KYAN PASCAL

for the Atari

USERS MANUAL

KYAN SOFTWARE INC.
SAN FRANCISCO, CALIFORNIA

KYAN PASCAL

Conforms to the ISO Standard for Pascal Compilers.

This software is designed for any 8-bit Atari computer with at least 48K of memory

This software is compatible with the Kyan Programming Toolkits

Kyan Software Inc.
San Francisco, California

TABLE OF CONTENTS

CHAPTER PAGE

Notice

Kyan Software reserves the right to make improvements to the products described in this manual at any time and without notice. Kyan Software cannot guarantee that you will receive notice of such revisions, even if you are a registered owner. You should periodically check with Kyan Software or your authorized Kyan Software dealer.

Kyan Software programs are sold only on the condition that the purchaser agrees to the terms contained in the Software License Agreement that begins on the following page. Please read this agreement before using the software.

Acknowledgement

Kyan Software would like to acknowledge the major contribution to this manual made by:

Technical Writers Inc.
P.O. Box 6687
New York, NY 10128
(212) 861-0216

Software License Agreement

IMPORTANT: Kyan Software products are sold only on the condition that the purchaser agrees to the terms of the following license. PLEASE READ THIS AGREEMENT CAREFULLY. If you do not agree to the terms, return the unused package to Kyan Software or to your dealer immediately for a refund. If you agree to the terms contained in this License Agreement, complete the enclosed registration card and return it to Kyan Software.

When you purchase a Kyan Software product, you acknowledge that:

1. Kyan Software has a valuable proprietary interest in the computer programs and printed documentation (hereafter called "SOFTWARE"). What you have purchased is a non-transferable and non-exclusive license to use the SOFTWARE. Kyan Software retains ownership of the SOFTWARE.

2. You may not copy or reproduce the SOFTWARE for any purpose, other than to make backup or archive copies as provided for in Section 117 of the U.S. Federal Copyright Law, without the express permission of Kyan Software.

3. You may not copy, distribute, or otherwise make the SOFTWARE available to any third party without the express permission of Kyan Software.

4. If you merge or use the Kyan Pascal Runtime Library (LIB) in conjunction with another program, it continues to be the property of Kyan Software. However .....

> Kyan Software hereby grants you a non-exclusive license to merge or use the Runtime Library (LIB) in conjunction with your own programs provided that:
>
> a) You acknowledge Kyan Software's copyright and ownership of the Library in a prominent location in the written documentation and on the magnetic media.

Software License Agreement (cont.)

b) You include a Kyan Software DISCLAIMER OF WARRANTY in the written documentation.

c) You notify Kyan Software in writing that you are exercising your rights under this agreement.

This license to merge or use portions of the SOFTWARE in your own programs is limited to the LIB file only. All other files are specifically excluded from this license. Please contact Kyan Software for information regarding the license and use of other Kyan software modules.

5. This license is effective until terminated. You may terminate it at any time by destroying the SOFTWARE along with all copies, modifications and merged portions in any form. It will also terminate if you fail to comply with any term or condition of this Agreement. You agree upon such termination to destroy the SOFTWARE together with all copies, modifications and merged portions in any form.

Copyright

Limited Warranty

Kyan Software warrants the diskette(s) on which the Kyan software is furnished to be free from defects in materials and workmanship under normal use for a period of ninety (90) days from the date of delivery to you as evidenced by your proof of purchase.

Kyan Software also warrants that this software performs substantially in accordance with the published specification statement, the documentation, and authorized advertising. Kyan Software, if notified of significant errors within 90 days from the date of purchase, will at its option:

a) correct demonstrable and significant program or documentation errors within a reasonable period of time; or

b) provide the customer with functionally equivalent software; or

c) provide or authorize a refund.

Except for the limited warranty described in the preceding paragraphs, Kyan Software makes no other warranties, either express or implied, with respect to the software, its merchantability or its fitness for any particular purposes.

Some states do not allow the exclusion or limitation of implied warranties for liabilities for incidental or consequential damages, so the above limitations or exclusions may not apply to you.

This Agreement constitutes the entire agreement between the parties concerning the subject matter hereof.

WELCOME TO THE KYAN SOFTWARE FAMILY

Kyan Pascal is the core of a powerful software development system. It has been tested for conformance to the ISO Pascal standard (level 0). Kyan Pascal is designed to be used with Kyan's Programming Toolkits. These toolkits, when used in conjunction with Kyan Pascal, make software development faster and easier. Kyan Pascal consists of this user manual and one flippy disk (where program files are recorded on both sides of the disk).

We strongly recommend that you make and use backup copies of the Kyan Pascal disk. Keep your original Kyan disk in a safe location in case something happens to your copies. (Remember ... Murphy is alive and well, and he loves to mess with computers!)

Kyan Software has enclosed an owner registration card. Please fill in and return this card as soon as possible. Registered owners of Kyan Software products are eligible for technical support and periodic low-cost software upgrades. Registered owners can also subscribe to **"Update ..... Kyan"**, a bimonthly newsletter which contains programming tips, utility programs, and the latest information regarding upgrades and new product releases.

Copy Protection

Kyan Software products are not copy-protected. As a result, you are able to make backup copies and load your software onto a hard disk or into RAMdisk. We trust you. Please do not violate our trust by making or distributing illegal copies.

Technical Support

Kyan Software has a technical support staff ready to assist you with any problems you might encounter. If you have a problem, we request that you first consult this users manual. We have worked very hard to identify and include in this manual, the answers to questions and problems most frequently encountered.

If you have a problem which is not covered in the manual, our support staff is ready to help. If the problem is a program which won't compile or run (and you are sure that it should), we can best help if you send us a description of the problem and a listing of your program (better yet, send us a disk with the listing on it). We will do our best to get back to you with an answer as quickly as possible.

If you question can be answered on the phone, then give us a call. Our technical staff is available to assist on Monday through Friday between the hours of 9 AM and 5 PM, West Coast Time. You may reach them by calling:

Technical Support: (415) 626-2080

Suggestion Box

Kyan Software likes to hear from you. Please write if you have sugges-tions, comments and, yes, even criticisms of our products. We do listen. It is your suggestions and comments that frequently lead to new products and/or product modifications.

We encourage you to write. To make it easier, we have included a form in the back of this manual. This form makes it easier for you to write and easier for us to understand and respond to your comments. Please let us hear from you.

Mailing Address: **Kyan Software Inc.**
1850 Union Street #183
San Francisco, CA 94123

INTRODUCTION

ISO PASCAL

Pascal is a language used to program computers. It was developed in the late 1960's by Niklaus Wirth, a professor of computer science at a major European university. Professor Wirth had become frustrated by the lack of a structured computer language which could be taught to students and used in large software development projects.

Professor Wirth teamed up with Kathleen Jensen, and the Pascal language was formally introduced in 1971. Their principal objectives in introducing the language were "...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." (Reference: "Pascal User Manual and Report", Jensen and Wirth, Springer-Verlag, Berlin, 1974).

As a programming language, Pascal has a number of significant advantages.

1. Pascal compilers are available for almost any computer, ranging from the smallest micro-computers to the largest super-computers. This means that programs written in Pascal can easily be transported from one computer to another.

2. Pascal is a structured programming language. Programs written in Pascal are well-organized, understandable and easy-to-follow. This structure facilitates the development and subsequent maintenance of large, complex computer programs.

3. Pascal is designed to encourage good programming habits. It is a self-documenting and self-structuring language. Its structure promotes top-down programming and modularization. Program modules allow the programmer to create user-defined functions and procedures. This means that students who learn Pascal also

learn how to program more skillfully and more effectively. These skills are carried forward and can be applied to other programming languages.

4. Pascal takes advantage of the latest developments in computer science and hardware. Programmers are able to take full advantage of the hardware they are using and are not constrained by limitations imposed by the programming language.

These advantages have resulted in Pascal becoming a widely used language for both elementary and advanced programming. Pascal has also been selected by the College Entrance Examination Board as the required language for high school students who seek advanced placement in college computer courses.

KYAN PASCAL

Kyan Pascal is a full implementation of the Pascal programming language based on the ISO (International Standards Organization) standard for Pascal. In addition to supporting all of the standard functions and procedures, Kyan Pascal includes many extensions to the language. It is a very powerful software development tool.

The following notes describe the general structure and some of the features of Kyan Pascal. More detailed technical information is presented in later sections of the manual.

Compiler/Assembler

Kyan Pascal is actually two products in one -- a Pascal compiler and a 6502 assembler. The Pascal compiler takes Pascal source code and produces assembly language source code; the assembler takes the compiler output and produces an executable machine code file.

The assembler makes two passes through the assembly source code before producing an executable file. This two-pass approach allows all forward references to be resolved and results in generation of the fastest and most efficient machine code possible for the 6502 microprocessor. It also results in an executable file which requires 10 - 20 percent less memory since jumps and jump tables are not needed.

Speed

Kyan Pascal produces code that runs about twice as fast on a 6502 microprocessor as the best selling Pascal does on a Z80 (assuming equal CPU clock rates). The benchmark used for this comparison is the Sieve of Eratosthenes Algorithm and the time required to generate the first 1,899 prime numbers (execution time: 12 seconds).

Arithmetic Precision

The arithmetic unit used in Kyan Pascal is either a 16 bit integer or 13 decimal digit Binary Coded Decimal. Kyan Pascal uses BCD real numbers to eliminate round-off errors of binary representations (i.e., the result of a simple division is displayed as 3.0 instead of 2.9999999).

Despite the high level of arithmetic precision, floating point benchmarks show that Kyan Pascal produces code that runs at approximately the same speed as compilers with 7 to 9 digit precision. Since calculation speed is proportional to the square of the precision of the real number, Kyan Pascal is actually running 2 to 4 times faster than programs offering the equivalent precision.

Linking

In Kyan Pascal, program modules can be linked together by "including" Pascal or assembly language source files into the main program. When the main program is compiled, the "included" source code files are called and compiled along with the main program. This "include" linking technique is superior to object module linking because it is faster; does not require two passes of the object modules to produce an executable file; and, does not introduce non-standard effects on modules of the Pascal program (i.e., no scope rules, no parameter checking, and no mechanism for assigning a lexical level to variables).

Pack And Unpack

PACK and UNPACK are standard procedures in Pascal. Kyan Pascal automatically packs all structures at the byte level. The only variable type which is not fully packed is boolean. Because of the poor bit handling of the 6502 microprocessor, Kyan Pascal does not support the packing of booleans. Packed and unpacked structures are identical.

Kyan Pascal's Runtime Library

The Runtime Library is a software module which contains the general purpose routines used in most Pascal programs. Routines in the Library include the input/output functions, the floating point package, the transcendental functions, and set routines.

The purpose of the Runtime Library is to conserve space on the disk. Rather than appending a copy of all Pascal routines to each program, the Runtime Library allows many programs to share one copy of the routines. Since the Runtime Library is approximately 10K in size, you can see how much disk space you gain if you want to put 3 or 4 programs on the same disk.

NOTE 1: If you want to run your compiled programs independently of the Kyan Pascal source disk, the Library must be copied onto your program disk.

NOTE 2: If you are an advanced Pascal programmer and want to append portions of the Runtime Library directly to your Pascal program, please contact Kyan Software. We offer a Code Optimizer Toolkit which includes the source code for the Runtime Library.

The KIX File Management System

Kyan Pascal has both a menu-driven user interface and a command-line file management system called KIX™. Beginners may want to rely on the menus until they gain proficiency with Pascal programming. Advanced programmers will want to by-pass the menus and take advantage of the command environment of KIX. KIX provides the programmer with powerful and extensive control over file management. Please see Chapter VI for more information about KIX.

Error Reporting

Kyan Pascal provides features that enable the programmer to find the syntax errors that account for over 90% of all compiler failures. Over 100 error messages report not only the type of error found, but also the line containing the error. In addition, Kyan Pascal does not stop looking for errors after the first one is found. Although compilation stops, error detection continues and all errors encountered are listed at the same time.

KYAN PASCAL: A Product in Evolution

A programming language is different from other types of software. Unlike a word processing or spreadsheet package, it is extremely difficult to define all the possible uses of the software. As an analogy, consider a spoken language such as English or French: How many different ways are there to use the language syntax? You can write a poem, a letter, or the great (or not so great) American novel. Will the language support all of these applications? Will the typewriter you use have all the symbols and characters you need? Will the readers of your creation know what all the symbols, characters, and words mean?

We are confronted with similar questions when implementing a programming language. Jensen and Wirth solved most of the Pascal syntax problems. But, Kyan must deal with the problem of accurately interpreting this syntax and correctly compiling a listing which is meaningful to the computer. We constantly face the question, "What program construction (legal or illegal) can cause a crash during compilation or runtime?"

During the development and beta testing of each new product, we subject it to a battery of test programs to make sure it works properly under as many conditions as possible. When we release the product, we have a high level of confidence that it will perform in a satisfactory manner.

However, once the product is in the field, users inevitably write programs which uncover bugs. When this occurs, the customer calls our tech support group and points out the problem (sometimes in very graphic terms). In 99 cases out of 100, we are able to correct the problem and send the customer a new disk. This fix is then added to the list of changes which will be released in the next general REVISION of the compiler (i.e., version 1.1 to 1.2).

Over a period of time, the bugs we find become far more subtle--99% of the users would never encounter them. But, since we want to ship the best possible product, we continue to document and fix every bug identified.

Then, just when the product is "perfect," the engineers or marketing staff come up with some new enhancements to the product (i.e., "Let's increase the size of the symbol table and add some new extensions!"). We then go through another development cycle and release a product UPGRADE (i.e., Version 1.3 to 2.0). And, the whole process begins again.

So, a programming language product like Kyan Pascal is never done--it is constantly evolving to a better and more refined state. We can never say with absolute certainty that it is "bug-free." However, we can say that when you buy a product from Kyan Software, you receive the highest quality possible, good technical support, and periodic revisions and product upgrades at the lowest possible cost.

OTHER REFERENCES

This manual is only an introduction to the basics of Pascal. If you would like to learn more about the history of Pascal or programming in Pascal or Assembly language, the following books may be of interest.

Oh! Pascal!, Second Edition
D. Cooper and Michael Clancy, W.W. Norton & Co. 1985

Standard Pascal User Reference Manual,
D. Cooper, W.W. Norton & Co., 1983

Programming in Pascal,
P. Grogono, Addison-Wesley Publishing, 1978.

Pascal, A Problem Solving Approach,
E.B. Kaufman, Addison-Wesley Publishing, 1982.

Introduction to Pascal, R. Zaks,
Sybex, Inc., 1981

Pascal User Manual and Report, (ISO Pascal Standard)
K. Jensen and N. Wirth, Third Edition, Springer-Verlag, 1974, 1985.

Programming a Micro-computer: 6502,
C.C. Foster, Addison-Wesley Publishing, 1978.

6502 Assembly Language Programming,
L.A. Leventhal, Osborne/McGraw-Hill, Inc., 1979

HOW TO USE THIS MANUAL

The KYAN PASCAL MANUAL is directed primarily to the novice and intermediate-level programmer. Someone who knows little or nothing about Pascal should read the chapters of the manual in strict sequence. If you know Pascal, you may want to skip the tutorial, but you should read the chapter on Kyan Pascal Programming to learn about the features that are unique to Kyan's implementation.

The manual is divided into 6 chapters and 6 appendices.

Chapter 1, "**Getting Started,**" is intended for all users. It explains how to boot Kyan Pascal, make backup copies, and use the Kyan disk most effectively with your hardware configuration.

Chapter 2, "**The Text Editor,**" explains the Kyan text editor. It shows how to select the editor from the Main Menu, create a source code file, and how to save that file. It provides a detailed explanation of all the features available with the Kyan Text Editor.

Chapter 3, "**Kyan Pascal Programming,**" explains the features of Kyan's Pascal compiler. It shows how to call the compiler and choose the options available to the programmer. The special features include special graphics capabilities, including files, and chaining procedures.

Chapter 4, "**The Pascal Tutorial,**" is a 2-part introduction to the Pascal language. It covers 15 lessons that introduce all of the major elements of the language. Novice users should read the tutorial carefully.

Chapter 5, "**Assembly Language Programming,**" illustrates how to include assembly code within a Pascal program. Since this section does not explain assembly programming techniques, only advanced programmers will need to read it. The ability to include assembly code, however, greatly increases the power of your programs.

Chapter 6, "**The KIX File Management System**" explains all the commands available to advanced programmers who desire to use the KIX system.

The Appendices contain technical information and other useful reference material which will facilitate programming with Kyan Pascal. It is worthwhile to spend a few minutes perusing this material.

