all of the students in the class.

By outputing the total number of tests failed or postponed by students in the class, the program aids in evaluating overall class performance.

```
PROGRAM TestGrades(Input,Output);
CONST ClassSize = 30;
TYPE
  GradeType = (A,B,C,D,F,I,P,NP);
  GradeSet = SET OF GradeType;
  StuGrades = RECORD
   StudentID: Integer;
   Grades: ARRAY[1..25] OF GradeType
  END;(*RECORD*)
VAR
  ClassGrades: ARRAY[1..ClassSize] of StuGrades;
  N,M,I: Integer;

BEGIN
  (*Statements would be inserted here to input class grades*)
  I := 0;
  GradeSet := [F,NP,I];
  FOR N := 1 TO ClassSize DO FOR M := 1 TO 25 DO
  IF ClassGrades[N].Grades[M] IN GradeSet THEN
  I := I + 1;
  Writeln('In this class ', I:3, 'tests were');
  Writeln('either failed, not passed or put off')
END.
```

FILES

In PASCAL, files are the means of input and output of data. A Read or Readln statement calls for input; a Write or Writeln statement produces output. When information is entered at the keyboard, it goes into a PASCAL file called Input. When information is output to the display, it is sent to a file called Output.

Files can also be used by the programmer as a data type. What programmer-created files have in common with Input and Output files is that they are sequential and that information is read to them and written from them one element at a time.

Programmer-created files enable data to be stored on magnetic tape or floppy discs: a piece of data assigned to a file variable can be saved. In contrast, when files are not used, values assigned to variables are stored in the computer's directly accessible memory, which is lost when the power is turned off.

Since the data in a file is on magnetic tape or a floppy disk, the size of the computer's directly accessible memory is not a limitation as it is in other data types: files give the programmer a great deal of extra memory to work with. However, working with files has the disadvantage of slowing the access time. This can become critical in real time programs.

Files are unique in that they are the only completely sequential data type. In fact, data items stored in a file cannot be used in a program until they are transferred sequentially, one element at a time (first to last), from the file.

File Declaration

Before a file can be used in a program, it must be declared. The first step in declaring a file is to specify the file name in parentheses after the program name:

```
PROGRAM Store(Input, Output, Listl);
```

This tells the computer that at least some of the data used by the program will come from a file other than Input.

Next, in the variable declaration section, the variable type comprising the file is specified:

```
VAR Listl : FILE OF Integer;
```

Here, the elements of the file Listl are Integers. A file may also contain characters or Real numbers. Arrays, sets, and records made of Integers, characters, or Real numbers are also allowed.

Writing to a File

In order to store data in a file, we first must open the file for writing. This is done by using the Rewrite statement, which also clears the file of any data previously stored in it.

```
Rewrite (List1);
```

To actually put data into the opened file, two more statements are necessary:

```
List1^ := J;
Put(List1);
```

"List1^" (which is identical to "List1↑") is called a file buffer variable. Before the value of an element can be Put into a file, it must be temporarily assiged to a file buffer variable. The first statement above assigns the value J to the file buffer variable.

The Put statement is used to write the value J from the file buffer into the file. The first value entered goes to the first element position. The next value entered goes to the second element position, and so on. It is impossible to read from or write to a file without starting from the first position.

The only memory space reserved for file variables is for the file buffer variable. (If List1 is a file of Integer, List1^ will be assigned two bytes.) This is because a file exists outside the memory space of the computer.

Program Store(List)

The following program stores the Integers 11 to 945 in a file named List1:

```
PROGRAM Store(Input,Output,List1);

VAR
 List1 : FILE OF Integer;
     J : Integer;

BEGIN
Rewrite (List1);
FOR J := 11 TO 945 DO
 BEGIN
 List1^ := J;
 Put(List1)
 END
END.
```

Reading a File

Before a file can be read, it must be opened for reading. This requires use of the command Reset:

```
Reset (List1);
```

The first element read is always the first element that was entered: just as a file must be written from beginning to end, it must be read sequentially. In addition, reading requires that the values of the elements be assigned to a file buffer variable:

```
J := Listl^;
```

A Get statement must be used to get all elements of the file after the first one. Thus, if J is the first element of the file, K is the second, and L is the third, the following statements get the first three elements:

```
J := Listl^;
Get(Listl);
K := Listl^;
Get(Listl);
L := Listl^
```

Usually the number of elements in a file is not known; therefore, to get all the data from a file, the file should be read until the end of file marker (EOF) is found. The end of file marker is always at the end of the file furthest from the first element. The following statements write all the elements of the file.

```
Reset (Listl);
WHILE NOT EOF(Listl) DO
BEGIN
 J := Listl^; Writeln(J); Get (Listl)
END;
```

Sometimes a specific element in a file is sought. The following statements find and write all the elements of the file equal to 77 (Listl^ is the file buffer variable.):

```
Reset (Listl);
WHILE NOT EOF(Listl) DO
BEGIN
 IF 77 = Listl^ THEN Writeln(Listl^ :4);
 Get(Listl)
END;
```

<u>Text Files</u>

Because files of characters are so frequently used, PASCAL has a standard type of file, called Text, that is predefined as "Text = FILE OF Char." To create a file of text, include the file name after the program name and also declare it as a variable:

```
PROGRAM WordProc(TextFileName);

VAR  TextFileName : Text;
```

Although the input and output of a text file may be handled in the same way as the input and output of other types of files, the following simplifications may be used:

```
Read(TextFileName, Identifier);
```

can take the place of

```
Identifier :=TextFileName^;  Get(TextFileName);
```

Also:

```
Write(TextFileName, Identifier);
```

can take the place of

```
TextFileName^ :=Identifier;  Put(TextFileName);
```

If no text file name is included in a Read or Write statement, the file accessed will be Input or Output, respectively. (Note: some PASCAL compilers will give an error message if no text file name is given and Input and Output have not been declared.)