I GETTING STARTED

Kyan Pascal is designed to run on any 8-bit Atari computer with at least 48K of random access memory. The software uses Atari's DOS 2.5 operating system which is included on the disk. This chapter describes:

- o The DOS 2.5 operating system
- o Using the DOS 2.5 menu
- o Creating a backup copy of Kyan Pascal
- o Configuring Kyan Pascal for your computer.

Kyan Pascal consists of a flippy disk which contains files on both sides. **Before you use Kyan Pascal, you should make a backup copy of both sides.** This chapter describes how to do this.

If you are already familiar with DOS 2.5, you can go directly to the section entitled "Using and Configuring Kyan Pascal."

Atari DOS 2.5

Kyan Pascal includes and uses version 2.5 of Atari's Disk Operating System. It is included on Side 1 of your Kyan Pascal disk.

If you are not familiar with DOS 2.5, we suggest that you to stop reading the manual at this time and boot this disk (remember, Side 1). Explore the DOS menu and the functions available. When you are finished, please return to this manual.

If you want to learn more about the capabilities and features of DOS 2.5, you should contact Atari Corp. for more information.

Note: If you are an experienced, user, you may want to use a more powerful DOS from a third party. Most DOS's are compatible, and we have gone out of our way to make Kyan Pascal that way. Some DOS's, however, may not be compatible. The only sure way to find out if a DOS is compatible with Kyan Pascal is to try it.

A Word About Disk Density

There are three different densities used by Atari DOS's -- single, enhanced, and double. DOS 2.5 supports single and enhanced densities only, and the Kyan Pascal disk is shipped in single density. Double density provides twice as much disk space but is not supported by DOS 2.5. If you want to use a double density disk format, you must provide a DOS which supports it.

DOS Filenames

When you list a directory of your Kyan Pascal disk, you will see that the files all have different names. With just about every DOS for your Atari, the filename must follow these rules.

1. The filename must be preceded by a device name such as **D1:** or **D:** for disk drive 1, or **D8:** for disk drive 8 (usually a RAMdisk).

2. In DOS 2.5 the filename must be in CAPITAL letters.

3. The first character of the filename must be a letter. Following characters can be letters or numbers. Other characters are not permitted.

4. The filename can have a maximum of 8 characters. (NOTE: You can add an optional three character extension to the filename. It must be separated from the main filename by a period and must follow the same naming conventions.)

The following is a list of legal filenames:

D:FILENAME.EXT
D2:PROG1.V2
D1:PR.I
D4:PROGRAM.P
D2:PROG1
D8:PC
D3:A1B2C3D4.E56

The following is a list of illegal filenames.

FILENAME.EXT	[no device is specified]
D9:PROGRAM.P	[max number of devices is 8]
D1:files.i	[lower case characters not allowed]
D2:1STPROG.P	[first character must not be a number]
D8:MY FIRST.FILE	[spaces not allowed and extension may not exceed 3 characters]

When you use the KIX File Management System, you will find that it allows you to use lower case letters when typing. However, you are only able to do this because KIX automatically converts the filenames to uppercase when it writes them to the disk.

Atari has a unique input/output system in which peripherals, or devices, each have a device name. You have already seen that **D** stands for disk drive, but there are more. **P** is for printer; **S** is for screen; **K** is for keyboard; and more. A colon (:) must follow the letter of the device name; the filename will follow only if a disk drive is being used. (Note: only the disk drive has a number after its letter.) You can experiment with different device names; your files will be read from or written to the different devices that you specify.

Main Menu

When you boot Kyan Pascal, the DOS will be loaded into memory and the Kyan boot screen will appear. You should press <RETURN> and type "MENU", followed by another <RETURN>. The Kyan Pascal Main Menu program will now be load and run.

```
User Device:  D1:

KYAN PASCAL MAIN MENU

Option      Description
ED          Editor
PC          Pascal Compiler
AS          Assembler
DOS         DOS Menu
CAT         Concatentate Files
CD          Change Device
CHMOD       Change Protection Status
CP          Copy Files
FORMAT      Format a Disk
LS          List Directory
MV          Rename File
PWD         Print Working Directory
RM          Delete File
SD          Screen Dump

Type MENU to repeat this Main Menu
```

To select an option, you must press **<Return>** followed by the name of an option and **<Return>** again. If you want to begin writing or editing a program, select **ED**. If you want to compile a source code file, select **PC**. Or, if you want to perform a file management task, select **DOS**. If you want to use the KIX commands, select one of them.

NOTE: For the moment, ignore the **KIX** commands. They are for advanced programmers. If you are just beginning, don't confuse yourself by trying to learn too many things at once. Kyan Pascal can be run exclusively from the DOS menu and, until you are comfortable with Pascal programming, don't bother with the KIX. When you are ready, Section VI which explains KIX.

For now, let's select the **DOS**. The DOS menu appears on the screen. Remember, you can call the DOS menu by typing "DOS" any time you see the KIX prompt (%).

REMEMBER: You can return to the Kyan Pascal Main Menu at any time you see the system prompt by typing "MENU".

Quick Guide to Selected DOS 2.5 Utilities

The following instructions will help you get started with Kyan Pascal. They explain how to:

- Copy a disk
- Format a blank disk
- Copy a file
- Delete a file
- View a disk directory

Read these sections carefully and follow the instructions that are relevant to your disk drive and system configuration. You will be making a backup copy of the Kyan Pascal disk.

Duplicating a Disk

You should immediately make a backup copy of your write-protected Kyan Pascal source disk. Use this backup copy whenever you are programming. That way, if the contents of a disk are accidentally destroyed, you will still have your source disk untouched. (Please refer to the enclosed Copyright Notice and License Agreement for a description of the limitation on the use of backup copies.)

The following instructions explain how to make a backup copy of a disk. When using the Duplicate function, you do not need to use a pre-formatted disk. The Duplicate function formats for you automatically.

Duplicating A Disk

1. If you are not already in the DOS menu, type **DOS** and press **<Return>**. This will load the DOS menu.

2. Press **J** for "Duplicate a Disk".

3. The computer will ask you which drives are the source and destination. If you have one disk drive, type **1,1** which means drive 1 will be used as both the source and destination drive. If you have two drives, type **1,2** which specifies drive 1 as the source and drive 2 as the destination. If you system is set up differently, remember that the first number is the source drive (i.e., it contains the disk that is to be copied) and the second number is the destination drive (i.e., it contains the blank disk onto which the source disk will be copied).

4. The computer will ask you to insert the disk(s) and press return to begin copying. First, the source will be read into memory. Then, if you are copying with one drive, you will be prompted to insert the destination disk. The destination disk will be formatted and written to. Two passes are required to completely read and write the source disk, so, if you are copying with one drive, you will again be prompted to exchange disks.

5. When the copy is complete, the following message will appear:

TYPE LETTER OR RETURN FOR MENU

Formatting a Disk

You should always keep several pre-formatted disks handy. Use them to store the Pascal programs you write.

Formatting a Disk

1. If you are not already in the DOS 2.5 menu, type **DOS** and press **<Return>**.

2. Type **P** to "Format a Single Density Disk".

3. The computer will ask which drive has the disk to be formatted, and you should type the appropriate number.

4. The computer will ask if you are sure you want to destroy the contents of the disk. If you are sure, type **Y**. The disk will then be formatted.

Copying a File

To copy a file from one disk to another, use the "Copy File" commands found in the DOS menu.

Copying A File (one disk drive system)

1. If you are not already in the DOS 2.5 menu, type **DOS** and press **<Return>**.

2. Press **O** to "Duplicate a File".

3. Enter the filename when the computer prompts for the name of the file that you want to copy.

4. Insert the source disk and follow the prompts. The file will be copied onto the destination disk in the same drive with the same name as the source file.

Copying A File (two disk drive system)

1. If you are not already in the DOS 2.5 menu, type **DOS** and press **<Return>**.

2. Press **C** to "Copy a File".

3. When the computer asks for source and destination filenames, respond with the filename of the file to be copied, followed by a comma and the name of the new file copy. Note that there are no spaces between the filenames and the comma. An example is: *D:LIB,D2:LIB*

Deleting a File

To delete a file from a disk, you must use the "Delete File" command found in the DOS menu. Always be absolutely certain that you really want to delete a file before you do it. Once a file is deleted, it is gone forever.

Deleting A File

1. If you are not already in the DOS menu, type **DOS** and press **<Return>**.

2. Press **D** for "Delete File".

3. When prompted, enter the filename of the file to be deleted.

4. The computer will ask if you really want to delete the file. If you do, type **Y** and the file will be deleted from the disk.

Listing a Disk Directory

When you want to see what files are contained on a disk, you must use the "List Directory" command found on the DOS 2.5 menu. Try this series of commands with a Kyan Pascal backup disk.

Listing A Disk Directory

1. If you are not already in the DOS menu, type **DOS** and press **<Return>**.

2. Press **A** for a "Directory List".

3. Press **<Return>** to list the files on drive 1 to the screen. Type **Dn:** (n being the drive number) to list the files on another drive. If you follow the directory device name with a comma and another device name, such as **P:** for the printer, the directory will be sent to that device.

CONCLUSION

If you are new to DOS 2.5, this section should have provided you with enough information to begin writing and saving your Pascal source code files. If you still feel a little confused, reread the section, or obtain a DOS 2.5 reference manual.

Final Note: Often during the use of DOS 2.5, the computer will ask if you want to cancel "MEM.SAV." This file is not needed with Kyan Pascal and you should respond with a **Y** to delete this file from the disk.

Using and Configuring Kyan Pascal

Kyan Pascal is designed to run on most Atari 8-bit systems. The software will work on any 48K Atari -- from an expanded Atari 400, to a 130XE, to a radically upgraded one-meg "super Atari." Only one disk drive is needed, although the software can be used with extra disk drives, hard disks, and/or RAMdisks.

To start programming with Kyan Pascal, first boot the DOS 2.5 operating system by turning on the power and inserting Side 1 of the Kyan Pascal disk (a backup copy!). DOS 2.5 will load and then Kyan Pascal. The Main Menu will appear on the screen and you are ready to begin programming.

Kyan Pascal is shipped on a "flippy" disk, i.e., files are included on both sides of the disk. Since Atari disk drives can only read one side of a disk, you must "flip" the disk to read the files on the second side. To flip the disk, you must remove the disk from the drive, turn it over, and re-insert it.

Kyan Pascal -- Side 1

Side 1 of the disk contains the DOS 2.5 operating system, the Kyan system files (PC, ED, AS, STDLIB.S), and the KIX commands. Side 1 is the disk you will use as your source disk.

Kyan Pascal -- Side 2

Side 2 of the Kyan Pascal disk contains the Kyan Assembler, Pascal Runtime Library (LIB), and various Pascal "include" files (which will be used in programming with Kyan Pascal)

Side 2 also contains two demonstration programs (both source and object code versions). To run these demo programs (PRINT and PRIME), boot Side 1 and press <RETURN>; then, insert Side 2 into the disk drive, enter the filename of the demo program (be sure to enter the OBJECT CODE filename which has no ".P" extension), and again press <RETURN>. The program will now run. To examine the demo source code, use the Editor to load the demo source code file.

Single Drive Users

Single Density: Place Side 1 of the Kyan Pascal disk in the drive and boot; you can now call any of the programs listed on the Menu, but, since the disk is full, you cannot store any of your programs on this side. You must use Side 2 to load, save, and/or run your programs. When you want to call any Kyan system program, always remember to remove Side 2 and re-insert Side 1 in the disk drive. (**Important Note:** Due to hardware limitations, single disk/single density users may not be able to effectively use some KIX commands (e.g., LS).

Enhanced Density: With the extra capacity of the enhanced density disk, you can store your own programs on Side 1 of the disk along with the Kyan files (thus eliminating disk swapping). If you choose this option, be sure to copy the Kyan Runtime Library (LIB) from Side 2 to Side 1 of the disk along with any "include" files used in your programs.

Multiple Drive Users

With more than one drive, you will find it most convenient to store your Pascal programs on a separate program disk. Place the Kyan Pascal disk (Side 1) in drive 1 and leave it there. Place your program disk, with copies of include files and the Kyan Pascal Runtime Library (LIB) in drive 2.

When you boot Kyan Pascal, Drive 1 (**D1:**) will automatically be set-up as the default drive. To set-up the second drive as the default device, use the KIX command CD (i.e., **CD D2:**). Now, all of your Pascal programs will be called from and written to drive 2.

RAMdisk Users

All of the Kyan Pascal system files (ED, PC, AS, STDLIB.S) and most of the KIX files can be loaded into RAMdisk. To load these files, use the KIX copy command "CP" (e.g., **CP D8:ED**).

The "DUP.SYS" file which is part of DOS 2.5 is automatically loaded into RAMdisk on boot-up. If you do not want to use DOS 2.5 (i.e., you want to work with KIX commands exclusively), you should delete this file from RAMdisk to free-up space for more KIX commands.

Searching for Files

The KIX system automatically searches for Kyan system files in up to three locations. First, it looks in RAMdisk (device D8:); if not found there, it looks for the Kyan file in the User device (e.g., D2:); and, finally, it looks in the system device (D1:). Thus it isn't necessary to have a copy of Kyan system files in the current (or default) directory.

II THE EDITOR

OVERVIEW

Kyan Pascal includes a full-screen, insert mode editor that you can use to write and edit programs. Insert-mode editing requires that you place the cursor where you want to enter text and begin typing. If you are editing existing text, place the cursor where you want to begin inserting text and start typing; if you are entering new text, just begin.

This section explains how to use the text editor to:

* Enter the EDITOR
* Create a file
* Edit an existing file
* Use the cursor control commands to control printing
* Delete and move lines of program text
* Use the special functions menu

The Editor has a HELP screens to assist you with the Control-Key commands. To look at the HELP screen, type the letter "H" when you are at the Editor Menu. The <ESC> key allows you to toggle between the Editor Menu and your text screen.

Note: The Kyan Pascal compiler is fully compatible with any text editing program which generates standard ASCII text files. You can therefore use your favorite text editor or word processor (e.g., AtariWriter) to write and edit programs.

SOURCE CODE VERSUS OBJECT CODE FILES

When you write a Pascal program using the Editor, you create a text file which is known as *source code*. When you name a source code file, you should append a ".P" to the filename to indicate that it is a Pascal source code file. When this file is compiled, a machine code file is created which is known as *object code*.

The object code file is saved on the disk along with your source code. The object code file has the same filename as the source file but without a ".P". If you look at a disk directory, you will see both the source code file ("MYPROG.P") and the object code file ("MYPROG"). Later, when you want to run the program, be sure to specify the object code, not the source code file.

ENTERING THE EDITOR

When you boot Kyan Pascal, you will see the Kyan Pascal Copyright screen. After a few seconds, you will see the Main Menu screen. The first command on the menu is **ED**. You will use this command to enter the EDITOR and begin writing the program.

What happens next depends upon whether you are creating a new file or editing a file that already exists.

CREATING A FILE

Once you have entered the Editor, the message appears:

Filename?

At this point, you can enter the filename identifier of any file -- even if it doesn't exist. You must remember, however, to include the device name (i.e., D1:, D2:, D8:, etc.) in the filename. The KIX operating environment will use the default device (usually set to D1:) if no other device name is used before the filename.

Since this is your first Pascal file, enter the filename **D1:TRIAL.P** (or whatever drive you are using for your user disk in place of D1:). Since your file doesn't really exist yet, the editor will respond with the message

FILE NOT FOUND.
PRESS ANY KEY TO CONTINUE.

When you press a key, the screen will go blank and the blinking cursor will appear in the home position (i.e., the upper left hand corner of the screen).. All text that you enter will now become part of your program file. Press <**RETURN**> at the end of each line of program text. When you have finished typing the program, press <**ESC**> to get to the Editor Menu and then either the **S, X, or Q** key. You don't need to do this now since you haven't written a program yet.

For practice, however, try entering the following program. Don't bother saving or even trying to compile it. Just make sure that you can enter it.

```
PROGRAM  Trial(Input,Output); <RETURN>
<RETURN>
BEGIN <RETURN>
  Writeln('Hi, This is Kyan Pascal.')  <RETURN>
END. <RETURN>
```

EDITING AN EXISTING FILE

To edit an existing file, follow the same procedure for creating a file. First select ED from the Main Menu. This time, however, when the screen requests a filename, enter the name of a file that already exists on the disk. Your existing file will be loaded into memory and appear on screen, ready for any changes you wish to make to it.

The Kyan text editor is an insert editor. Whenever you enter text, it appears where the cursor is positioned. If text exists after the cursor, it will be moved forward to make room for the new text as you type. As you will see, it is important to position the cursor correctly when you edit the text of a program. To make editing text easier, Kyan Pascal has a number of control-sequence commands that let you

* move the cursor
* delete text
* move blocks of text

A control-sequence command is executed by pressing the <CONTROL> key while simultaneously pressing the appropriate letter key. The keys must be pressed at the same time -- they are **not** entered one after the other. For example, <CONTROL>-S means that you should press the <CONTROL> key and the letter S at the same time (i.e., just like pressing <SHIFT>-"5" for the "%" character).

Cursor Movement Commands

The following control-key commands let you move the cursor to different positions in the text.

<CONTROL> - "+" moves the cursor 1 space back (to the left)

<CONTROL> - "*" moves the cursor 1 space forward (to the right)

<CONTROL> - "-" moves the cursor 1 character up

<CONTROL> - "=" moves the cursor 1 character down

<CONTROL> - "L" moves the cursor 1 word back (to the left)

<CONTROL> - "R" moves the cursor 1 word forward (to the right)

<CONTROL> - "U" moves the cursor 20 lines back (up)

<CONTROL> - "D" moves the cursor 20 lines forward (down)

<CONTROL> - "CLEAR" moves the cursor to beginning of text

<CONTROL> - "E" moves the cursor to the end of text

Try using these commands to move the cursor around a file. At this point it doesn't matter if you are actually writing a program. Just enter text and get comfortable moving the cursor around the screen.

Notice that the control key sequences to move move the cursor one character left, right, up, and down are the arrow keys. You can hold down a control key sequence and it will repeat automatically until you release the keys.

If you are editing and cannot remember a cursor control command, just escape to the Editor Menu and press "H". A HELP screen will appear which lists the control key sequences and the actions.

Delete Commands

Text can be deleted from you file by using the following commands (note that DEL/BS is the DELETE BACK SPACE key):

<CONTROL> - DEL/BS	deletes the character the cursor is on and pulls the text following the cursor one character back.
DEL/BS	deletes the character to the left of the cursor and pulls the text following the cursor one character back.
<SHIFT> - DEL/BS	deletes the entire line the cursor is on.

Note: The <CONTROL>-"M" command (Move to buffer) works well for deleting large amounts of text. This command is described in the next section.

Move Text Commands

You may find that you want to delete, copy, or move an entire section of text in your file. Each of these activities is accomplished with "cut and paste" commands. Each activity starts by defining or "cutting" a block of text using the **<CONTROL> - M** command. Then, depending on the desired end, the block is "pasted" in the location(s) specified using the **<CONTROL> - P** command.

Just for practice, enter some text in the file. If you are already working in a file, decide what text you want to delete, copy, or move. Don't worry that this is not a real program. Remember that you are just learning how to use the text Editor commands. Once you have text printed on the screen, try some of the following commands.

CUTTING A BLOCK OF TEXT

1. Move the cursor to the beginning of the block you want to cut.

2. Press <CONTROL> - M.

3. Move the cursor to the end of the block. The block will be highlighted as you move the cursor.

4. Press <CONTROL> - M again. The block will disappear, but don't worry. The block is stored in the computer's memory. [Note: The cut buffer will hold up to 2K of text.]

DELETING A BLOCK OF TEXT

If you only want to delete the block, you are finished. In the preceding step, the block has been cut from your text file and stored in a memory buffer. If you don't want it anymore, just leave it there! The buffer will be erased when you perform the next "cut" command or when you exit from the editor.

COPYING A BLOCK OF TEXT

1. Copying the block consists of pasting the cut block in one or more locations. Press <CONTROL> - P. This will paste a copy of the block back in its original location.

2. Move the cursor to the next place where the block should be pasted and press <CONTROL> - P. Another copy of the block will be printed there.

3. Repeat the paste procedure as often as you want.

MOVING A BLOCK OF TEXT

1. Moving the block consists of pasting the cut block in a new location. Move the cursor to the location where you want the block pasted.

2. Press <CONTROL> - P. The block will reappear in the new location.

Try cutting and pasting text until you are comfortable with the procedure. You should become skilled enough so that you don't worry about losing text every time you try to move lines of your program.

EDITOR MENU

Whenever you are entering program text, you can access the Editor Menu. This menu lists the commands which allow you to change characteristics of the program without losing the data you have already written. Don't worry if you need to call the Editor Menu while you are in the middle of typing a program. Your program will still be there when you return from doing whatever you decide to do.

To enter the Editor Menu, press <ESC>. Try it. You won't lose the text you are working on. Press <ESC> again. The text of your program reappears on the screen. You can keep pressing <ESC> to alternate between your program text and the Editor Menu.

The Editor Menu allows you to make changes that affect the entire program. You can:

* Get HELP
* Change the filename of the file you are working on
* Insert another file in the text
* Go to a specific line number in your program
* Change a string of characters in your program
* Find a string of characters
* Save your text and leave the Editor

Editor Menu

When you are editing text, you access the Editor Menu by pressing the <ESC> key. The following menu appears on the screen.

```
                EDITOR COMMANDS
-------------------------------------------------

Filename: D1:MYFILE.P

A:
B:

<ESC> to Resume

COMMAND          DESCRIPTION
    H            Cursor Control Help
    X            Save File & Exit Editor
    S            Save File & Resume Editing
    F            Set Filename
    Q            Discard File & Quit
    I            Insert File
    G            Go To Line Number
    A            Set "A" String
    B            Set "B" String
    C            Change A Strings to B Strings
```

To select an item from the menu, press the key that represents the selection.

For the present, we'll skip the Save commands. If you want to see the Help menu, press "**H**". The other commands are explained below. Each function description has a PROCEDURE that illustrates how to use the function. To try these procedures, you must have some text entered on the Editor screen. Since you won't be running a program, for the time being, you can use any text you want when you try the procedure.

SET FILENAME

This command lets you rename the file you are working on. You can change just the filename, or you can also change the volume if you are going to save the file on a different disk

CHANGING A FILENAME

1. Press **F**. The following request will appear:

   NEW FILENAME (BLANK TO QUIT) ?

2. Type the new filename (Device:Filename). If you want to quit and leave the Filename unchanged, just press <**RETURN**>.

3. After entering the new filename, press <**RETURN**>. The new filename will appear at the top of the screen.

INSERT FILE

This command lets you insert another file into the current program.

INSERTING A FILE

1. Press <**ESC**> to leave the Editor Menu and return to your program.

2. Move the cursor to the point in your program where you want the file inserted.

3. Press <**ESC**> to return to the Editor Menu.

4. Press **I**. The following request will appear:

   FILENAME OF FILE TO INSERT?

5. Type the filename of the file you want to insert and press <**RETURN**>. To cancel the process press, <**RETURN**> without entering a filename. You will be returned to your text.

GOTO line number

This command lets you move the cursor immediately to a line number in the file which you specify. The GOTO command is useful when you are working on a large program and you need to position the cursor.

REMEMBER: Line numbers should not be entered as part of the program; the number specifies the line on the editor screen or on the printed listing of the program file.

GOTO

1. Press G. The following request will appear: *LINE NUMBER ?*
2. Enter the number of the line you want the cursor moved to.
3. Press <**RETURN**>. Your program will appear on the screen with the cursor positioned on the line you have specified.

CHANGE STRING

This command lets you change some or all occurrences of a string used in your program. This command is equivalent to the "Search and Replace" function in other text editors. For example, you might want to change the expression **A+B*C** to **A+B*D**.

The Change String function distinguishes between upper and lower case letters. For example, a search for "**CAT**" will not find the word "**cat**". The maximum length of the string is 40 characters.

CHANGING STRINGS

1. Press A. The screen will display: *A:*
2. Type the string you want to replace, exactly as it appears in the text, and press <**RETURN**>.
3. At the bottom of the screen, you will see the string appear:

   "A" String is:

4. Press **B**. The screen will display: *B:*

5. Enter the new string to replace the old one and press **<RETURN>**.

6. At the bottom of the screen, the new string appears:

   "B" String is:

7. Now press **C**. The following prompt will appear:

   CHANGE ALL STRINGS OR SOME
   (A/S/Q)?

   You have 3 choices.

   A causes ALL occurrences of the string to be changed. After pressing **A**, the program is displayed with the new string.

   S lets you change SOME occurrences of the string. The program will be displayed with the cursor positioned on the first occurrence of the string you want to change. If you want that string changed, press **Y**. If you do not want to change this occurrence of the string, press **N**. The cursor advances the next occurrence and you can repeat your choice. This continues until no more instances of the string can be found.

   Q lets you QUIT the Change String function and returns you to the Editor Menu where you can select another function.

FIND STRING

This command lets you find a particular string in your program. It is like the search and replace function, but it just locates the desired string without changing it.

FINDING STRINGS

1. Press A. The screen will display: *A:*

2. Enter the string you want to find and press <ESC>
   NOTE: Be sure to Press <ESC> and NOT <RETURN>.

3. You can now search forward or backwards through the file by pressing either <CONTROL>-Z or <CONTROL>-W.

   <CONTROL>-Z moves the cursor forward through the file to the next occurrence of the string. Pressing the control sequence continues the search until you reach the end of the program.

   <CONTROL>-W moves the cursor backward through the file until it finds the next occurrence of the string. Repeating the process continues the search until you reach the beginning of the program.

Saving Files And Quitting

The Editor Menu gives you 3 ways to save your program and leave the Editor mode. By making the appropriate selection, you can

* save your program and exit the Editor mode
* save your program but return to it for more editing
* quit the Editor without saving the program

Whenever you save a program, it will be saved with the filename specified at the top of the Editor Menu.

Remember, when you are in the process of writing or editing a program and you want to save it, press <ESC> to get the Editor Menu.

If you have been editing an existing file and save it under its existing filename, the most recent version will overwrite the old version, and the old file will be

lost. If you want to save both versions of the program, select the CHANGE FILENAME function and change the filename of the latest version of the program before you save it.

Once you have set the filename that will identify your program on the disk, select one of the Save and Exit options.

SAVE AND EXIT Press X

This option saves your file under the filename indicated at the top of the menu and exits the editor. You are returned to the system prompt.

SAVE AND RESUME Press S

This option saves the file under the filename indicated at the top of the menu and returns you to the program. (NOTE: You should use the SAVE AND RESUME selection often while you are writing a program. This will insure that you always have a fairly recent copy of the program in case a problem develops with your computer or disk drive.)

DISCARD AND QUIT Press Q

This option exits the Editor without saving the program. When you select this option, the following message appears:

```
THE CHANGES YOU HAVE MADE HAVE NOT BEEN SAVED.
ARE YOU SURE (Y/N) ?
```

NOTE: Press Y only if you are absolutely certain that you do not want to save a copy of the current program. You will be returned to the operating system and will see the system prompt.

Press N if you decide that you don't want to discard the file. You will be returned to your file in the Editor.

CONCLUSION

You should now be fairly comfortable with the text editor and the cursor control commands. You should also be able to use the Editor Menu to rename, include, or save files that you write, as well as locate lines and change strings.

HELP Screens

The Editor Menu contains a HELP screen which can assist you if you forget one of the control key commands. You can call the HELP screen by pressing "H" in the Editor Menu.

Other Text Editors

The Kyan Text Editor is simply a word processing program that lets you enter source code in a form that the Pascal compiler can understand and translate into object code. You may already have a word processing program which you are familiar with and prefer over the Kyan Editor. If this is the case, feel free to use your own editor. **The only requirement is that your editor must generate a sequential ASCII text file.** Otherwise, the Kyan Pascal compiler will not be able to read your Pascal source code file.

This page left blank for your notes.

III KYAN PASCAL

Kyan Pascal contains many features that are not part of standard Pascal. These features make writing Pascal programs even easier; they also make the programs more powerful.

This section explains how to

* Compile Source Code Programs
* Direct Input and Output
* Include Other Files in a Program
* Chain Files
* Declare and Manipulate Strings
* Use Graphics and Sound
* Create "Stand-Alone" Disks and "Auto-Run" Files
* Run a Program

If you are new to Pascal programming, read the section on compiling a source code file. Then skip the rest of this section until you have studied the Tutorial.

If you already know Pascal, read this section to become acquainted with Kyan Pascal's special features.

THE PASCAL COMPILER

SECTION II explained how to use the Text Editor to write and save a Pascal source code program. This program contains all the program logic and Pascal syntax. The computer, however, can not understand the program as it now exists. Your Source Code or Source Program must be translated into statements that make sense to the computer itself. Another program must translate the Source Code into Object Code. This translator is called a Compiler.

The Pascal Compiler reads the filename of your program and then, after locating the file, translates all the Pascal statements into what is known as

Assembly Code. A second software module, known as the Assembler, then translates the Assembly Code into Object Code which the computer can run.

When the Compiler creates an Object Code file from a Pascal source code file, it identifies the new file by deleting the **.P** extension to the name of the Source file. If you compile a program named **MYFILE.P**, the Object Code file will be named **MYFILE.**

COMPILING A PROGRAM

To compile a program, first enter the Main Menu.

NOTE: If you are in the Editor, save your current program and exit the Editor. When you see the system prompt (%), type MENU. If your are in some other menu, follow the directions at the bottom of the screen until you are back in the Main Menu. If you are using a single drive system, you should make sure that Side 1 of the Kyan Pascal disk (or the disk with the Kyan system files) is in the drive. If you are using multiple drives, remember to use complete filenames to get back to the Main Menu.

When the Main Menu appears, it looks like:

```
Prefix:  D1:
-------------------------------------------------------
KYAN PASCAL MAIN MENU

Option          Description

ED              EDITOR
PC              PASCAL COMPILER
AS              ASSEMBLER
DOS             DOS 2.5 MENU
KIX             KIX COMMAND MENU
-------------------------------------------------------
Press <RETURN> to enter a command
```

Note the device prefix at the top of the menu. This is the default prefix when you boot the Kyan.Pascal Disk.

To begin the process of compiling a program, press <**RETURN**> and type **PC**. The Compiler Menu will appear on the screen. This menu lists the options that are available with the Compile command.

```
**********************************************************
                    Kyan Pascal Compiler
          Copyright, 1986, Kyan Software Inc.

COMPILER OPTIONS

 Option          Description

-D               Emit line number and filename on Runtime
                 error.  Default:  Do not generate.

-O Filename      Give the compiled file a new filename.
                 Default:  Object file is assigned the source
                 filename less the ".P" extension.

 -P              Show progress of compiler and assembler.

-S               Generate intermediate assembly language
                 file.  Do not assemble into object code.
                 Default:  Generate object code file.

>Device          Direct compiler error listing to Device.
                 Default: output to screen only.  Alternately,
                 specify filename and save as text file.

        Enter Device:Filename and Options
        PC:
**********************************************************
```

NOTE: If you do not specify an option, the compiler automatically assumes the option default.

The Compiler Options

The compiler menu lets you select options that affect the output of the compiled code.

-O Device:Filename

Assigns a new filename to the compiled file. Use it to redirect the compiled file to another disk or rename the compiled program.

> Device:Filename

Directs an error listing to the device and filename specified. A file saved to disk would be denoted: "D2:ERROR.LST". Alternately, a listing may be sent to the printer by specifying "P:". The default device is "S:", or the screen. Error listings identify Pascal syntax problems which the Compiler encounters when it tries to translate source code into Assembly code. If syntax errors are detected, you must re-edit and recompile the source code.

-S

The compiler generates an Assembly code source file but does not assemble the file (i.e., it does not generate an object code file). The assembly language file will have the filename "P.OUT". To save this file, use the DOS 2.5 or KIX rename command and add a ".S" extension to the filename (e.g., MYPROG.S).

-D

This compiler option is useful for debugging Runtime errors in the program. When this option is specified, Runtime errors will return the filename and line number of the offending statement.

-P

This option lets you observe the progress of the compiler and assembler. First, a file is opened which is the intermediate assembly language file ("P.OUT"); then, the Pascal source code is compiled (dots are printed on the screen); then, P.OUT is closed and assembly begins. With this option you can observe the relative speed of DOS 2.5 (in opening and closing the files), the Pascal compiler, and the macro assembler.

When you decide which options you want to use when compiling the program, follow these instructions. (NOTE: The Device:Filename and option specifications can appear in any order).

1. Enter the Device:Filename of the Pascal source code file.

2. To change the Filename, use the **-O** option.

3. Include a **-S**, **-P** or a **-D** for the other options.

4. If you want to redirect error listings, conclude the options list with the **>Device:Filename** directive.

IMPORTANT NOTE: During the compilation of a Pascal program, the compiler calls the following files: your Pascal program source code file; any "included" files called in your program; the Kyan STDLIB.S macro file; and AS, the Kyan macro assembler. All of these files must be found in the RAMdisk or on the current working device for the program to compile. If a files is not found, an error message will be printed or the compiler will crash!