Files of Records

Most files are files of records. In the following example, the status of each truck in a company's fleet is kept in a file called BigRigFile.

```
PROGRAM Trucks(Input,Output,BigRigVar);
TYPE
 String = ARRAY[1..16] OF Char;
TruckType = Record
 NextSrvc : Integer;
 ID : String;
 Status : (OnRoad, MachShop)
END;(*TruckType*)
BigRigFile = FILE OF TruckType;

VAR
 BigRigVar : BigRigFile;
        S1 : String;

BEGIN(*Body of Trucks Program*)
Reset(BigRigVar);
WHILE NOT EOF(BigRigVar) DO
 BEGIN
  BEGIN
  Writeln;Writeln('Truck #', ID);
  IF BigRigVar^.Status = OnRoad THEN
    S1 := 'On the road     '
  ELSE S1 := 'In the shop      ';
  Writeln('Status is ', S1, 'Next Service is  ',
            NextSrvc : 7)
 Get(BigRigVar)
 END (*WHILE LOOP*)
END.
```

Random Files - ADVANCED KYAN PASCAL ONLY

Although standard PASCAL does not include random access files, there are many instances where a program might wish to access only part of a file and that part might be in the middle of a file or at the end, making sequential access very slow.

Most files in Advanced Kyan Pascal have been changed from sequential storage to relative storage to allow random access of the elements in the file. (However please note that files of Char (text) or files of Boolean, remain sequential files.) The elements of a relative file must be less than 128 bytes in size.

The function Seek has been included in Kyan PASCAL to access parts of relative files, called elements that might be in the anywhere in a file. This procedure is used as follows:

```
Seek(F,N);     (* Position the buffer of file F at the Nth element*)
Put(F)         (* Put contents of the file buffer into Nth element*)
   (* Either Put or Get follow Seek *)
Get(F);        (* Get contents of Nth file element and put in buffer*)
```

The first element of a file has the element number 1. As was stated previously the first element of a file is the first element put (using the PASCAL procedure Put) into a file. Most often, the elements of a file are PASCAL record types. In that case the 1st element of the file is its 1st record and its Nth element is its Nth record.

```
PROGRAM SeekDemo;
TYPE String = ARRAY[1..32] OF Char;
VAR F: FILE OF String; C: Char;

PROCEDURE RdRec;
VAR i: Integer;
BEGIN
 Write('Record Number? ');   Readln(i);
 Seek(F,i);
 Get((F);
 IF NOT EOF(F) THEN Writeln(F^)   (* EOF is true if element empty *)
END;

PROCEDURE WrRec;
VAR i: Integer;
BEGIN
 Write('Record Number? ');   Readln(i);
 Write('Data? ');   Readln(F^);     (* assign data to file buffer *)
 Seek(F,i);
 Put(F)
END;

BEGIN
 Reset(F,'DATA');
 REPEAT
  Writeln('R-Read W-Write Q-Quit ');  Readln(C);
  IF C='R' THEN RdRec;    IF C='W' THEN WrRec;
 UNTIL C='Q'
END.
```

If it were desired to open a file for the first time, or to clear an existing file of all data the procedure Rewrite would have been used. The above program assumes the file "Data" may have useful information in it, so the procedure Reset is used to open it instead.

Pointers and Nodes

```
VAR   Count : Integer;

BEGIN Count := 54;
```

If we could examine the computer's memory, we would find that the above statements put 54 into specific memory locations. Just for the sake of this discussion, assume that 54 goes into memory locations 12156 and 12157.

```
Count =  5  12156
         4  12157
```

There is another way to get 54 into memory and that way is to use pointers:

```
VAR Locate : ^Integer;

BEGIN New(Locate);
      Locate^ := 54;
```

If we could now examine the computer's memory, we would again find 54 in specific memory locations, perhaps 11343 and 11344. We would also find that the value 11343 is stored in memory:

```
Locate  =  1  11338
           1  11339
           3  11340
           4  11341
           3  11342
Locate^ =  5  11343
           4  11344
```

Locate is the group of memory locations that point to the place in memory where 54 is stored. There was no such "pointer" in the first example.

Locate is called a pointer variable, while Locate^ is called stored data or a node. The pointer symbol (^ or ↑) appears on the left side of the Type in the pointer variable declaration (Locate : ^Integer), but on the right of an identifier for stored data (Locate^ := 54).

It is also possible to declare pointer types such as:

```
TYPE LocateType = ^Integer;
```

New

New is the standard PASCAL procedure used to assign memory locations to a pointer variable. Each time the New statement is executed, a new set of locations is assigned to Locate.

If we deleted the New statement from the example above, the computer might put 54 into memory locations occupied by other data. This would probably cause run time errors.

Peek and Poke

Although New allows us to put data into memory, we have no idea where in memory the data is going. Peek and Poke give us the power to examine or change the data in specific memory locations.

Peek and Poke are most often used with memory locations that have a dedicated function such as specifying a character on the screen, the color of a character, or a sound emitted from the speaker.

Suppose we wish to check what actually is in memory locations 11343 and 11344:

```
VAR Locate : ^Integer;

BEGIN Assign(Locate, 11343);
      Write(Locate^);
      Assign(Locate, 11344);
      Write(Locate^);
```

Although "Assign(Locate, 11343)" is not part of standard Pascal, it is included in Kyan PASCAL. When the Assign statement is used with a Write statement, the result is a Peek.

In standard PASCAL it is not possible to decide where in memory to store data. The compiler makes that decision. However the Kyan PASCAL Assign statement allows us to Poke data into a specific memory location as follows:

```
VAR  Locate : ^Integer;

BEGIN Assign(Locate, 11343);
      Locate^ := 5;
      Assign(Locate, 11344)
      Locate^ := 4;
```

In the Commodore 64, location 1024 maps the first character on the screen. The screen is 40 characters wide by 25 characters tall. The following program uses Poke to put white periods into a rectangular area from the top to the middle of the screen.

```
PROGRAM Dots;
CONST  NRow = 25;  NCol = 40;
TYPE Screen = ARRAY[1..NRow,1..NCol] of Char;
VAR Colormem, Charmem : ^Screen; I,J : Integer;

BEGIN
 Assign(Charmem, 1024);
 Assign(Colormem, -10240);
 FOR I := 1 TO 12 DO FOR J := 1 TO NCol DO
  BEGIN  Charmem^[I,J] := '.';
    Colormem^[I,J] := Chr(1)
  END
END.
```

In the Poke above, an Integer value (1024) is assigned to a pointer variable (Charmem). This is where in memory Charmem^[1,1] will be stored. To Poke "." into the specified memory location, we assign it to Charmem^.