Examples

Suppose you want to compile the file named **MYPROG.P** which is stored on the disk in drive 1 (D1:). The following illustrates the use of compiler option commands which could be entered after the **PC** prompt.

1. **MYPROG.P**

   Compiles **MYPROG.P** with all the default options.

2. **MYPROG.P -O D2:MYPROG**

   Compiles **MYPROG.P** from the default device (usually D1:) and writes the object code file, MYPROG, to disk drive 2 (D2:)

3. **MYPROG.P -S >P:**

   Creates an Assembly language source code file named **P.OUT** and directs error messages to the printer. No object code is created.

ERROR MESSAGES

If the compiler discovers any syntax errors in the program it is trying to translate, it does not produce Assembly code. Instead, it generates a list of lines containing errors. Each line is identified by its position in the program. (Note: Line numbers are not used in Pascal programs as they are used in BASIC. The line numbers in error messages are only used for identifying a line of source code where an error exists.)

After the line number, the line itself is displayed. A caret (^) indicates where the error occured in the line, and an error message describes the type of error which the compiler identified.

For example, if you tried to compile a program containing the following statement in the tenth line of code

```
Writeln('Error illustration').
```

the compiler would return the following error message:

```
ERROR ON LINE 10 OF FILE "MYPROG.P"
Writeln('Error illustration').
                              ^
";" OR "END" EXPECTED
```

The compiler error message indicates that the tenth line of code contains an error. It then says that the line should end with a semicolon or that there should be an END statement before the period.

As you gain experience compiling programs, you realize that a single error often generates an entire series of syntactical errors. For example, the compiler counts up the number of opening and closing segment indicators to make certain that there is an equal number of both-- BEGIN/END statements, are a prime example. If you omit any part of a pair of such statements, every BEGIN/END pair and every period may cause an error in the program.

Since the compiler checks only for syntactical errors, you may write a program that produces an object code file, but then produces unexpected results or even fails to run. The problems may be threefold.

1. You may have written the formula for a circle as "r*r" instead of "3.14*r*r" . This mistake will simply cause the program to

miscalculate results.

2. You may have introduced errors that cause the Assembler to crash. These are called Assembler errors.

3. Or, finally, you may have introduced system errors that only become apparent when the program runs. These are called runtime errors.

NOTE: For a complete list of error messages and their meanings, please refer to the Error Message List in the Appendices.

OUTPUT CONTROL

Kyan Pascal has special features that allow you to list a source code file on the printer and redirect output within a program to the printer.

Printing Source Files

Often, you will want a hard copy of the program you are working on. It's always easier to spot mistakes on paper than it is to see them on the screen. Kyan Pascal offers you two options for printing a listing of your programs. The first option is to use the **CAT** command which is described in the KIX section (Chapter VI) of this manual. The second option is to use the **PRINT** program which is described below.

The **PRINT** program allows you to print a listing of any source program. You can load and run this program whenever you see the system prompt. If you are editing a program, press X to save the file and exit the Editor. If you are in the Main Menu, type <RETURN> to get the **%** prompt.

To print any text file enter "**PRINT**" at the **%** prompt. The computer will load the program and display the prompt "**Filename:**". Enter the complete filename of the program you want to print. Make sure that the printer is turned on before you execute this command.

After you enter the filename, the program will ask if you want program line numbers printed. It is usually best to select this option. When you compile

the program, the Compiler lists the line number and the line containing the error. If you already have a copy of the program with the lines numbered, it is easier to locate the lines with errors.

Redirecting Output Within a Program

Kyan Pascal allows you to redirect program output from the screen to the printer. You can redirect the output back and forth as many times as you want.

Before you can redirect output within a program, however, you must first set-up a file called "printer". Then, declare a variable "printer" as a file of text (i.e., VAR Printer:Text;). In the body of your program (before your first use of output redirection) insert a Rewrite statement which defines this file to be the printer (e.g., Rewrite (Printer, 'P:');). Then in the body of your program, use the following command syntax to direct output to the printer:

Write or Writeln (Printer, '....text....');

If you want to direct output to a second printer, simply insert a new Rewrite statement (e.g., Rewrite (Printer, 'P2:');). Output will then be directed to printer 2.

See the second lesson in the Tutorial for a sample program that illustrates the redirection output to the printer.

INCLUDE

Kyan Pascal lets you "include" other procedures, functions, or text files within a program. This include capability allows you to create a library of frequently used functions and procedures.

The Include feature also allows you to incorporate routines found in Kyan's Advanced Graphics, System Utilities, and other programming toolkits into your own Pascal programs.

Including Subroutines

To "include" a subroutine, put the INCLUDE statement in the declaration of the program between the list of variables and the body of the program that calls the included subroutine.

#i Device:Filename

The pound sign (#) and the letter I (or "i") must appear in the first two columns. If you use the filename without a device prefix, the compiler will use the default prefix.

For example, assume the following procedure has been saved on drive 2 with the filename **HELLO.I**.

```
****************************************
PROCEDURE HELLO;

BEGIN
  Writeln('Hello, World.')
END;
****************************************
```

You can include the PROCEDURE HELLO in the following program as follows.

```
PROGRAM  Main;

#i D2:HELLO.I    (*  full device:filename is specified  *)

BEGIN
  Hello
END.
```

When including files, you must obey the Pascal rules of syntax. The included file(s) (like Functions and Procedures) must be located between the variables list and the body of the program. To help avoid syntax errors, try to visualize the included lines of text inserted in the main program in place of the include statement. If the resulting program follows Pascal rules of syntax, then the block is properly included.

Once you know how to include other files, you can begin including the special Kyan procedures and functions that are contained on the Kyan.Pascal disk. This enables you to use special String and Graphics features.

STRINGS

String is not a predefined Pascal data type. But if you declare a String as an array of characters, you may use the following string subroutines which have been predefined in Kyan Pascal.

Length
Index
Substring
Concatenate

To use a string manipulation routine, declare a String, declare the maximum size of the string as a Constant named **Maxstring**, and include the string routines you will use.

The following example creates a String ten characters long and then includes the Length routine which can be called from within the body of the program.

```
PROGRAM SampleString;

CONST
  Maxstring = 10;

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

VAR
  Line : String;

#i D1:Length.I

BEGIN
  ...
END.
```

The value of Maxstring can be any size that meets the needs of the program. It cannot, however, exceed the maximum integer size for Kyan Pascal (MAXINT = 32767).

Length

The Length routine returns the actual length of a string. This routine assumes it has reached the end of the string when it encounters either the first blank space or the last character of a full string.

Use the Length function to test strings or to eliminate trailing spaces when you print the string. The following program illustrates both of these applications and the use of the include file **Length.I**.

```
=================================================
PROGRAM   DemoLength;

CONST
  Maxstring = 10;

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

VAR
  s : String;

#i D1:Length.I     ( * predefined Kyan routine * )

BEGIN
  s := 'abcd      ';
  Writeln(s: Length(s));
    Writeln('The string is ', Length(s), ' characters long')
END.
=================================================
```

The string in this program is defined as an array of 10 characters. The actual string defined in the program, however, contains only four characters. The Length function is used to eliminate the trailing 6 spaces when the string is printed on the screen. The function is also used in the final Writeln statement.

Index

Index returns the position of one string within another string. To use the Index function in a program, the file **INDEX.I** must be included in the program declaration.

The actual function used in the body of the program takes two parameters, String1 and String2.

As an example, a program declares a string as 10 characters long. It then declares S1 and S2 as string variables. If the program defines the substring S2 as 'a ' and the string S1 as 'baby ', the statement "**Index(S1,S2);**"returns the value, 2.

Note that the letter "a" is the second character in the second string. If S2 is not found in S1, the Index returns zero.

Substring

Substring extracts part of a string. The file **SUBSTRIN.I** must be included in the program declaration. The Substring statement takes four parameters: the source string variable; the destination string variable; the position in the string where the substring begins; and, the length of the substring to be extracted. Its syntax is

Substring(String1,String2,Begin,Length);

If the length of the substring is less than Maxstring, trailing spaces are added to the substring. For example, if Maxstring is defined as 10 and the source String variable , S1, is defined as 'abcdef ', the statement "**Substring(S1,S2,2,2);**" returns the substring 'bc '.

The substring begins at the second character in the string and extracts two characters. The length defined by Maxstring adds the trailing spaces. You can use the Length function with the Substring function if you want to eliminate the spaces when writing the substring. Example,
"**Writeln(Substring(S1,S2,2,2): Length(Substring(S2)));**"
prints only the characters "bc".

Concat

Concat is an abbreviation for "Concatenate," which means to join two strings in a third string. To use this procedure, the file **CONCAT.I** must be included in the program declaration.

The statement takes three parameters: the first string variable; the second string variable; and, the result string. Its syntax is

```
Concat(String1,String2,String3);
```

If Maxstring equals 10, and S1 equals 'Any ', and S2 equals 'Body ', then the statement "Concat(S1,S2,S3);" produces a third string, S3, that equals 'AnyBody '. Note again the trailing spaces which may be eliminated by using the Length function in conjunction with Concat.

GRAPHICS

Kyan Pascal supports the Atari's ability to use graphics with many different resolutions and a multitude of colors. Routines are available to set graphics modes, set the screen colors, draw lines, and more.

To use one of the graphics-related routines on the Kyan Pascal disk (Side2), it must first be included using the #i directive. The graphics files on the disk are:

GRAPHICS.I	PLOT.I
DRAWTO.I	SETCOLOR.I
LOCATE.I	POSITION.I

Relocating the Program Origin for Graphics

Because graphics use a lot of memory (as much as 8K) and may interfer with your Pascal program, it is necessary to relocate the origin of your program in memory before using graphics. This is done by inserting the following lines of assembly language code at the very beginning of your Pascal program (i.e., before the "PROGRAM" statement).

```
#A
_ORIGIN EQU $4000
#
```

This code is an assembler directive which moves your program up in memory so that the graphics below it do not overwrite your program.

Different graphics modes require different amounts of memory. For example, low resolution mode (mode3) requires only 400 bytes; super high resolution (mode 8), however, requires nearly 8K of memory. You can change the program origin to a lower location in memory if you are using a graphics mode which does not require a lot of memory.

How to Set the GRAPHICS Mode

The GRAPHICS.I file, not surprisingly includes the procedure "Graphics." Call it in the following manner:

Graphics (Mode_Number);

Mode_Number is an integer that corresponds to the particular mode you want set. The number 0 is the text mode. When you want to use a mode other than zero, you must call the following procedure immediately before your call Graphics (Mode_Number).

```
PROCEDURE Use_Graphics;
BEGIN
#A      (* Be sure the # sign is placed in column 1 *)
   LDA #<_origin  ; get MSB of program starting location
   STA $6A        ; and store in RAMtop (106).
#
END;   (* of the Use_Graphics procedure *)
```

To return to the text mode, you must call Graphics (0). Before doing so, you must call the following procedure.

```
PROCEDURE Use_Text;
BEGIN
#A      (* Be sure the # sign is placed in column 1 *)
   LDA #$C0       ; put MSB of highest RAM
   STA $6A        ; location :C000 into RAMtop (106)
#
END;   (* of the Use_Text procedure *)
```

After calling this procedure, you may then immediately call Graphics (0).

How to PLOT and DRAWTO

First, you must include the files PLOT.I and DRAWTO.I in your program. Then, you can call PLOT (to plot a point) in the following manner:

```
Plot (X_Number, Y_Number, C_Number);
```

X_Number and Y_Number are integers that correspond to the horizontal and vertical position, respectively, of the point you want to plot. C_Number is an integer that corresponds the color of the point being plotted.

You can call DRAWTO in the following manner:

```
Drawto (X_Number, Y_Number, C_Number);
```

X_Number and Y_Number are integers that correspond to the horizontal and vertical position, respectively, of where you want your line drawn to. The line is drawn from the last position plotted. C_Number is also an integer; it corresponds to the color of the line being drawn.

How to Set the Screen Colors

First, include the SETCOLOR.I file. Then, call the procedure in the following manner:

```
SetColor (Register, Hue, Luminance);
```

The parameters Register, Hue, and Luminance are all integers.

How to Locate Data on the Screen

First include the file LOCATE.I. Then call the procedure:

Locate (X_Pos, Y_Pos, Data);

All of the parameters are integers. X_Pos and Y_Pos are the horizontal and vertical positions of the cursor, respectively. Data will be changed to hold the number of the character or color at the particular position you specify.

How to Position the Cursor

First, include the POSITION.I file. Then,, call the procedure in the following manner:

Position (X_Pos, Y_Pos);

X_Pos and Y_Pos are integers that correspond to the horizontal and vertical position of the cursor, respectively. This procedure is the same as GOTOXY, which is found in other implementations of Pascal.

Points to Remember about Graphics

When using any of the routines that deal with the position of the cursor on the screen, remember not to exceed the boundaries of the particular graphics mode you are in.

The Position procedure can be used in the text mode (Graphics 0), as well as in graphics modes, to position the cursor.

When your program uses graphics, make sure that you reset to Graphics 0 when you end the program; otherwise, garbage (nonsense) will appear on the screen and STAY THERE!

SOUND

To utilize the Atari's sound capabilities, there is a file SOUND.I on the Kyan Pascal disk.

Call this procedure in the following manner:

Sound (Voice, Pitch, Distortion, Volume);

All of the parameters are integers. They must be in the following range:

Parameter	Range
Voice	0 - 3
Pitch	0 - 255
Distortion	0 - 7
Volume	0 - 15

To turn off a voice, give its number and zero out the remaining parameters (e.g., Sound (1,0,0,0)).

NOTE: If the total volume of all the voices exceeds 32, the monitor speaker is likely to buzz and/or distort the sound output.

CHAINING PROGRAMS

Kyan Pascal features a Chain statement that links compiled files. When you chain files, the first file calls the object code file of the second program. Under certain conditions, it can pass variables to the second file. In essence, when two files are chained, the last command of the first program tells the computer to RUN the object code of the second program.

The syntax of the chain statement is

```
Chain('D1:FILENAME');
```

To use the Chain feature, put the Chain statement in the last line of the Main program just before the END statement. (Since control is immediately passed to the second program, more statements in the first would be irrelevant unless the second program was being linked as part of a conditional sequence and the second program itself chained back to the first.)

The first file can pass variables to the second file only if the global variables in the second file are declared in the same order and as the same data types.

An Example Of Chaining

In the following sample programs, the first uses the Chain feature to run the object code file of the second. The first program asks the user, a salesperson, to enter the name and price of a product. When the information is entered, the program chains to the second program. The chained program then requests the cost of the item to the company. Using this information and the price data that was passed to the program, it calculates the profit and displays the information on the screen.

The individual programs are very straight-forward. Note, however, that the first part of the variable declarations in both programs are identical. Only the last two variables in the second program's list and the Chain statement at the end of the first program are different.

First Program

```
PROGRAM Retail(Input,Output);

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

VAR
  ProductName : String;
  Price       : Real;

BEGIN
  Writeln;
  Writeln('What is the name of');
  Writeln('the product? ');
  Readln(ProductName);
  Writeln;
  Writeln('And what is the price in dollars ');
  Write('and cents?  $ ');
  Readln(Price);
  CHAIN('D1:PROFIT')
END.
```

Second Program

```
PROGRAM FindProfit(Input,Output);

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

VAR
  ProductName : String;
  Price, Cost, Profit  : Real;

BEGIN
  Writeln;
  Writeln(' > You have chained to program #2 < ');
  Writeln('What was our cost of the ', ProductName);
  Write('that you sold?  ');
  Readln(Cost);
```

```
  Writeln;
  Writeln('Okay, you sold a ', ProductName);
  Writeln('for $', Price:4:2);
  Writeln('It cost us $', Cost:4:2);
  Profit := Price - Cost;
  Writeln('We made a profit of $', Profit:4:2)
END.
```

Comments

1. Compare the declaration sections of both programs. They are identical until the second program adds two more variables. Note that these new variables, Cost and Profit, are declared after the variables which were passed.

2. The first program chains to the object code of the second with the filename **'PROFIT'**. It directs the computer find a file named **PROFIT** which was successfully compiled previously. This file may be stored on any device including D1: and D8: (RAMdisk). The Kyan Pascal compiler will automatically search all of these locations for the file to be chained.

3. The syntax of the Chain statement must be followed exactly. This includes the parentheses and single quotes which surround the filename.

4. A String Variable may be used as the filename for a Chained program.

5. The second program receives the variables and string information from the first. It then requests new information and, using both the new and the passed data, calculates the value of Profit.