The procedure New is not used with a Poke: the next memory location (1025) is automatically mapped to the next Charmem^ in the loop. Each element in the array, Charmem^[I,J], takes one memory space; thus, the entire array is mapped into memory locations 1024 to 1503.

Suppose Charmem is defined as above, but now we wish to Peek at the character displayed at the upper left hand corner of the screen:

```
BEGIN
 Assign(Charmem, 1024);
 Write(Charmem^[1,1])
END.
```

Some memory locations exceed 32767, the maximum Integer size allowed in Kyan PASCAL. In those cases, the equivalent memory location is a negative number

Equiv. Mem. Loc. = Mem. Loc. - 65536

Linked Lists and NIL

In addition to being used with Peek and Poke, pointers are used in linked lists, which allow a database to be of variable size.

Below is a program that has a pointer variable, Appointm, which points to the location of a Record, AppointRec (just as Locate pointed to the location of an Integer). Each record is an appointment including time and person to meet.

```
PROGRAM Meetings(Input,Output);
TYPE String = ARRAY[1..15] OF Char;
 TimeType = (Hr,Min,Day,Mon,Yr);
 AppointRec = RECORD
  Person : String;
  Time : ARRAY[TimeType] OF Integer
 END(*RECORD*);
VAR  Appointm : ^AppointRec;

BEGIN
 New (Appointm);
 Appointm^.Person := 'Ernie          ';
 New (Appointm);
 Appointm^.Person := 'Bob            ';
 New (Appointm);
 Appointm^.Person := 'Gina           ';
 Writeln (Appointm^.Person)
END.
```

It is important to notice that each time another name is entered, the pointer is moved to a <u>new</u> location:

```
New(Appointm);
```

Although the above sequence of statements inputs three names into memory, each with a different pointer, it does not provide for retrieval of any of the names except the last. When Writeln is executed, "Gina" will be printed.

In the example that follows, a pointer type, Appointer, is declared; and the appointment record includes a pointer, Link, that will link all the records and thus allow all the data to be retrievable:

```
PROGRAM Pointer(Input,Output);
TYPE  String = ARRAY[1..15] OF Char;
 TimeType = (Hr,Min,Day,Mon,Yr);
 Appointer = ^AppointRec;   (*Pointer Type*)
 AppointRec = RECORD
  Link : Appointer;
  Person : String;
  Time : ARRAY[TimeType] OF Integer
 END;(*AppointRec RECORD*)

VAR   Appointm, Pt : Appointer;  (*Pointer Variables*)

BEGIN
 Pt := NIL;
 New(Appointm);
 Appointm^.Person := 'Ernie          ';
 Appointm^.Link := Pt;

 Pt := Appointm;
 New(Appointm);
 Appointm^.Person := 'Bob            ';
 Appointm^.Link := Pt;

 Pt := Appointm;
 New(Appointm);
 Appointm^.Person := 'Gina           ';
 Appointm^.Link := Pt
END.
```

The list of appointments is now retrievable because the "next" pointer (i.e., the linking pointer) is included in each record as the pointer field, "Link."

The standard PASCAL identifier NIL is used to indicate the last element in the list. Records are linked backward (first in = last out). NIL indicates the last element to be retrieved:

```
Pointer^.Link := NIL;
```

In the program above, Appointm points to the first name to be retrieved, Appointm^.Link points to the second, and (Appointm^.Link)^.Link (NIL) points to the third. The following statements output the names contained in the three linked records:

```
WHILE Appointm <> NIL DO
 BEGIN
  Writeln(Appointm^.Person);
  Pt := Appointm^.Link;
  Appointm := Pt
 END;(*WHILE*)
```

Dispose - ADVANCED KYAN PASCAL ONLY

When pointers and lists are created by the procedure New, they remain in memory even after the list to which they point is no longer used and all the elements on the list have been removed.

The following statement frees the memory location at Appointm^. It must be used for each of the elements of the list if all the memory locations on the list are to be freed:

```
Dispose(Appointm)
```

THE ASSEMBLED PROGRAM AND ITS USES (ADVANCED KYAN PASCAL ONLY)

How to Include Procedures and Functions from Other Files

Kyan PASCAL facilitates the inclusion of a user defined library of procedures and functions during compilation time. That is, procedures and functions that are used in many programs may be declared each in a file of its own and easily included for use in many programs.

To include a function or procedure in a program use the following format:

```
#i FileName
```

A pound sign (#) must appear in column 1 and i (for include) in column two. (Note: Use #I if in upper case mode and #i if in upper case and lower case mode>.) The name of the file (in which the declaration of the function or procedure is written) follows.

For example, the program HELLO was discussed in the Editor and Compiler chapter at the beginning of this book. Written as a procedure the file Hello would be:

```
PROCEDURE Hello;
BEGIN
 Writeln('Hello, world')
END;
```

The file Hello may be be included in any program by using the format just discussed:

```
PROGRAM Main;
#i Hello
BEGIN
 Hello;
END.
```

Use the same name for the procedure or function as the file name. Although it is possible to use different names, such would be poor style.

Including Files, Other Applications

Files that are included may be any text file, not just procedures and functions. It is important to try to visualize the insertion of the lines of the included file in place of the #i "FileName" line.

Assembly Language Routines

Kyan PASCAL accepts in-line assembly code, which enables the user to create many powerful routines and not be limited by the structure of standard PASCAL. In-line assembly routines do have one restriction though: they must appear in the body of the program, procedure, or function, i.e. they must appear between the BEGIN and END.

Some distinction must be made if the computer is to tell whether or not to interpret the lines that follow as assembly language or PASCAL. Assembly language lines are simply left as they are during compilation.

If the lines that follow are to be in assembly language they should begin with the pound sign (#) in column 1 and the letter "A" (or "a" if in upper/lower case mode) in column 2. End the assembly language lines with the pound sign in column 1. For example the procedure Delay is written with in-line assembly language:

```
PROCEDURE Delay;
BEGIN
#A
         LDY #100     (* IMPORTANT !!! *)
WLOOP    DEY          (* LABELS MUST START IN COL. 1 *)
         BNE WLOOP    (* ONLY LABELS START IN COL. 1 *)
#
END;
```

It is important not to use labels in the assembly language routines that begin with the letter "L." The compiler uses the labels Lxxxxx (xxxxx is a number) and if you use labels that begin with L, it is likely to fail.