Important Points About Chaining

When Chaining

1. The filename to the second program must specify an object code file, not a text file.

2. No statements in the first program will be executed after the Chain statement is called. The only exception to this rule is if the Chain statement is part of a conditional test and if the chained program itself is chained back to the calling program.

When Passing Parameters

1. The global variables must be declared in the same order in both programs. Note, however, that the second program can add additional variables after it duplicates the passed variables of the first program.

2. The data types of the variables must be identical.

How Kyan Pascal Stores Passed Parameters

1. Variables are stored in the variable stack. When parameters are passed from program 1 to chained program 2, they remain in the same stack location (i.e., variables stored in locations A, B and C in program 1 stay in those locations when passed to program 2). If new variables are declared in program 2, they will be stored in subsequent locations (e.g., locations, D, E, F).

2. The Stack begins in high memory and grows downward.

Other Notes and Features

AUTO-RUN PROGRAMS/STAND-ALONE DISKS

To create a stand-alone program that runs automatically when the disk is booted, you must follow these instructions.

1. Compile your program.

2. Copy the following files onto the formatted disk.

DOS.Sys	(from Side 1 of Kyan Pascal disk)
LIB	(from Side 2 of Kyan Pascal disk)
MyProg	(your object code file)
DUP.Sys	(Optional from Side 1 of Kyan disk)
RAMDisk.Com	(Optional from Side 1 of Kyan disk)
AutoMake	(from Side 1 of Kyan disk)
CAT	(From Side 1 of Kyan disk)

3. Insert the disk in drive 1 and enter the following command at the KIX prompt:

```
% CAT_MyProg_Automake_>AutoRun.Sys
```

These instructions append your program to the AutoMake file and creat a new file called AutoRun.Sys. (You may then delete MyProg and AutoMake from the disk if you need extra space.) Now, whenever this disk is booted, your program will execute after DOS is loaded.

CAUTION: DOS.SYS, the Kyan Pascal Runtime Library (LIB), and certain other software files are copyrighted products of Kyan Software and Atari Corp.. Use of these files on stand-alone disks is subject to restrictions outlined in the Software License Agreement found in the Preface to this manual. Please be sure to carefully read the Software License Agreement and understand the restrictions noted. Failure to comply with these restrictions may result in a felony violation of Federal Copyright Laws.

RUNNING A COMPILED PROGRAM

A compiled program can be loaded and run whenever you see the system prompt (%).

1. Make sure that a copy of the Pascal Runtime Library (LIB) and copies of any chained files are located on the same disk as the program to be run or that a device:filename is specified which indicates the location of these chained files.

2. Enter the full filename of the program and press <Return>. When you run the program, be sure that you specify the object code version of the file. The program will load and run. When finished, the system will return you to the prompt (%).

RANDOM NUMBERS

The Kyan Pascal disk contains a special routine which is used to generate random numbers. The file RANDOM.I must be included in the declarations portion of your program. Then, calling the function Random in the body of the program will return a random number between 0 and 1.

This routine is written in assembly language. To see a listing of it, use the Kyan text editor and, when prompted, enter the device:filename for the file (e.g., D1:RANDOM.I).

ADDRESS FUNCTION

ADDRESS is a special predefined, non-standard function used in Kyan Pascal. It returns an integer which is the actual address in memory of a variable. The function is frequently used when programming with special libraries such as Kyan's System Utilities and Advanced Graphics Toolkits.

The syntax for the function is: **ADDRESS (Variable.Identifier)**

PAGE PROCEDURE

Kyan Pascal supports PAGE as a standard predefined procedure. The function of this procedure is to skip from the current page to the next page of a Text file. The procedure causes the system to clear the file buffer by executing a WRITELN statement. It then advances the output to a new page of the specified text file or clears the screen and moves the cursor to home.

The syntax for the PAGE procedure is: **PAGE (File.Identifier).**

CONCLUSION

This section has introduced you to the unique aspects of Kyan's implementation of Pascal that make it efficient and powerful. It has explained how to:

* Compile Programs
* Control Output
* Include Files
* Manipulate Strings
* Create Graphics
* Chain Programs
* Create Auto-Run Files on Stand-Alone Disks
* Run a Compiled Program
* Use the Address function
* Generate Random Numbers

The only feature of Kyan Pascal not covered in this section is its use of memory. Since this information is required only by advanced programmers, it is included in Appendix B.

This page left blank for your notes.

IV TUTORIAL: PART I

This section contains 15 lessons that introduce you to the Pascal programming language. The lessons are divided into 2 parts. Part 1 covers the elements of a Pascal program and introduces the most important commands. Part 2 explains more advanced techniques used in writing Procedures, Functions, Records, and Files.

If you are unfamiliar with Pascal, read this section carefully. When you enter the sample programs, make certain that you enter them exactly as they appear in the text. A misplaced colon, semicolon, or period will prevent your program from compiling.

Before attempting to enter and run the programs in the Tutorial, you should read Sections I, II, and III which explain how to format a disk, use the text editor, and how to compile and run a Pascal program.

NOTE: FOR USERS WITH SINGLE DISK SYSTEMS

The Tutorial assumes that you are writing and saving your programs on the disk that you made in Section I. If you have not done so, read "Getting Started" and configure a disk for use with your system.

The disk created in Chapter 1 contains a limited amount of space because it also contains Kyan Pascal files. You may find that the disk becomes full as you work through the Tutorial. To avoid this problem, you should make several copies of the disk.

NOTE: SOURCE CODE VERSUS OBJECT CODE FILES

When you write a Pascal program, you create a text file which is known as *source code*. When you create a source code file, you should append a ".P" to the filename to indicate that it is a Pascal source code file. When you compile this file with the Pascal compiler, you create a machine code file which is known as *object code*. The object code file is saved on the disk along with your source code. The object code file has the same filename but without a ".P". If you look at a disk directory, you will see both the source code file (YOURPROG.P") and the object code file ("YOURPROG"). When you run the program, be sure to specify the object code and not the source code file.

The Tutorial covers the following topics:

Part I

1. The Pascal Program
2. Using Formulas
3. Decision Making
4. FOR Loops
5. Strings and Arrays
6. Boolean Variables
7. Scalar Data Types

Part II

8. Procedures
9. Functions
10. Scope and Nests
11. Arrays
12. Records
13. Sets
14. Files
15. Pointers

1. PASCAL PROGRAMS

If this is your first time writing a program in Pascal, you should pay special attention to this lesson. It explains:

* the basic format of every Pascal program
* the use of Reserved and Predefined words
* the procedure for compiling and running a Pascal program

OVERVIEW

The sample program shows how to print a message on the screen. Although this may not seem like much of an accomplishment, be patient. The program illustrates some very important points about the Pascal language.

Notice that the program is identified by the word **PROGRAM**, that it is given a name, **Ego**, and that the main part of the program is marked by the words **BEGIN** and **END**. When you enter the program, make certain that you copy it exactly as you see it. Like any computer language, Pascal is very precise and requires that you adhere to its rules exactly.

THE SAMPLE PROGRAM

The program in this lesson will simply print the message, "My name is *YourName*," on the screen.

1. First boot DOS 2.5 and the Kyan Pascal and select the Text Editor from the Main Menu. (Type "ED" after the prompt). When prompted for a filename, type **Dn:Ego.P** The screen will display the following message:

```
ED:  FILE NOT FOUND.
PRESS ANY KEY TO CONTINUE.
```

Press any key, the screen will clear, and the blinking cursor will appear in the upper left-hand corner. You can now enter the program which will be called **Ego**. To make sure that this is the case, press <**ESC**>. Note: the filename at the top of the menu is **Dn:Ego.P**. Press <**ESC**> again to get back to the editor.

2. Enter the program listed below, paying special attention to the punctuation.

```
PROGRAM Ego(Input,Output);

BEGIN
  Writeln;
  Writeln;
  Writeln('My Name is your name')

END.
```

3. Press <**ESC**> to get the Text Editor Menu.

4. Press **X** to save the program and exit the Text Editor.

5. When the **%** prompt appears, type **PC** and press **RETURN**.

6. When the prompt "**pc:**" appears, enter **Dn:Ego.P** followed by a hyphen P (-P). The compiler will translate your source code, i.e. the program, into code that the computer uses to run. The compiler saves the object code under the filename you have indicated, but it deletes the **.P** extension to the filename.

7. If no errors are reported, the prompt will reappear.

   * If there were error messages, note them, and return to the editor (you can return to the editor by typing **ED** when you see the prompt) Enter **Dn:Ego.P** to call your file. Make the corrections, save, and recompile your program, (i.e., repeat steps 3 through 6).

8. To run the program, enter **Dn:Ego** when the **%** prompt appears. Remember to delete the **.P** extension.

THE LOGIC OF THE PROGRAM

1. The first statement declares the name of the program, **Ego**, and tells the computer to expect input from the keyboard and to produce output on the screen.

   *In this program there isn't any input. However, a good programming technique is to always includes the Input-Output declarations in the name of the program.

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

3. The two **Writeln** statements are Pascal words for "write this line exactly as indicated." These two statements produce two blank lines on the screen.

4. The third **Writeln** statement is followed by a parenthesis, a single quote, and the message to be printed, which is followed by another single quote and a closing parenthesis. This statement prints the line on the screen.

5. The **END** statement, followed by a period, tells the computer that the program is completed.

TOPICS FOR PROGRAMMERS

Program Format

Every Pascal program begins by declaring the name of the program and by indicating, inside parentheses, that there will be input and output. This declaration is ended with a semicolon (;). Any Pascal program assumes that input will come from the keyboard and that output will go to the screen. Even though this program does not get any input from the keyboard, you should form the habit of telling the computer to expect input and to produce output.

The **BEGIN** and **END** statements open and close the body of the program. This part of the program tells the computer what it should do with all the information that it is processing. Put a period after the **END** statement to indicate that the program is finished.

Within the body of the program, use semicolons to indicate the end of separate commands. The line preceding an **END** statement, however, requires no punctuation.

Think of the body of the program as a single, very general statement; it **BEGIN**s and, after executing a series of commands, **END**s with a period. Individual commands within the general statement are separated by semicolons.

Indentations show the different parts of the program. The Pascal compiler (which translates the program into data that the computer understands) ignores blank spaces. The spaces, however, help you and anyone using your program to see the structure of the program. Indenting the lines between the **BEGIN** and **END** statements clearly shows that these commands are part of the body of the program.

If your are going to be a serious programmer, it is very important to write neatly formatted programs which clearly show the logic of your program.

Reserved and Predefined Words

The Pascal compiler, which translates your Pascal program into data that the computer understands, immediately recognizes a small list of words. These are called **Reserved** words. You can use these words only in situations that make sense to the compiler. **PROGRAM, BEGIN,** and **END** are examples

of "Reserved" words. In general, Reserved words indicate commands.

PROGRAM tells the compiler that what follows is an entire program, and not a Procedure or a Function. [We'll discuss these later.]

BEGIN and **END** tell the compiler that the main body of the program begins and ends at these points.

In addition to Reserved Words, Pascal uses **Predefined** Words. **Predefined** words indicate different types of data or identify functions that have already been defined by the program.

Other predefined words tell the compiler to expect whole numbers, characters, or operations that it should perform.

NOTE: Do **not** use any of Pascal's reserved words for the name of anything within the program.

In the manual, RESERVED words are printed in CAPITAL letters; all Predefined words are printed with only an initial Capital letter.

Declaring a Program

Every Pascal program has two parts: the **declaration** and the **program body**.

The first line of a program tells the compiler that what follows is a program. This line is the **Program Declaration.** Indicate that this is a program by writing the RESERVED word **PROGRAM** and the name of the program.

Then, inside parentheses, write the words **Input,Output**. This tells the compiler to expect input from the keyboard and to direct output to the screen. (Later, you will learn how to include filenames and other important information in the **Program Declaration**.)

The rest of this program consists of the **body**, which is a list of commands for the program to execute.

BEGIN tells the compiler that any following statements should be executed.

END tells the computer that the program is over. When it marks the end of the program, it is followed by a period. If the END statement indicates the conclusion of a block within the program, i.e. a Procedure, a Function, or a separate block of commands, it is followed by a semicolon.

ADVANCED TOPICS

Literals

A **Literal** is any printed character or characters which appears between single quotes. A series of characters contained within single quotes is called a String.

In the sample program, the **Literal**, "My name is *YourName*", is printed on the screen exactly as it appears within the single quotes. Such **Literals** always follow a **Writeln** statement.

Comments

Comments may be placed anywhere in a program; and, the more comments there are, the easier the program is to read. Indicate the beginning of any comment with a parenthesis and an asterisk. Indicate the closing of any comment with an asterisk and a parenthesis. The following is a sample comment

```
(* This is a comment *)
```

CONCLUSION

This sample lesson taught you how to:

* enter and organize a Pascal program
* use RESERVED and Predefined words
* get input and direct output
* compile and run a Pascal program

The next chapter introduces different types of data and demonstrates how to write a program that uses a formula.

This page left blank for your notes.

2. ENTERING FORMULAS

This program demonstrates how to write and enter formulas. It explains:

* Identifiers: **CONST**ants and **VAR**iables
* Read and Write Commands: **Read, Write, ReadIn, WriteIn**
* Directing output to the printer

OVERVIEW

This sample program calculates the cost of constructing a building based upon the cost of the materials, which is fixed, and the cost of the labor, which depends upon the number of man-hours worked and the rate of pay.

Since the total cost depends upon a number of factors, you will enter a formula that computes the answer for different values. The formula consists of a fixed value or constant, the cost of materials, and two variables, the hours worked and the rate of pay.

Notice that in addition to the Program Declaration and the Program Body discussed in the first sample lesson, this second lesson uses two new terms: **CONST** and **VAR**. Also notice that you are not just printing information to the screen; you are going to ask the user of your program to enter information at the keyboard.

THE PROGRAM

After you have entered the Kyan Pascal Editor, enter **D2:CONSTRUC.P** press any key at the **File Not Found** message, and type in the following program. Remember that the indentations are useful to you, the programmer, not the Pascal Compiler. The indentations highlight the separate parts of your program and show the logic of relationships. As in the first lesson, make certain that you follow the punctuation exactly.

If you have any question about which disk to use when you have a single or a double disk drive, or if you are not certain how to enter, save, compile, and run the program, refer back to lesson 1.

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

CONST
  Material = 325.0;

VAR
  Hours, Rate, Labor, Total : Real;

BEGIN
  Writeln ( 'Enter hours worked, and press RETURN.');
  Writeln ( 'Then enter rate of pay and press RETURN.');
  Readln(Hours);
  Readln(Rate);
  Labor := Hours * Rate;
  Total := Labor + Material;
  Writeln ('Labor = $', Labor: 8:2, ' Total = $', Total : 8:2)
END.
```

Once you have entered the program, press <**ESC**> to get the Editor Menu. Check the device:filename to make sure that it is named **D2:CONSTRUC.P.** If the name is different, use the **F** command and rename the file. Then press **X** to save it to disk. When you see the prompt "**%**", type Menu and <Return>. Select the compiler option "PC" and compile the program.

If the compiler detects any errors, it will print a listing on the screen. Return to the editor; correct the errors; and recompile the program. If the compiler does not detect errors, the compiled program will be saved to the disk and the prompt will reappear. Your program is now ready to be run. Remember that the compiled version of your program does not have the ".P" appended to the filename.

THE LOGIC OF THE PROGRAM

1. Request the number of hours worked and the rate of pay; then read the user's input.

2. Multiply the Hours times the Rate to calculate the cost of the Labor.

3. Add the cost of the Material to the cost of Labor to determine the Total cost.

4. Print the Labor and Total cost on the screen in a readable format.

GENERAL COMMENTS

When you program in Pascal, the first line identifies the name of the program and tells, in parentheses, if there will be input and/or output. The user provides the input from the keyboard. The computer directs output to the screen.

If the computer requires values that are fixed, these values must be identified before the main body of the program begins to execute. You can't ask the computer to do something with a value it doesn't know. Similarly, if you are going to have the user enter values on the keyboard while the program is running, you must tell the computer at least the names of those values and the type or types of numbers they will be. With this information, the program knows that it should save some space in memory for these user-entered values. Only then will the main program be ready to deal with the numbers it receives.

In Pascal, all of these types of information are labeled by **Identifiers**.

This lesson introduces two new types of **Identifiers:** CONST and **VAR.**

CONST tells the computer that whenever the identifier **Material** appears in the program, the value declared under the heading CONST will be used. In the sample program, the value 325.0 is the constant value assigned to **Material.**

VAR tells the computer that the identifiers listed under this heading will be assigned values at some other time during the execution of the program. At this point, the computer doesn't care what those values will be. It just needs to know the type of value. [You'll learn more about DATA TYPES later.] In this case, they are what Pascal calls REAL. This means that they have a decimal point in them.

The **BODY** of the program starts with the **BEGIN** statement. The next two lines print messages on the screen which tell the user how to enter the necessary information. The next line reads the values entered on the keyboard. Once the program knows these values, it calculates the Labor and Total costs of the project. The final line prints the results on the screen in a readable format.

TOPICS FOR PROGRAMMERS

Identifiers

In a Pascal program, an **IDENTIFIER** is a name. It may be the name of a program, of part of a program, or of a value, which can be constant or variable. Pascal requires only that you tell it the names of the identifiers and the types of data they will represent before you try to use them in a program. When you name an identifier, you tell the computer to reserve space in its memory to store a value. If you don't include the identifier and the computer encounters an unknown value, it doesn't know what that value is supposed to represent or where to store it. Pascal uses several different types of identifiers. At this point, we are concerned with only two of them: **constants** and **variables.**

Naming Identifiers

An **IDENTIFIER** can have almost any name you want to give it. Only 2 rules govern your choice:

1. The name must start with a letter.

2. Any combination of letters or numbers may follow including underscore characters). (**WARNING:** The compiler sees no difference between upper and lower case characters.)

To make a name unique, make sure that the identifiers within the same program are different. Your programs will be easier to read and rewrite if the identifiers clearly indicate what they stand for. For instance, "cost" is a better identifier than "C" even though both are equally acceptable to the computer.

Constants

Use constants to identify values that you will use frequently in your program. This practice makes your program easier to understand. If you come back to it in a few months, chances are that you may not remember what the number 325.0 stood for. But if the value is identified as "Material," you won't have to puzzle over it. Also, if the cost of material changes, you have to make only one change in the program. In a long program where a constant is used frequently, you would have to make many alterations to change one constant.

A constant is first identified by the word **CONST**; a semicolon (;) ends the declaration of the constant's value. Once you declare a **CONST**, you can use its identifier in any formula. You can not, however, try to change its value. Consequently, the identifier of a constant can only appear on the right side of an assignment statement. For example, you could use the constant **Material** in the following statement:

```
Total := Material + Labor;
```

But you could not use the constant **Material** to store a calculated value as in the following statement:

```
Material := Parts + Repairs
```

Since **Material** is defined as a constant, it **must** remain the same throughout the program.

Variables

Variables will be discussed throughout this manual. For now, it is important to know that they represent values that will be passed to the main body of the program at a later point. In this sense, they are the opposite of Constants which always remain the same. This value might be entered by the user while the program is running, or it may be a value that another part of the program will calculate before passing it to the main program.

When the compiler prepares the program for execution, it must expect and reserve memory space for these, as yet, unspecified values. The label VAR tells the program that what follows is a list of the identifiers, or names, of the variables. All you have to do is indicate that you are listing the variables: Tell the computer the names to expect, and define the type of data that each variable represents. In this example, we are concerned with Real numbers, i.e. numbers with a decimal point. Other types of data will be discussed in later examples.

NOTE: A semicolon (;) indicates the end of the variable list.

Input and Output

The Pascal program uses four commands to get information from the keyboard and output it to the screen.

* **Read**
* **Readln**
* **Write**
* **Writeln**

The two read commands ("Read" and "Readln") tell the computer to accept information from the keyboard. The two write commands ("Write" and "Writeln") print information to another device.

Read gets one element of data which has been labelled by an identifier.

Readln reads an entire line of input (e.g. the computer gets data until it senses that the <RETURN> key has been pressed).

Note: When a Readln statement is given in the program, more than one variable may be input. Simply separate the items by a space. In the sample program, the user enters one value, presses <RETURN>, and then enters the next value. It is also possible, because the program is executing a Readln statement, to enter both values, separated by a space, before pressing the <RETURN> key.

Write prints the quoted line, or a value represented by an identifier, on the screen. It does not, however, advance the cursor to the next line, but waits at the end of the printed line for the next read or write statement.

Writeln prints the quoted line, or a value represented by an identifier, on the screen. Unlike the Write command, the Writeln command advances the cursor to the next line on the screen.

Output to the Printer

After you name -- or declare -- a Pascal program, you must also tell the computer to accept input or print output. You do this by telling the program to acccept input and output files. This use of the term "file" may seem strange if you are new to the Pascal lanaguage. A file, in Pascal, indicates a device. The default device for input is the keyboard; for output, it is the monitor screen.

When the computer reads a statement like

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

it assumes that information entered at the keyboard goes to an input file and that information to be output is directed to the screen.

Often, however you may want to direct output to another device, either to a printer or to a disk drive.

Kyan Pascal allows you to redirect output to the printer by setting up a redirection file called "Printer". First, declare a variable "Printer" as a file of text. Then, in the body of the program, insert a Rewrite statement to initialize

the filename. Finally, use the following syntax to redirect the output:

Writeln (Printer, '... text ...');

The following program is the same as the sample program except that it directs the output to the printer instead of to the monitor.

```
PROGRAM Construction(Input, Output);

CONST
  Material = 325.0;

VAR

  Hours, Rate, Labor, Total : Real;
  Printer: Text;

BEGIN
  Rewrite(Printer, 'P:');
  Writeln ('Enter hours worked and press RETURN');
  Writeln ('Then enter rate of pay and press RETURN');
    Readln (Hours);
    Readln (Rate);
  Labor := Hours * Rate;
  Total := Labor + Material;
  Writeln(Printer, 'Labor =$', Labor:8:2, ' Total = $', Total:8:2);
END.
```

ADVANCED TOPICS

Formating Topics

When you want to control where the output is printed--either to the screen or the printer--you must tell the computer how to print it. In the original sample program, notice line 5. It tells the program to print **'Labor = $'** and is then followed by **Labor : 8:2.** The first statement prints text on the screen. The second statement tells the computer how to format the printed output of the value **Labor**. It indicates that whatever value **Labor** currently has should be allowed 8 decimal positions (including the decimal point) and 2 positions for the following decimal value.

CONCLUSION

In this lesson, you have learned how to:

* Construct a formula
* Declare constants and variables
* Get input and write output
* Format data on the screen.

In the next lesson, you will learn how to manage and manipulate different types of data.

(This page left blank for your notes.)

3. DECISION MAKING

In this lesson, your will learn how to:

*Assign values to variables
*Use the **IF-THEN-ELSE** statement to make decisions

OVERVIEW

In the previous lesson, you learned how to construct a formula that calculates a value and then how to print that value to the screen. This lesson also gets information from the user and prints output to the screen. In addition, however, it makes a decision based upon the information supplied. This involves using an **IF-THEN-ELSE** statement.

This sample program calculates the amount of social security tax deducted from a paycheck. It first asks the user to enter three values: the hours worked, the rate of pay, and the amount of tax already paid. **IF** the tax on this payment plus the tax already paid is greater than the maximum tax which can be collected, the program calculates how much must be paid to reach the maximum tax. Otherwise (**ELSE**), the tax is computed and added to the amount of tax to date. Finally the results are printed on the screen.

Note: The **IF** statement tests for the existence of a certain condition. When that condition is true, it performs one group of actions which are listed under the **THEN** statement. When the condition is not true, it performs the sequence of actions which follow the **ELSE** statement.

THE PROGRAM

Beginning with this lesson, we assume that you know how to enter the Text Editor and how to write, compile, and run the program.

Remember that every program must be declared, that you must define the variables, and that the BODY of the program is enclosed between BEGIN and END statements. Also remember that punctuation must be followed exactly and that indentation allows you to indicate the logical parts of the program.

```
PROGRAM SocialSecurity  (Input, Output);

CONST
  TaxRate = 0.075;
  TaxMaximum = 4275.0;

VAR     (*  These values will be entered by the user  *)

  Hours, Rate, TaxNow, TaxToDate :  Real;

BEGIN               (*  The BODY of the program*)
  (* Get hours, rate, and tax-to-date values *)
  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 pay period  *)

  TaxNow :=  Hours * Rate * TaxRate;

  (* Determine if Tax paid-to-date + tax for this pay
     period is greater than the maximum tax allowable *)
```

```
  IF TaxToDate + TaxNow > TaxMaximum THEN
    BEGIN
    TaxNow := TaxMaximum - TaxToDate;
    TaxToDate := TaxMaximum
    END  (* of the IF-TRUE statement *)
  ELSE  (* if the IF statement is 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.
```

Remember to save and compile the program. Also remember to delete the .P extension from the filename when you want to run the program. If you need to re-edit the program, enter the Editor and use the filename.P to access the source code file.

THE LOGIC OF THE PROGRAM

1. Declare the program's name and that there will be input and output.

2. Define constants using the "=" sign.

3. Declare variables and specify their data type.

4. Write requests for information to the screen.

5. Read input from the keyboard.

6. Compute the tax due on the current paycheck.

7. If the tax for this period would make the total tax withheld greater than the maximum, subtract the tax to date from the maximum. This returns the amount of tax still due. This is the real amount to be deducted from this paycheck.

8. If the tax from this period would not make the total tax paid exceed the maximum, simply add the amount to the tax paid to date.

9. Write the results to the screen.

GENERAL COMMENTS

A few reminders from the previous lessons should help make this program easy to understand.

CONSTANTS must be declared, using an equal "=" sign, immediately before the **VAR**iables.

VARIABLES tell the computer to expect values that will be entered while the program is running. These variables may be input by the user; they may be read into the program from another file; or they may be temporary storage places where the program stores intermediate calculations. In any case, the computer will associate the value with the name you have defined. The Data Type of the variable must be indicated at the end of the variable list. In this case, they are all real numbers, i.e., numbers that may contain decimal points.

The **IF-THEN-ELSE** Statement examines the current status of the program; it then performs one action if that condition is true, and another if it is not true. For example, if 2 numbers are greater than a third number, one action is taken; if they are not greater than a third number, then another action is taken. The **ELSE** part of the IF conditional statement is optional. If the condition is true, the program executes the commands following the THEN statement. If the condition is not true, the program simply skips all statements associated with the THEN statement and goes to the next statement line of the program.

This decision-making ability is the basis for all machine intelligence decisions.

TOPICS FOR PROGRAMMERS

Assigning Values: The Assignment Statement

Pascal has two ways of assigning values and it is crucial that you understand the difference between them.

= is used to assign a value to a constant in the declaration section of the program.

```
TaxRate = 0.075
```

It is also used in conditional statements such as:

```
If TaxNow = TaxMaximum THEN
```

or

```
If X = Y THEN
```

Otherwise, it is only used to indicate the Identifier of a user-defined data type (You'll learn more about this later.).

:= is used to indicate that the value on the left of the symbol now equals the value or values on the right.

For example,

```
TaxToDate := TaxToDate + TaxNow
```

tells the computer to take the current value of TaxToDate, add it to the value of TaxNow, and then let TaxToDate represent the new value.

Note: Use "=" to define constants, logical relationships, and user-defined data types. Use ":=" in equations that assign values from the right side of the equation to the term on the left side.

Conditional Statements: IF-THEN-ELSE

Conditional statements take the following form:

> **IF** a condition is TRUE
> **THEN** perform these commands
> **ELSE** perform other commands

The sample program uses conditional statements to determine which of two actions it should take. In this program, there are two possible alternatives depending upon the total amount of tax paid after tax has been deducted from the paycheck. If it is more than the maximum, one set of actions is taken. If it is still less than the maximum, another action is taken.

Note: Ordinarily, if only one action is taken by each decision, a **BEGIN** and **END** statement is not needed. In the sample program, however, if the condition is true, 2 actions are taken -- the current tax is calculated and the tax to date is updated. Since a group of commands will be executed when the **IF** condition is true, they should all be listed within a **BEGIN-END** pair. Otherwise, only the first action in the group would be associated with the **IF** statements.

Also note that the statements are separated by semicolons except for the last statement before **END**. You never need punctuation before **END**. Since this **END** is just the end of the **IF-TRUE** part of the condition, it is not followed by a period. A period would indicate the end of the program.

When the **IF** condition is false, the program skips all the statements in the **THEN** section and turns control over to the **ELSE** statements. In the sample program, there is only one **ELSE** command executed, so the **BEGIN-END** pair is not required. In fact, this program doesn't really need the ELSE statement at all. If the condition is not met, the program skips the **THEN** statements, moves to the next command line, and calculates the Tax To Date value by the alternate method.

Operators: Arithmetic and Relational

Operators are symbols that are used to indicate relationships between numbers and other items in a Pascal program.

Arithmetic Operators

A Pascal program can perform the four basic arithmetic operations.

Add	+
Subtract	-
Multiply	*
Divide	/

When evaluating a mathematical expression, multiplication and division are performed before addition and subtraction. Thus,

6 + 8/2 = 10 (not 7)

Remember that in mathematical expressions, "=" is not an operation. Pascal uses := to assign the value of an expression to a symbol that represents that expression.

Relational Operators

Relational Operators are used to indicate logical relationships between items. They are primarily used in conditional statements to indicate which action the program should take. The 6 relational operators are:

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

Look again at the sample program. Arithmetic operators are used in the calculations that determine the values of TaxNow and TaxToDate. The Relational Operator > is used in the condition statement.

ADVANCED TOPICS

Nested Conditions

In advanced programming, you may find that several complex conditions determine which action the program should take. In that case, it is possible to nest **IF-THEN-ELSE** statements. Simply replace the **ELSE** statement with another **IF-THEN-ELSE** statement. The logic of this nested condition is illustrated below.

```
IF condition is true
  THEN do x
        (Otherwise perform another test by replacing the ELSE
        statement with another IF statement.)
  IF next condition is true
    THEN do y
    ELSE do z.
```

It is possible to create complex conditional branches using nested **IF-THEN-ELSE** statements, but be careful. It is very easy to have an ELSE statement inadvertently associated with the wrong IF statement.

CONCLUSION

In this lesson you have learned how to:

* Assign values to variables
* Write and use conditional statements

In the next lesson, you will learn more about the types of data that a Pascal program can manipulate.

4. INTEGERS AND FOR LOOPS

In the last lesson you learned how to assign values to variables and how to tell the program to make decisions.

In this lesson you will learn:

* how to use the data type **INTEGER**
* how to use the **FOR** loop
* how to use 3 predefined functions:
  TRUNC, ROUND , and **MAXINT**

OVERVIEW

This program calculates the average of a series of whole numbers (i.e., numbers without fractional or decimal parts) that a user inputs. It is similar to the previous lesson in that it accepts numbers from the keyboard and performs a calculation using those numbers.

It is different, however, in that it requests first the number of items to be averaged. It then requests the numbers to be averaged. The number of times the request is made depends upon the number of items to be averaged. Next, if a number is not a whole number, the program converts it to a whole number. Finally, the program computes the average of the group of numbers.

The program introduces three new concepts: **Integers** (or whole numbers), the **FOR..DO** loop, and the **ROUND** statement.

THE PROGRAM

This program declares a new type of Variable, the Integer.
It also uses a FOR..DO loop that repeats a series of statements a given number of times. The ROUND function corrects the user's entry if it is not a whole number.