Assembler Directives

Assembler directives are also known as pseudo-code because they appear in the assembly language listing of a program but are not part of the language of the microprocessor. Instead they are part of the language of the assembler.

Kyan PASCAL has six assembler directives. (They must not start in column 1 because they would be mistaken for labels.)

ORG	origin
EQU	equate
DB	define byte
DW	define word
>	least significant byte
<	most significant byte

ORG is used to tell the assembler that the following code is to start at the specified memory location.

When a label is given a value using the directive EQU that value will be substituted for the label throughout the program when the program is converted to object code. Anotherwords EQU defines constants.

DB and DW are used in building tables and strings that reside in certain parts of the assembly code. When the program executes, the values placed by these directives may be read by setting the index register to the address in program where the DB or DW statements are, then loading the value at the index register.

When the > and < operators are used with a label or immediate value in a program, either the least significant byte or most significant byte is extracted. For example, the following equalities are true:

```
>$FF01 = $0001
<$FF01 = $00FF
```

Parentheses are not allowed in assembler directives. Expressions are evaluated from left to right. There is no precedence of one directive over another.

How to Use Assembly Routines to Modify Pascal Variables

If the above assembly language routine were to be inserted into a PASCAL program it would cause a delay every time the program came to the place in which it was inserted. It does not modify any of the values of the variables in the PASCAL program.

In order to use assembly code to modify PASCAL variables, the location of these variables must be known. These locations are never absolute, but always relative to a pointer maintained by the compiler called LOCAL. The location of PASCAL variables may also be calculated relative to the stack pointer (SP).

In the example that follows in-line assembly code puts the value of the PASCAL variable "Cee" into the A accumulator of the microprocessor:

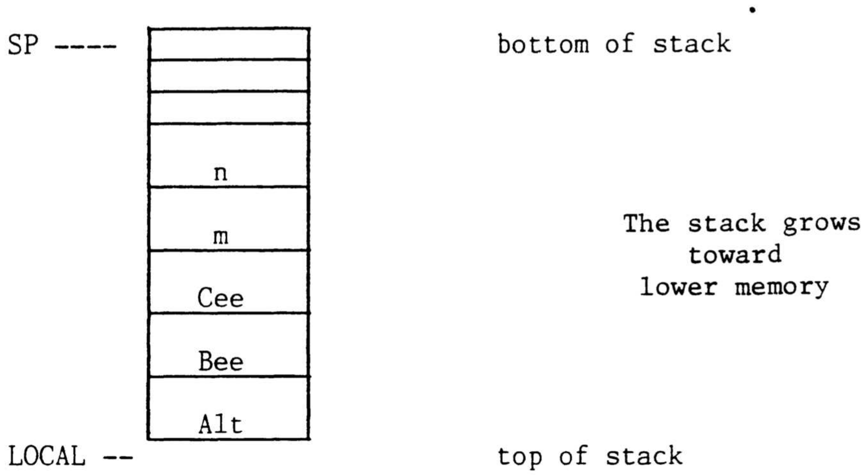
```
PROCEDURE Zen(Alt,Bee,Cee : Integer);
VAR
 m,n : Integer;
 BEGIN
#a
      LDY   7
      LDA   (SP),Y
#
END;
```

The first line loads the Y accumulator with 7 the distance that Cee is from SP. (The first variable declared is the first one on the stack and the one furthest from SP.) The offset from SP is calculated by adding 3 to the space taken by variables following the declaration of Cee:

Offset(of Cee above SP) = 3 bytes + Last-in Stack bytes

Since 2 bytes are required for each integer variable and both "m" and "n" are pushed on the stack after Cee, the total offset is 7 = 3 + 2*2.

The second line of the assembly code loads the accumulator A with the value in memory that is stored at where SP is pointing plus 7:



The offset from LOCAL is simply the total space taken by variables preceeding and including the declaration of Cee:

Offset(of Cee below LOCAL) = First-in Stack bytes + Cee bytes

The offset from LOCAL is 6 bytes, due to the Integer variables Alt and Bee which are pushed before Cee and 2 bytes for Cee itself.

The 3 bytes added to the value in the stack pointer are 3 bytes are preserved at the top of the stack for stack linkage.

The 6502 X register is used by the compiler as a stack pointer. It is very important to save and restore the X register if you need to use it.

The following table designates how many bytes of memory each type of variable or constant is provided on the stack:

Real	8	bytes
Integer	2	bytes
Char	1	bytes
Boolean	1	bytes
Pointer	2	bytes
ARRAY[1..n] OF Char	n	bytes
ARRAY[1..n] OF Boolean	n	bytes
ARRAY[1..n] OF Integer	2*n	bytes
Value Parameter(Real)	8	bytes
Value Parameter(Integer)	2	bytes
Value Parameter(Char,Boolean)	1	bytes
Value Parameter (ARRAY[n] OF Char,Boolean)	n	bytes
Value Parameter (ARRAY[n] OF Integer)	2*n	bytes
Variable Parameters(All)	2	bytes

Variable parameters are the parameters in the parentheses of the declaration of a procedure or function. They differ from value parameters because memory space is not allocated for the value of the variable but only for a pointer to a variable outside the procedure or function. Since each pointer takes two bytes, each variable parameter takes two bytes.

In PASCAL programs all the declarations come before the body of the program, function or procedure thus, the location of the variables is easily calculated. Always calculate the location of the variables relative to the beginning of the procedure, function, or program in which they appear.

It is inappropriate and misleading to calculate the stack location of variables based on their relative scope in the program, i.e. based on variables outside the scope of the ones in-line with the assembly code.

Predefined Labels

The following table gives the absolute locations of SP, LOCAL, and T. The first two pointers maintained by the compiler contain the addresses of the bottom and top of the Pascal variables stack and are two bytes long. The last label is the start of the temporary registers. There can be up to 16 temporary labels going from T to T+15.

```
SP      EQU  4
LOCAL   EQU  2
T       EQU 16
```

Passing Parameters through Chain

Parameters passed from one executable program to another executable program using Chain are passed by value and are only passed forward; i.e., to the next file to be run.