```
PROGRAM Average (Input,Output);

VAR
  X,Y :  Real;
  Number, Count :  Integer;

BEGIN
  Writeln;
  Writeln;
  Writeln('Enter the number of ');
  Writeln('items to be averaged.');
  Readln(Number);
  FOR Count  :=  1 TO Number DO
   BEGIN        (* Begin FOR loop *)
     X := 0;
     Writeln('Enter a whole number.');
     Readln (Y);
     Y := Round (Y);
     X := X + Y
   END;         (* End of FOR loop *)

  X := X/Number;          (*  Calculate the average  *)
  Writeln('The Average is  ', X : 5:2)

END.
```

Remember to save and compile the program. To run it, delete the .P extension to the filename.

THE LOGIC OF THE PROGRAM

1. Declare the program.

2. Define the Variables.

   * List the variable names and, after a colon (:) indicate the type of data, Real or Integer, that they will represent.

3. Request the number of items and read the input.

4. Use the number of items as a control for how many requests will be made for input.

5. If the user's entry is not an integer, round off the entry.

6. Calculate the average.

7. Print the average to the screen using formatted decimal positions.

GENERAL COMMENTS

This program illustrates several important programming concepts. Pascal treats integers very differently than it treats real numbers. Your program should always determine which type of data it is dealing with. The **FOR..DO** loop shows how to make the computer repeat a sequence of actions a certain number of times. Finally, the **Round** function corrects any input mistakes the user might make. A good program tries to anticipate and correct erroneous entries.

TOPICS FOR PROGRAMMERS

Data Types: Real Numbers and Integers

Pascal understands a number of data types. So far, you have used two of them -- Real numbers and Integer numbers.

Real Numbers

Variables that are Real numbers must be declared as such in the Variables List. The form is:

```
VAR
  X, Y, Z : Real;
```

X, Y, and Z may be any legal name.

In Lesson 2 you learned about Real numbers. Recall that they are numbers that contain fractional or decimal values. Integers, on the other hand, are whole numbers. Kyan Pascal requires that you distinguish between these two data types very carefully.

Real numbers may be positive or negative and are represented in either decimal or scientific notation, i.e., 12.8, -15.7, 3.456E+11, or -2.5555E+4. [If you are unfamiliar with the last two versions, they are simply scientific notation. It specifies how many decimal places should follow or precede the number.]

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 following decimal and scientific notations represent the same value:

```
Z := 345.55
Z := 3.455E+2
Z := 34555E-2
```

The + or - 2 after the letter E just indicates how many decimal places the point should be moved and what direction to move it in.

The range of values that may be assigned to a Real number is from ±9.9999999999E-99 to ±9.9999999999E+99.

Integers

Integers are whole numbers. Pascal can use any Integer between the range of -32768 and +32767. Integer variables are declared in the Variable List. The format of the declaration is identical to that for Real Numbers:

```
VAR
  Number, Count : Integer;
```

In the sample program, both types of variables are listed under the term VAR and are separated by a semicolon. A semicolon also indicates the end of the Variable List.

If an arithmetic expression is the result of combining Real and Integer values, it is considered to be a Real number.

When you want to print numbers, either Real or Integer, to the output device, remember that you must indicate the number of integer and decimal positions that the value will have. You do this by following the variable with a colon, the total number of digits that you want printed, another colon, and the number of decimal positions.

> * See Lesson 2 if you don't remember how to do this. For example, in this lesson, the program prints numbers up to 5 digits, and reserves 2 decimal places for fractions.

If the number has fewer digits than the number of spaces reserved for it, the correct number will appear on the screen, but the program will fill in the extra spaces with blanks or zeros. If a number in decimal format has more digits than the number of spaces reserved for it, a run-time error will occur. That is, you will not learn about the error until the program is actually run. Run-time errors also occur when an Integer is greater than 32767 or less than -32768.

A Real, or decimal, number is also limited to 13 significant digits. Writing a number that requires more than 13 places will not make the number more accurate. The computer will present the correct number, but it will fill the digits beyond the 13th place with zeros or blanks. On the other hand, truly accurate calculations will not be produced if the program does not take advantage of the full 13 digit capability. Kyan Pascal is unique in that it can handle figures of this size, so take advantage of this capability. Most Pascal compilers support less than 8 digits.

FOR..DO Loops

The **FOR..DO** loop is a control statement that causes the program to execute a series of commands the number of times indicated by the loop. Its format is:

```
FOR count.name := count.beginning TO count.end DO
  BEGIN
   statements
  END;
```

Note: A semicolon (;) is used to end the **FOR** loop. Also remember that all statements between the **BEGIN-END** pair are separated from each other by a semicolon (except the last statement before the **END**).

This command accepts an initial value and repeats the indicated commands, increasing the initial value by 1 each time the sequence is executed. When it completes the number of repetitions indicated by the control value, it moves on to the statement following the **END** command.

Integers are most commonly used in **FOR** loops, although it is possible to use Real number variables.

The **FOR** loop can also decrement the loop control variable if you use **DOWNTO** instead of **TO**. The following example is a valid definition of a control loop:

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

In this example, **Number** must be greater than or equal to 1. Otherwise, the FOR loop can not count down to the control number.

ADVANCED TOPICS

Predefined Functions

Kyan Pascal has a number of functions that you may use in computing values. Three of them are useful in manipulating Real numbers:

TRUNC	Truncate
ROUND	Round
MAXINT	Maximum Integer

The sample program used the **ROUND** function to round the value entered by the user to the nearest Integer. This prevented the program from crashing if the user entered a number with a fractional or decimal part.

The 3 functions are defined below:

TRUNC Truncate eliminates any decimal value after the decimal point.

Trunc(9.6) becomes 9
Trunc(9.1) becomes 9

ROUND Round returns the value that is closest to the decimal number.

Round(9.6) becomes 10
Round(9.1) becomes 9

MAXINT Maximum Integer is a Pascal constant that equals the largest integer that your computer can handle. In Kyan Pascal, this integer equals 32767.

CONCLUSION

In this lesson you have learned how to:

* work with **Integers**
* use **FOR..DO** loops
* use predefined function words

In the next lesson, you will learn how to manipulate text.

5 STRINGS AND ARRAYS

This lesson introduces a new type of data -- the **Character**. It shows how to use this data type to create strings of text. You will learn how to:

* Declare and use the **Char** data type.
* Use the Reserved Word **ARRAY** to declare a new data type called **STRING.**
* Create **WHILE** loops that repeats a series of actions.
* Control the format of output.

OVERVIEW

Beginning with this lesson the programs become more complex, and they will often require you to use programming techniques introduced in previous lessons.

The sample program asks the user to enter words. It then prints a message telling the user how many words were entered and which is alphabetically first. Since the size of the list is not known in advance, the program expects a signal, which in this program is a plus sign (+) to indicate the end of the list.

The program also shows how to format the output to the screen in a pleasing display.

THE SAMPLE PROGRAM

This program uses the **WHILE** loop to determine whether the user wants to end the list. It also uses an **IF** loop to repeat the request until the **WHILE** condition is no longer satisfied. Finally, it introduces a new type of data, the **ARRAY**, which is used to define a String of characters.

```
PROGRAM FirstWord (Input,Output);
        (*  This program requests a list of words, selects the alphabetically
        first word, and counts the number of words entered.  *)
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;  (*  of the WHILE loop  *)
Writeln;
Writeln;
Writeln(LoopCount: 5, ' words were entered.');
Writeln;
Writeln(LeastWord:25);
Writeln;
Writeln('    is alphabetically first.')
END.
```

After compiling the program, remember to delete the .P filename extension to run it.

THE LOGIC OF THE PROGRAM

1. Declare the program's name.
2. Assign the "+" to the variable **Signal.**
3. Define **String** as an ARRAY of Characters.
4. Declare the Variables and their data types.
5. Request and read the first word to be entered.
6. Initialize the variables:
   LeastWord := Word
   LoopCount := 0
7. Begin the **WHILE** loop which continues to operate until a "+" is entered.
   a) Compare the entered word to the previous word. If it is alphabetically first, save it as LeastWord.
   b) Increment the LoopCount
   c) Repeat the request message.
   d) Read the next word.
8. Print the formatted output to the screen.

GENERAL COMMENTS

This program introduces a new data type that Pascal understands -- **Char**. Using **Characters**, it defines another type of data -- the **String**. **Strings** allow the program to manipulate text and make comparisons between entries. After asking the user to enter a word, i.e. a **String**, it compares that string to the previous string.

The computer can compare **Strings** because every letter has a numeric value that represents it to the computer. The letter "a" has the lowest numeric value and the letter "z" has the highest numeric value. The program simply tells the computer to check the value of the first letter of each word. If it is less than the previous value, this word becomes the first word. If the letters are the same, the program will check the next letter. This checking continues until the computer detects a difference and defines the least value as the first word.

The two control loops, **WHILE** and **IF**, are the brains of the program. The **WHILE** loop will continue to request words as long as a + has not been encountered. The **IF** loop compares the numeric values of the current and the previous entry and determines which is first.

The empty **Writeln** statements and reserved character positions format the output to the screen.

Note: The **[1]** after **Word** in the **WHILE** loop condition indicates that the first character of the entry should be checked. Remember that **Word** is an **ARRAY**. You must tell the program which element of the **ARRAY** to check. This will be explained below.

TOPICS FOR PROGRAMMERS

DATA TYPES: Char, ARRAY, and String

Until this lesson, you have used only two types of data that a Pascal program understands -- Integers and Real numbers. You also want to be able to work with text. The **Char** data type allows your program to manipulate this new type of information.

Char

Char, like Int and Real, is a predefined data type. It tells the computer to expect a single character. A **Char** variable can be any printable letter, number, or symbol; it can also be a space or a **<RETURN>**. A number that is represented as a character, however, cannot be used in arithmetic operations.

A **Char** data type is declared in the same way as an Integer or a Real number. If your program expects the user to enter a letter to select an item from a menu, you might declare a variable, Selection, that will represent that choice. In the declaration part of the program you would enter:

```
VAR
  Selection :  Char;
```

This tells the program to expect a variable called "Selection" and that the variable will be a character.

Arrays

An **ARRAY** is a collection of similar types of data. The word **ARRAY** in a program tells the computer to expect a specific number of items that it will group together. It is like a list. It could be a list of the days of the week or the months of the year. It could also be a list of letters, integers, or real numbers..

When you declare an **ARRAY**, you must tell the computer how many elements will be in it. The computer then associates each element in the array with its position. The format for declaring an array is:

```
ARRAY [I..X] OF data type;
```

The numbers in the brackets indicate the first and last elements in the array, and an ellipsis (..) separates them. This ellipsis means that all numbers between the first and last are to be included in the array. Remember that the number indicates the position of the data element within the array.

The words "Pascal is fun" comprise an array of 13 elements. Element [1] is "P", and element [9] is "s".

When you declare an **ARRAY**, you must also indicate what type of elements it will consist of. The sample program defines an **ARRAY** of characters, but

you can define an array of any data type. You can even declare an **ARRAY** of an **ARRAY**.

Strings

A String is a sequence of characters. In Kyan Pascal, strings must be defined as **ARRAY**s of characters. The sample program defines **String** as an **ARRAY** of 15 characters. Note that since a String is a user-defined data type, it must be declared under the heading **TYPE**. This signals that you are defining a unique type of data. You must define the TYPE String before you can identify a variable as a String TYPE.

Once you define a String as an array of a number of characters, the String will always be of that length. If you use less than the maximum number of characters, the gap will be filled with spaces. If you try to use more characters, they will be lost.

Re-examine the sample program to see how it defines the data type String as an **ARRAY** of 15 characters. It then declares 2 variables, Word and LeastWord, to be Strings. When you run the program and enter a word with less than 15 characters, the program works fine. The missing letters are filled with spaces. If you try to enter a word with more than 15 letters, the extra characters will not appear.

If you want to assign String or Character values in the body of a program, they must be enclosed in single quotes. The following section of a program assigns the word "Help" to a String and the letter "A" to a Char.

```
TYPE
  String = ARRAY [1..10] OF Char;

VAR
  Word : String;
  Letter : Char;
```

```
BEGIN
  Word := 'Help      ';
  Letter := 'A';
    .
    .
    .
```

Note that the String defined in the body of the program must contain the exact number of items that were indicated in the declaration. In this case, the string contains 4 letters and 6 spaces to fill the 10 elements in the **ARRAY** of characters.

More About Reading and Writing

The program in this lesson showed two ways to print messages to the screen -- **Write** and **Writeln. Write** prints the text inside the quotes and leaves the cursor positioned immediately after the text. **Writeln** prints the text, but also moves the cursor to the next line; in other words, it forces a carriage return. If you want to have the user enter information on the same line as the text, use **Write**; otherwise, use the **Writeln** command.

Read and **Readln** get information from the keyboard. **Read**, however, only gets 1 piece of data, while **Readln** gets all the data entered until **RETURN** is pressed. Later in this tutorial, you will learn the most common use of the **Read** command. For the present, it is best to use only the **Readln** statement.

Matching The Declared Data Type To The Input

In your programs, you must always declare the type of data that will be entered for each variable. You should also insure that the user enters data of the appropriate type. If the program expects the user to enter an Integer and he enters a letter, it won't know how to treat the data.

This point is so important, it bears repeating.

> Note: When a program reads a variable, the data entered at the keyboard must be the same type of data that was identified in the declaration.

For example, your program contains the line **Readln(Number)** and the variable, **Number**, was defined as an Integer. If the user enters **Tom**, the

computer doesn't know what to do with the data. Similarly, if you defined **Number** as a String, a **Read(Number)** would only get the **T** and then it would not know how to treat the letter as a number. A **Readln** command would return the entire String, but it still would not know what to do with an Integer, **TOM**.

To illustrate the correspondence between data type a variable is declared to be and the data type of the actual entry. Suppose that the user enters "123 Ralph" in response to a request for data. The following table shows what the computer reads, depending upon how the variable is declared and whether the **Read** or **Readln** command is used.

The following table illustrates what happens when the declared data type doesn't precisely match the type of data the user enters.

The user enters: **123 Ralph**

Command	VAR Type	What the Program reads
Read	Int	1 (the number)
	Real	1.0 (the number)
	Char	1 (the character)
	ARRAY[1..9] OF Char	1 (the character)
Readln	Int	123 (the number)
	Real	123.0 (the number)
	Char	1 (the character)
	ARRAY[1..9] OF Char	123 Ralph (the characters)

To read the entry "123 Ralph" as a number and a literal string, use 2 variables. Define the first as an Integer or Real number and the second as an **ARRAY**

OF Char. **Readln(X1)** will get the number "123" and **Readln(X2)** will get the string "Ralph". You may also put the two variables within a single **Readln** statement, as in **Readln(X1,X2)**.

While..Do Loops

The **WHILE** loop is similar to the **IF-THEN-ELSE** and the **FOR..DO** loops in that it forces the program to execute a command or a sequence of commands. Its format is also similar:

```
WHILE conditional statement DO
  BEGIN
    Command 1;
    Command 2;
    Command 3
  END;
```

The **WHILE** loop defines a condition, and as long as that condition is true, it continues to execute. The **FOR** loop predetermines the exact number of times a routine will execute. The **IF** loop can repeat a routine or begin another one. **WHILE** loops continue to execute until the validity of the control statement is FALSE. This test is made before the WHILE executes. As soon as the test proves FALSE, the program "falls through" the loop to the next statement after the end of the loop. You will learn more about Validity statements in the next lesson.

There is one caution that should be strictly adhered to when you use a **WHILE** loop:

> **Never write a loop that can not exit its sequence of commands.**

The program, at some time, must achieve the condition that will force it to exit the **WHILE** loop.

ADVANCED TECHNIQUES

Formatting Output

Advanced programmers use a number of techniques to make the output to the screen more readable. They may clear the screen before printing text and they force the output to align on the monitor in a easy-to-read format.

Reserved Places

In the sample program the output was carefully directed so that it is easy to read. You should use empty **Writeln** commands to force blank lines between output. You can also use the **:number** option to allocate a number of spaces for any kind of output. The actual text or value will be printed at the leftmost extreme of the spaces you reserve for it. In the sample program, 5 spaces are allocated for the number of items entered. Consequently, the number, if it is less than 10, will be indented five spaces. Then a blank line is printed, followed by the LeastWord, which is given 25 spaces to insure that it will be indented on the line. After another blank line, the statement "is alphabetically first" contains enough spaces to make it centered under the LeastWord.

Chr

Every ASCII character corresponds to an Integer from 1 to 128. The function **Chr** returns the ASCII character that corresponds to the integer indicated in the parentheses. This is what allows the computer to determine which letter comes before another.

Some characters don't print to the screen but control how data is printed to the screen. The statement:

```
Writeln(Chr(125));
```

will clear the screen. If you begin your program with this statement, your output to the screen will be much more readable.

CONCLUSION

By now you are more than a beginner programmer. In this lesson you have learned how to:

* Use Characters, Arrays, and Strings
* Write or read data
* Format output to the screen

The next lesson explains how to make logical decisions using a new data type: **Boolean.**

(This page left blank for your notes.)

6 BOOLEAN VARIABLES

This lesson introduces the data type -- **Boolean.** Boolean variables are either TRUE or FALSE. In this lesson, you will learn how to use:

* Boolean Variables and Equations
* Boolean Operators
* The operators **DIV** and **MOD**
* The order of operations used in evaluating expressions

OVERVIEW

The computer makes decisions based on the logical state of a defined condition. This logical state is either TRUE or FALSE. Boolean logic is used to determine the TRUE or FALSE state of complex conditions.

The program in this lesson tests your skill at division. It asks you to enter a number, a factor of that number, and finally, the other factor. It then determines whether the second two numbers are correct. Correct answers are determined by a Boolean equation. If they are correct, the user is congratulated and asked if he wants to try again. If the answers are wrong, the user is asked if he wants to make another attempt.

The program uses two new operators, **DIV** and **MOD**, as part of a Boolean equation, to determine whether the answers are correct or not. **DIV** returns the quotient when two numbers are divided. **MOD** returns the remainder.

THE SAMPLE PROGRAM

The following division lesson uses a large **WHILE** loop which continues to execute the program as long as the answer to "try again" is Yes. A nested **IF** loop controls the output, which depends upon whether the answer is right or wrong. The test for the correct answer is a Boolean variable, **Correct.**