The parameters passed are the ones in that match type and position in the declaration section of the program. All parameters that follow any parameter that does not match cannot pass values through Chain.

```
PROGRAM Alpha;                    PROGRAM Beta;
VAR                               VAR
 A,B,C : Integer;                  D,E,F : Integer;
 X : Integer;                      Y : Real;
 P : Char;                         L : Char;
BEGIN                             BEGIN
 ...                               ...
```

If program Alpha calls program Beta through file the values of A, B and C are passed to D, E and F. Y does not match and a value is not passed. Although L matches P, no value is passed because it follows the mismatch of the parameters Y and X.

How to Chain Source Code Files

Sometimes a program is broken into sections that are to be loaded from the floppy disk when and where they are needed. This strategy is called chaining.

To chain files together:

```
PROGRAM MyExample;
BEGIN
 ...
 Chain('NextOne');
 ...
END.
```

The file to be included may be named either be a string constant, as above, or an array of characters.

The next statement executed will be the first statement in "NextOne."

STRING MANIPULATION

String

String is not a predefined PASCAL type; however, in order to use the Kyan PASCAL string manipulation functions and procedures it must be declared in the programs that use it.

As stated in previous chapters a string is simply an array of characters:

```
CONST
 Maxstring = 10;                    (* = 10 as an example *)
TYPE
 String =  ARRAY[ 1..Maxstring ] OF Char;
```

Maxstring must also be declared as a constant to whatever value is appropriate to the use of String in the program.

To use string procedures and functions in a program, along with the above declarations, the file containing the specific function or procedure you wish to use must be included using the #i format in the procedures and functions declaration section of the program. The three string manipulation functions and one procedure are Length, Index, Substring and Concat. For example, in order to use Substring, include the file Substring.I:

```
PROGRAM MyExample;
CONST
 Maxstring = 10;                    (* = 10 as an example *)
TYPE
 String =  ARRAY[ 1..Maxstring ] OF Char;
#i    Substring.I
BEGIN
 ...
```

The file containing the string manipulation function or procedure always is appended with .I as above. All the examples that follow use Maxstring = 10, although any value up to Maxint may be used.

Length

A string ends with the first blank space or the last character in the array. Length, a nonstandard function, returns the length of a String. For example, suppose:

```
PROGRAM MyExample;
CONST
 Maxstring = 10;
TYPE
 String =  ARRAY[ 1..Maxstring ] OF Char;
VAR
  s : String;
#i    Length.I

BEGIN
 s := 'abcd      ';
 IF (Length(s) = 4) THEN Writeln('This is true');
END.
```

Concat

Concat is an abbreviation of concatenate which means to put two strings together to produce a third. If S1 = 'ANY   ' and S2 = 'BODY   ', then S3 = 'ANYBODY   ' where the program calls:

```
Concat(S1,S2,S3);
```

Index

Index is a functions that returns the position of one string within another. If Index is used to find the position of S1 := 'a   ' in S2 := 'baby   ' then the following statement is true:

```
Index(S1,S2) = 2;
```

If the S2 is not found in S1, then the value of Index = 0.

Substring

Substring extracts part of a string, indicated by its two indices m and n. If a string of length 1 is to extracted from S1 := 'abcd   ' starting at the second position then the value for Substring would be 'b   '.

```
Substring(S1,2,1) = 'b         '     (* This has a true value *)
```

APPENDIX A
PEEK AND POKE EXAMPLE PROGRAMS

Program Graph

The program below draws lines on the CRT screen based on pixel-mapped coordinates which start with 0,0 at the lower left hand corner and go to 199,319 in the upper right hand corner.

For example, to draw the line that starts at 25,30 and ends at 100,120, enter the following while running the program:

25 30 100 120 <RETURN>

Wait for the screen to be cleared before entering coordinates.

Do not be concerned that the screen remains blank when you are entering these numbers. The pointers to the screen are now mapped by the lines you are entering and not the characters you type.

To exit the program and return to the mode where you can see what you enter, let one of the coordinates be negative. (If the program is exited by <RESTORE>-<RUN/STOP>, the pointers to the screen will not be restored.)

```
PROGRAM GRAPH;
TYPE CHARMEM=ARRAY [0..7999] OF CHAR;
     SCREENMEM=ARRAY [0..1023] OF CHAR;
     BLANK=SET OF 0..255;
     BLKARRAY=ARRAY [0..249] OF BLANK;
VAR I,X,Y:INTEGER; HIGHRES:^CHARMEM;
    SCREEN:^SCREENMEM;
    BANK:^CHAR;
    SCRADD:^CHAR;
    MODE:^CHAR;
    BLK:BLANK;
    BLANKHIG:^BLKARRAY;
    X1,X2,Y1,Y2,PIXEL:INTEGER;
    ENDFLAG,XINC,YINC,RAMP:INTEGER;
    SLOPE,SLOPEINV:REAL;
    MULTI:^CHAR;

FUNCTION ORF(BYTE:CHAR;NUM:INTEGER):
     CHAR;
TYPE CONVERT=RECORD
           CASE SELECT:INTEGER OF
           1:(PART:INTEGER);
           2:(MASK:BLANK)
           END;(*RECORD*)
VAR CONV:CONVERT;DUMMY:BLANK;

BEGIN
   CONV.PART := ORD(BYTE);
   CASE NUM OF
   0:DUMMY := [7];
   1:DUMMY := [6];
```

```
   2:DUMMY := [5];
   3:DUMMY := [4];
   4:DUMMY := [3];
   5:DUMMY := [2];
   6:DUMMY := [1];
   7:DUMMY := [0]
   END;
   CONV.MASK := CONV.MASK + DUMMY;
   ORF := CHR(CONV.PART);
END;

BEGIN
   ASSIGN(SCREEN,23*1024);
   ASSIGN(HIGHRES,24*1024);
   ASSIGN(BANK,-8960);
   ASSIGN(SCRADD,-12264);
   ASSIGN(MODE,-12271);
   ASSIGN(BLANKHIG,24*1024);
   ASSIGN(MULTI,-12266);

   (*ERASE SCREEN*)
   BLK := [ ];
   FOR I := 0 TO 249 DO
   BLANKHIG^[I] := BLK;

   (*HIGH RESOLUTION GRAPHICS ON*)
   BANK^ := CHR(198);
   SCRADD^ := CHR(120);
   MODE^ := CHR(59);
   MULTI^ := CHR(8);
   (*BLACK PIXELS, RED BACKGROUND*)
   FOR I := 0 TO 1023 DO
   SCREEN^[I] := CHR(2);

READ(X1,Y1,X2,Y2);
WHILE (X1>=0) AND (X1<=319) AND (Y1>=0)
 AND (Y1<=199) AND (X2>=0) AND (X2<=319)
 AND (Y2>=0) AND (Y2<=199) DO
BEGIN
 X := X1; Y := Y1;
 ENDFLAG := 0;
 XINC := -1; YINC := -1;
 IF X2>X1 THEN XINC := 1;
 IF Y2>Y1 THEN YINC := 1;
 IF Y2=Y1 THEN
  BEGIN
   SLOPE := 0;
   RAMP := 2;
  END
 ELSE
  BEGIN
   IF X2=X1 THEN
    BEGIN
     SLOPEINV := 0;
     RAMP := 1;
    END
   ELSE
    BEGIN
```

```
      SLOPE := (Y2-Y1)/(X2-X1);
      SLOPEINV := (X2-X1)/(Y2-Y1);
      IF ABS(SLOPE)>=1 THEN RAMP :=1
      ELSE RAMP :=2;
     END;
   END;
  REPEAT
  BEGIN

 (*DRAW A PIXEL*)
 PIXEL := ((199-Y) MOD 8)+8*(X DIV 8)
  + ((199-Y) DIV 8)*320;
 HIGHRES^[PIXEL]:=ORF(HIGHRES^[PIXEL], X MOD 8);

   CASE RAMP OF
    1:BEGIN
       IF Y=Y2 THEN ENDFLAG:=1;
       Y:=Y+YINC;
       X:=ROUND((Y-Y1)*SLOPEINV +X1);
      END;
    2:BEGIN
       IF X=X2 THEN ENDFLAG:=1;
       X:=X+XINC;
       Y:=ROUND(SLOPE*(X-X1) +Y1);
      END
   END;(*CASE RAMP*)
 END;
 UNTIL ENDFLAG = 1;
 READ(X1,Y1,X2,Y2);
 END;
 (*RESTORE FOR EDITOR*)
 BANK^ := CHR(199);
 SCRADD^ := CHR(22);
 MODE^ := CHR(27);
 END.(*PROGRAM GRAPH*)
```

The memory for high resolution graphics is described using an array of 8000 characters (bytes) located, in the example, 4k below the stack. In Kyan PASCAL the stack starts at H9000 (hexadecimal) with the compiler resident or HD000 in the stand- alone mode. Using the nonstandard Assign function, a pointer to the array is initialized at H6000 (24*1024). The screen memory must be in the same 16k bank (Commodore hardware limitations) and is an array of 1024 characters. The pointer to the screen array is initialized at 23*1024, 1k below the high resolution character memory. The various pokes needed to point the graphics chip at the arrays are done by pointers to the required Commodore registers.

In order to linearize the way the Commodore maps the array into pixel elements, a number of calculations must be performed including an "or" function. "ORF" is defined using a variant record to perform the operation.

Once the pixels are turned on, some code is added to read in the starting point X1,Y1 and the end point X2,Y2. There is no way to prompt or display the coordinate numbers, since the standard I/O points to another memory area.

Although the speed of drawing the line is not optimized, the initialization of the background is. Instead of initializing each character, the program uses a zero set (32 bytes) and loops only 250 times. The set equate uses in-line machine code and is extremely fast.

Program Sprites

The following program places a small red square in different positions on the CRT screen. It is similar to the previous program in that the screen is remapped. When you type:

    100 100 <RETURN>

the numbers do not appear, but the small red square appears in the position 100,100.

In order to restore the screen to the editor, enter one or more negative coordinates such as:

    -200 300 <RETURN>

The program is called Sprites because it uses sprite graphics and in particular sprite 0, which is the small red square. Note that coordinates less than 50, such as 48,100 are off the screen: the upper right hand corner of the screen is 50,50.

```
PROGRAM SPRITES;
TYPE CHARMEM=ARRAY[0..7999] OF CHAR;
     SCREENMEM=ARRAY[0..1023] OF CHAR;
     SPRITEMEM=ARRAY[0..7,0..63] OF CHAR;
     ROMMEM=ARRAY[0..2047] OF CHAR;
     COLORMEM=ARRAY[0..999] OF CHAR;
     SPRITECOLOR = ARRAY[0..7] OF CHAR;

VAR I,J,K,X,Y:INTEGER;
    HIGHRES:^CHARMEM;
    SCREEN:^SCREENMEM;
    BANK:^CHAR;
    SCRADD:^CHAR;
    MODE:^CHAR;
    INTER:^CHAR;
    SELECT:^CHAR;
    ROM:^ROMMEM;
    COLOR:^COLORMEM;
    SPRITE:^SPRITEMEM;
    SPRTEN:^CHAR;
    SPRITEX:^CHAR;
    SPRITEY:^CHAR;
    SPRCOLOR:^SPRITECOLOR;

BEGIN
   ASSIGN(SCREEN,23*1024);
   ASSIGN(HIGHRES,24*1024);
   ASSIGN(BANK,-8960);
   ASSIGN(SCRADD,-12264);
   ASSIGN(MODE,-12271);
   ASSIGN(INTER,-9202);
```

```
   ASSIGN(SELECT,1);
   ASSIGN(ROM,-12288);
   ASSIGN(COLOR,-10240);
   ASSIGN(SPRITE,22*1024);
   ASSIGN(SPRTEN,-12267);
   ASSIGN(SPRITEX,-12288);
   ASSIGN(SPRITEY,-12287);
   ASSIGN(SPRCOLOR,-12249);

(*SPRITE GRAPHICS ON*)
BANK^ := CHR(198);
SCRADD^ := CHR(120);
MODE ^ := CHR(27);

(*POINT TO SPRITES*)
SCREEN^[1016] := CHR(96);

(*COPY CHAR ROM TO RAM*)
J := ORD(INTER^);
INTER^ := CHR(0);
K := ORD(SELECT^);
SELECT^ := CHR(2);
FOR I := 0 TO 2047 DO
 HIGHRES^[I] := ROM^[I];
SELECT^ := CHR(K);
INTER^ := CHR(J);

(*COLOR CHARACTERS*)
FOR I := 0 TO 999 DO
 COLOR^[I] := CHR(14);
FOR I := 0 TO 1000 DO
 SCREEN^[I]:= CHR(I MOD 20);
FOR I := 0 TO 63 DO
 SPRITE^[0,I] := CHR(255);

(*ENABLE SPRITE 0*)
SPRTEN^ := CHR(1);

(*COLOR SPRITE*)
SPRCOLOR^[0] := CHR(2);

(*MOVE SPRITE*)
READ(X,Y);
WHILE (X>=0) AND (Y>=0) DO
 BEGIN
  SPRITEX^ := CHR(X);
  SPRITEY^ := CHR(Y);
  READ(X,Y);
 END;

  (*RESTORE FOR EDITOR*)
  SPRTEN^ := CHR(0);
  BANK^ := CHR(199);
  SCRADD^ := CHR(22);
  MODE^ := CHR(27);
END.
```

The Sprites program uses the same screen and character memory as the program before it, Graph. The character memory could be reduced significantly if the number of character types actually used is smaller. In Sprites, the first 2k of the character ROM is copied to RAM, and the first 20 characters of the alphabet are placed repetitively on the screen by writing them out to the first 1000 locations of screen memory. The last 8 memory locations of the screen point to the sprites. Sprite 0 is in location 1016 of the screen.

In general, a sprite is defined as an array of 64 characters, and there are 8 sprites available at one time. The Commodore graphics chip is programmed using pointers as in Graph.

This program provides most of the framework for developing very complex sprite graphics and saves much time in getting started. The data structures are generalized and not used fully by the example. For instance, all potentially active sprites can be stored in the [0..7,0..63] array, while only Sprite 0 is is enabled in the example.

APPENDIX B
COMPILER AND RUN-TIME ERROR MESSAGES

Compiler Error Messages

1. syntax error
2. unexpected end of input
3. array dimension expected
4. to or downto expected
5. type specification expected
6. ordinal type expected
7. := expected
8. : expected
9. , expected
10. ; or end expected
11. compiler directive expected
12. do expected
13. end expected
14. = expected
15. identifier expected
16. [expected
17. constant expected
18. (expected
19. of expected
20. type identifier expected
21. . expected
22. program expected
23.] expected
24.) expected
25. ; expected
26. .. expected
27. then expected
28. unsigned integer expected
29. file name expected
30. can't open file
31. illegal file name
32. ; or until expected
33. missing end statement(s)
34. extraneous end statement(s)
35. ; or case expected

Assembler Error Messages - ADVANCED KYAN PASCAL ONLY

1.	A	addressing mode error
2.	L	label required with EQU
3.	M	multiply defined symbol
4.	U	undefined expression
5.	O	unrecognizable opcode
6.	S	syntax error
7.	J	branch address is out of range

Run-Time Error Messages

1. bad subscript
2. too many active files (Maximum is 5 files)
3. file not active
4. set element out of range
5. heap overflow
6. bad ln(argument)
7. bad exp(argument)
8. read past eof (End of File)
9. out of memory
10. arithmetic overflow

APPENDIX C
QUICK GUIDE TO KYAN PASCAL

Predefined Types:
   Integer, Boolean, Real, Char, Pointer,
   (scalar values..)

Predefined File Types:
   Input, Output, Text,
  Standard Kyan Pascal Only - Printer

Predefined Procedure to Redirect Output File:
  (Advanced Kyan Only)
    PRON, PROFF

Compound Types (Reserved Words)
   ARRAY[..] OF.., RECORD OF..
   SET OF.., FILE OF..

Predefined Functions with Real or Integer Parameters:*
   Abs(Real or Integer), Arctan(Real or Integer),
   Cos(Real or Integer), Exp(Real or Integer),
   Ln(Real or Integer), Round(Real),
   Sin(Real or Integer), Sqr(Real or Integer),
   Sqrt(Real or Integer), Trunc(Real)

Predefined Functions with Other Parameters*
   Ord(scalar), Pred(scalar), Succ(scalar),
   Chr(Integer), Odd(Integer), EOF(file),
   EOLN(Text file)

Predefined File Procedures:
   Reset(file), Get(file), Rewrite(file),
   Put(file), Page(file), Read(..), Readln(..),
   Write(..), Writeln(..)
 Advanced Kyan Pascal Only - Chain(file), Seek(file, record number)

Predefined Pointer Procedures:
   New(pointer)
 Advanced Kyan Pascal Only - Dispose(pointer)

Predefined Non-standard Pointer Procedures:
   Assign(pointer, integer)

Predefined Constants:
   True, False, Maxint

Value Reserved for Unassigned Pointer:
   NIL

Conditional Instructions (Reserved Words):
   IF-THEN-ELSE, WHILE-DO, REPEAT-UNTIL,
   FOR-TO-DO, FOR-DOWNTO-DO

* Allowed parameter types appear in parentheses

Operators (Reserved Words):
   Arithmetic Operators: DIV, MOD
   Boolean Operators: AND, NOT, OR, IN

Operators (Reserved Characters):
   Arithmetic Operators: + - * /
   Relational (Comparison) Operators: = < >

Miscellaneous Reserved Characters:
   Punctuation: . , ; : ' ( ) [ ]
   Pointer: ^ (equivalent to ↑)

Grammatical Identifiers (Reserved Words):
   CONST, FUNCTION, LABEL, PROCEDURE,
   PROGRAM, RECORD, TYPE, VAR,
   BEGIN..END, CASE..OF.., GOTO, WITH..DO..

Pre-Compilation Instructions (Non-standard)
   #i.....#   (include file)
   #a.....#   (include assembly code)

String Functions and Procedures (Non-standard)
  (String and Maxstring must be declared)
  Length(string), Concat(string,string,string),
  Index(string, string), Substring(string, integer, integer)

APPENDIX D
SPECIFICATIONS

Integer: Range of -32768 to +32767
Maxint = 32767

Real: Range of -1.00E+99 to +1.00E+99
Precision of 13 decimal digits

Char: Character
Printable and nonprintable ASCII characters
corresponding to ordinal values 0 to 256

Pointer: Represented by 16-bit Integer

SET: Maximum number of members is 256

Requirements: Disk Drive

Maximum Program Size (Standard Kyan Pascal)
Kyan PASCAL in Memory: 23K-bytes
Stand-Alone Mode: 23K-bytes + 16K-bytes (data only)

Maximum Program Size : 40K-bytes

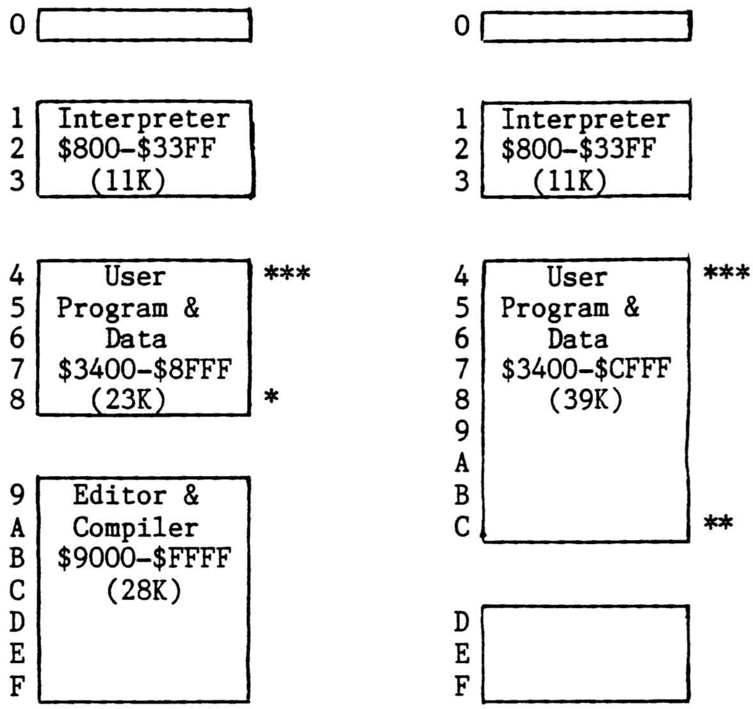
Significant Identifier Length: 8 characters

Significant File Name Length: 8 characters

Advanced Kyan Pascal Only:
Variable and Parameter Stack: Pointer to top at 0002
Pointer to bottom at 0004
Maximum Relative File Element Size : 128 bytes

Memory Maps - Standard Kyan Pascal Only

Kyan PASCAL Environment   Stand-Alone Environment

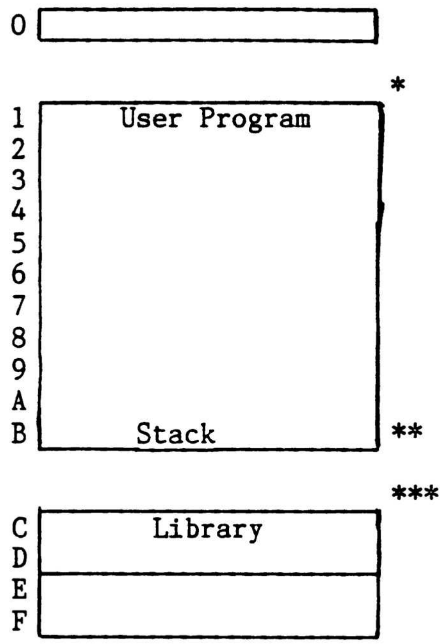


* Stack starts at $8FFF and grows toward low memory.

** Stack starts at $CFFF and grows toward low memory.

*** User program is loaded starting at $3400. In both environments the maximum program size is 23K. Heap starts at end of user program and grows toward high memory. In the stand-alone environment an additional 16K is available for data.

Memory Map - Advanced Kyan Pascal Only



* User program is loaded starting at $0800 and grows toward high memory.

** Stack starts at $A7FF and grows toward low memory.

**** Library is located from $A800 to $CFFF.

SOFTWARE MEDIA LIMITED WARRANTY

Kyan Software warrants to the original consumer purchaser of **Kyan PASCAL** for a period of ninety (90) days from the date of purchase that the recording medium, and only the recording medium, on which the software program is recorded will be free from defects in materials and workmanship. Defective media returned by purchaser to Kyan Software during that ninety day period will be replaced free of charge provided that the returned media has not been subjected to abuse, unreasonable use, mistreatment, neglect or excessive wear.

Following the initial ninety day period, defective media will be replaced for a $9.50 replacement fee. To qualify for replacement, defective media must be returned postage paid in protective packaging to:

Kyan Software
1850 Union St. #183, San Francisco, CA 94123

Defective media must be accompanied by (1) proof of purchase, (2) a brief statement describing the defect, (3) a $9.50 check payable to Kyan Software (if beyond the ninety day warranty period), and (4) your return address.

THIS WARRANTY IS LIMITED TO THE RECORDING MEDIA ONLY AND DOES NOT APPLY TO THE SOFTWARE PROGRAM ITSELF WHICH IS PROVIDED "AS IS."

THIS WARRANTY IS IN LIEU OF ALL OTHER WARRANTIES, WHETHER ORAL OR WRITTEN, EXPRESS OR IMPLIED. ANY APPLICABLE IMPLIED WARRANTIES INCLUDING WARRANTIES OF MERCHANTABILITY AND FITNESS ARE HEREBY LIMITED TO NINETY DAYS FROM THE DATE OF PURCHASE. CONSEQUENTIAL OR INCIDENTAL DAMAGES RESULTING FROM A BREACH OF ANY APPLICABLE EXPRESS OR IMPLIED WARRANTIES ARE HEREBY EXCLUDED.

Some states do not allow limitations on how long an implied warranty lasts or do not allow the exclusion or limitation of incidental or consequential damages, so the above limitations or exclusions may not apply to you.

This warranty gives you specific legal rights and you may also have other rights which vary from state to state.