```
PROGRAM DivLesn (Input,Output);
        (*  This program requests information from the user and tests
        whether the second two numbers are factors of the first  *)

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 its 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   (*  of the THEN clause  *)
    ELSE
        BEGIN
          Write('Wrong! . Try again? Enter Y,N');
          Readln(Ans)
        END     (*  of the ELSE clause  *)
   END          (*  of the WHILE statement  *)
END.            (*  of the program  *)
```

THE LOGIC OF THE PROGRAM

1. Name the program and declare the 3 types of variables -- Integer, Char, and Boolean.

2. Initialize the "Answer" to "Yes."

3. Begin the WHILE loop which continues to execute as long as the "Answer" is the letter Y.

4. Write the requests for data and read the information from the keyboard.

5. Define the conditions of a Correct answer, i.e., the first number can be divided by the second with no remainder.

6. Begin the IF loop. If the answer is correct, print the results. If it was not correct, print "wrong".

7. Ask if the user wishes to continue. Stop when the user enters N.

8. Close the BEGIN statements with corresponding END statements.

GENERAL COMMENTS

This program introduces the new data type, **Boolean.** It also uses two new functions, **DIV** and **MOD.** These items will be explained in the "Topics for Programmers" section of the lesson.

For now, you only need to understand that when you define "Correct" to be a Boolean data type, you mean that "Correct" can only equal TRUE or FALSE.

In the program, you define the conditions that will make "Correct" TRUE. "Correct" will be true if there is no remainder after the numbers are divided (i.e., X MOD W= 0) and the 2 numbers the user enters are really factors of the first number (i.e., X DIV W=Z).

Also note the punctuation of the nested loops. A BEGIN statement requires no punctuation. Semicolons separate elements within the BEGIN statement

but none is required before the END statement. The END statement does not require punctuation until the end of the program.

TOPICS FOR PROGRAMMERS

DATA TYPES: BOOLEAN

A **Boolean** data type always equals either TRUE or FALSE. To use a Boolean variable you must first declare the variable identifier as Boolean; then, in the program body, define the conditions that make it true. The sample program defines "Correct" as a Boolean variable. In the program body it states that a division without remainder and a division yielding a specific number are both required to make a TRUE condition. Once the condition has been defined, the sample program uses an **IF** test which executes one set of instructions if the Boolean variable is True and another set if it is False.

Boolean Operators

Boolean expressions use three operators to define conditions:

NOT
OR
AND

These Operators follow the rules of formal logic.

NOT indicates that the opposite of the condition is true.

NOT True = False
NOT False = True

For example, a Boolean variable that is NOT the True condition is False. One that is NOT the False condition is True.

OR indicates that if either element in a pair of conditions is True, the result is True. Otherwise, it is false. Consequently,

True OR False = True
False OR True = True
True OR True = True
False OR False = False

For example, two cars are racing. The race is over (True) when car A OR car B crosses the finish line. Only one of the conditions needs to be True for the result to be True.

AND indicates that both conditions must be True for the result to be True.

True AND True = True
True AND False = False
False AND True = False
False AND False = False

For example, the environment is clean (True) only when the air AND the water are clean. If both conditions are not met, the environment is NOT clean.

Use Boolean variables to define logical conditions. Set the condition to TRUE or FALSE. Then use one of the loop statements to continuously execute statements until the condition is in the logical state you set. Be sure to include a test of conditions within the loop that will change the state of the logical condition when the loop should end.

DIV and MOD Operators

DIV and MOD are two predefined functions that can be used in Pascal programs. DIV returns the quotient of one number divided by another. Its syntax is

A DIV B

MOD returns the remainder of a division. Its syntax is:

A MOD B

For example, if A = 14 and B = 4, then A DIV B = 3 and A MOD B = 2.

ADVANCED TOPICS

PRECEDENCE OF OPERATORS

You can construct complex equations or logical conditions using the Arithmetic and Boolean operators.

Remember, however, that the computer follows strict rules that govern how it evaluates expressions. Operations of greater precedence are executed before operations of lesser precedence.

The five levels of precedence are:

1st--Highest Precedence	( )
2nd level	NOT
3rd level	*, /, AND, DIV, MOD
4th level	+, -, OR
5th--Lowest Precedence	=, <=, >=, >, <, <>

Because parentheses have the highest level of precedence, you can use them to direct the order of operations within an expression. Any part of the expression in parentheses will be evaluated first. For example, 4*(5+1) = 24; but (4*5)+1 = 21. If parentheses are nested, the innermost pair will be evaluated first, e.g., 3*(2+(6/2)) = 15.

When you nest parentheses, always count to make sure that there are an equal number of left "(" and right ")" symbols. If they are not equal, the expression will not be evaluated correctly.

CONCLUSION

This lesson has introduced you to the concept of Boolean data types. Since this is only an introduction, you may want to consult other books for a complete explanation of Boolean logic and how to use it in programs.

The next, and last, lesson in Part 1 of the Tutorial explains a number of important principles. When you finish it, you should be ready for the more complicated (and more powerful) uses of Pascal.

7. SCALAR VARIABLES

This lesson introduces a number of new data types, statements, and functions. It explains:

* The data type: **Scalar**
* The data type: **Subrange**
* The **REPEAT..UNTIL** statement
* The **CASE..OF** statement
* The functions: **Ord, Succ,** and **Pred**

OVERVIEW

Sometimes you may want a variable to represent a short list of items. The variable "Days", for example, may represent the items: Sunday, Monday, Tuesday, etc. You can do this by defining the variable as a **Scalar** variable. Unlike the other variable declarations, Scalar variables are not named **Scalar**; the variable identifier is simply followed by a list of items, enclosed in parentheses, that it can represent. It is defined under a **TYPE** heading to indicate that it is a user-defined **TYPE** of data. When the program runs, the variable can contain any of the elements that were assigned to it in the declaration.

The sample program declares a **Scalar** type that contains the words "Yes" and "No," i.e., the variable can represent either of those two values. It will continue to run only as long as the Scalar variable equals "Yes".

The program asks the user to enter a hexadecimal number and then converts it to a decimal equivalent. To do this, it uses a **REPEAT..UNTIL** statement that continues to execute until a condition is met. Within the **REPEAT** are two **IF** conditional tests that determine if the user has finished entering the hexadecimal number.

THE SAMPLE PROGRAM

This hexadecimal to decimal conversion program introduces the CASE..OF statement which is used to assign values from a list to the designated variable. In this program, the list is the list of decimal equivalents for the hexadecimal digit entered by the user. The program contains a formula that keeps track of the value of the position of each digit. It also uses a Boolean variable to decide whether it should continue to request input.

```
PROGRAM HexToDec(Input,Output);

TYPE
  YesNo = (Yes, No);

VAR
  Digit, Signal :  Char;
  Number, OldNumber :  Integer;
  Answer : YesNo;  (*  a scalar variable  *)
  Continue :  Boolean;

BEGIN
  OldNumber := 0;
  Writeln('Enter the most significant digit');
  Write('i.e. the one that begins on the far left.');
  Readln(Digit);

REPEAT   (*  Start the REPEAT loop  *)
  CASE Digit OF  (*  start the CASE list which takes the Digit
                     and finds its decimal equivalent  *)
    '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; (* of the CASE list *)

  OldNumber := Number + OldNumber * 16;
  Writeln;
  Writeln('Is there another digit');
  Writeln('after this one? (Yes or 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
  ELSE
   BEGIN
     Continue := False
   END;
  UNTIL NOT(Continue);

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

THE LOGIC OF THE PROGRAM

1. Name the program.

2. Define the scalar **TYPE** "YesNo."

3. Declare the variables:

   * Digit and Signal are **Char**
   * Number is **Integer**
   * Answer is **YesNo**
   * Continue is **Boolean**

4. Initialize the OldNumber to zero and clear the screen.

5. Print the message requesting information:

   * the **Writeln** writes the text and advances the cursor to the next line

   * the **Write** statement positions the cursor at the end of the text and awaits input.

6. Read the input and compare the item to the **CASE..OF** list.

   * the number read by the **Readln** statement is converted to its equivalent in decimal notation. If the user enters an "A", the value is converted to 10.

7. Calculate the current value of the digit and add it to any pre-existing values.

8. Ask if there is another digit to be entered.

9. Enter the **IF** loop:

   * If another digit is to be entered, indicate this by making the Scalar variable "Yes." Otherwise, make the variable "No."

10. Enter the second **IF** loop.

    * If the Scalar variable is "Yes", then make "Continue" True and request the next digit to be entered.

    * If the Scaler variable is "No", then make "Continue" FALSE and exit the **IF** conditional test.

11. When the entry is complete, write the results to the screen.

GENERAL COMMENTS

One of the advantages of Pascal is that you can define variable types to fit your program. In the next part of this manual, "Programming Techniques," you will learn how to define complex types of data. This program, however, introduces the idea of defining your own variables by using a simple Scalar variable. After defining the TYPE **YesNo** as a **Scalar** list of Yes and No, the program declares the variable Answer as a **YesNO** TYPE. The body of the program assigns one or the other value to the Answer variable to indicate if the user wants to enter more digits in the hexadecimal number.

The **REPEAT..UNTIL** statement controls the execution of the program which continues to request hexadecimal digits **UNTIL** the Boolean expression "NOT(Continue)" is False. If this logic seems convoluted, remember that you want to keep executing the **REPEAT** loop until a condition is NOT met.

TOPICS FOR PROGRAMMERS

DATA TYPE: SCALAR

A Scalar variable is actually a list of items. It may represent any one of those items while the program executes. It must be defined under the heading **TYPE**.

The sample program defines the Scalar type **YesNo** as the list (Yes, NO). It then declares the variable **Answer** as a **YesNo** type. Two sample Scalar types are illustrated below. Note that the TYPE must be defined before a variable can be declared as that TYPE.

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

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

The values or items in the user-defined **Scalar** TYPE may not be defined in terms of any other type. In addition, they may not be characters, strings, integers, or real numbers. They can only be a list of items. For example, an item in single quotes like 'A' or an item like 'Sun' is unacceptable since the single quotes indicate a character string.

The principle is simple; the elements of a Scalar variable can only be a list of items which are separated by commas.

The only exception to this rule is explained in the next section on the **Subrange** TYPE.

DATA TYPE: SUBRANGE

The **Subrange** TYPE is a form of the **Scalar** TYPE because it is a list of items. It is different, however, because you need to specify only the first and last items in the range. Obviously, if the subrange is a list of names, the full list must be defined in a Scalar TYPE. Subranges may contain integers since the computer understands the list of numbers as a Subrange of the entire set of Integers it uses. The syntax of the TYPE Subrange is:

```
TYPE
 Name = first item .. last item;
```

The following sample declaration illustrates the use of Scalar and Subrange TYPES. First a Scalar TYPE is declared. Then a Subrange of the full Scalar list is identified. Finally, a Subrange of integers is defined. After the TYPE declarations have been made, Variables are identified as their respective TYPES.

```
TYPE
 Week = (Sun,Mon,Tues,Wed,Thurs,Fri,Sat);
 WorkWeek = Mon..Fri;
 Days = 1..7;
```

```
VAR
  Date : Week;
  WorkDay : WorkWeek;
  DayNum : Days;
```

When the program containing the above declarations executes, the Variable "Date" may be assigned any one of the items in the Scalar list. "WorkDay" can only represent Monday through Friday. "DayNum" can equal any number from 1 to 7. Remember that "Week" is a full Scalar TYPE, "WorkWeek" is a Subrange TYPE, and "Days" is a Subrange of Integers.

The elements of the Subrange do not have to be included in parentheses.

Note: Use a Subrange TYPE to make a list of integers. A Scalar TYPE cannot contain numbers or integers. This restriction prevents you from inadvertently redefining a predefined type.

REPEAT..UNTIL

The **REPEAT..UNTIL** loop is similar to the WHILE loop that you learned in Lesson 5. The statements inside the loop will continue to execute until a specific condition is met. The syntax of the statement is:

```
REPEAT
  command 1;
  command 2;
  command 3;
UNTIL condition;
```

Notice how the REPEAT UNTIL statement differs from the WHILE statement. The WHILE statement declares the condition before it begins to execute commands and continues until the condition is no longer true. The REPEAT UNTIL statement places the test condition at the end of the loop. It continues to loop until the test condition does become true. Since it ends execution when the UNTIL condition is met, the loop does not requires its own END statement.

CASE..OF

In the sample program, you asked the user to enter a number and then determined its decimal equivalent. You might have handled that situation with an entire series of IF tests such as:

```
Readln(Digit);
IF Digit = 1 THEN
  Number := 1;
IF Digit = 2 THEN
  Number := 2;
    .
    .
    .
IF Digit = A THEN
  Number := 10
```

Using a series of IF statements takes a lot of time and effort just to determine which value you want. Pascal uses the **CASE..OF** statement when you want to select an item from a list of possible values. Simply identify the name of the variable between **CASE** and **OF**; then list the possible entries, a colon, and the action to be taken. Remember to END the list and include a semicolon.

With these principles in mind, look again at the sample program. The user enters a value that is name "Digit." The program then sets up a **CASE..OF** list. Each possible entry is listed in single quotes and the value of the variable "Number" is assigned to it. Depending upon the value of "Digit," a corresponding value is assigned to "Number." "Number" is then used to compute the decimal equivalent.

ADVANCED TECHNIQUES

Functions: ORD, SUCC, and PRED

In Lesson 5, you learned how to use the function **Chr**. Pascal has other functions which you can use in your programs. Three of them are especially useful in dealing with Scalar Data Types. Since the items in a Scalar data type are declared in a particular order, you can use that order in designing your program. The following three commands allow you to manipulate the items in a Scalar list.

Ord

Each position in the Scalar list implies a number. In the Scalar TYPE , we defined as "Week", the first position is occupied by "Sun", the second by "Mon", and so on. The **Ord** function will return the value of the position of the item in parentheses. Remember, however, that a computer begins counting with zero. So:

Ord(Sun)	will return	0
Ord(Mon)	will return	1
Ord(Sat)	will return	6

Also, you can use the statement ORD('A'), where A is a character, to determine the ASCII value of that character.

Succ and Pred

Succ (succeeding) and **Pred** (preceding) also operate on Scalar lists. **Succ**(item) will return the element in the list that follow "item." **Pred**(item) will return the element that precedes it. Using the Scalar Type we defined as "Week":

Succ(Mon)	will return	Tues
Pred(Mon)	will return	Sun

Never, however, try to find an item which precedes or follow the beginning or ending of the list. It will produce errors. "**Pred**(Sun)" won't make sense to the computer since nothing comes before Sun in the list.

If several Scalar types are declared, items in the different lists will have the same order values. For example, if the days of the week and the months of the year are declared as two Scalar variables, Sunday and January will have the same ordinal values.

CONCLUSION

This is the last section of the tutorial which uses sample programs to introduce you to the basic concepts of the Pascal language.

You have been introduced to the elementary Pascal data types and most of the command statements.

In this lesson alone, you have learned about

* Scalar data types
* Subrange data types
* The REPEAT..UNTIL statement
* The CASE..OF statement
* The functions: Ord, Succ, and Pred

We have also tried to include some suggestions about what makes a good program. Unfortunately, some of the sample programs don't illustrate those techniques. That is because Pascal allows you to define more than data types. It also lets you define your own procedures and functions. Good programming technique defines subroutines that the main body of the program can call whenever it needs them. The next section of the tutorial will explain how to write your own procedures and functions, and how to manipulate blocks of information in the form of records and files.

IV TUTORIAL: PART 2

This part of the tutorial demonstrates the real power of Pascal. It explains how to define routines that permit you to create subprograms called **Procedures** and **Functions** which your program can call with a simple command. This section also explains how to define complex data types. The ability to define routines and declare data types greatly increases the power and complexity of the programs you can write. Finally, a discussion of pointers shows you how to access memory locations directly.

In this section you will learn how to declare and use:

* Procedures
* Functions
* Arrays
* Sets
* Files
* Pointers

Procedures and **Functions** are Pascal's equivalent of subprograms. They enable the main body of the program to call routines that you have defined. **Arrays, Records, Sets,** and **Files** are data structures which your program can define, manipulate, and store. **Pointers** enable you to control dynamic variables.

* **Procedures** are a group of instructions that execute a specific task or execute a group of instructions that perform a specific task.

* **Functions** accept values that are passed to them, perform calculations with those values, and return a value to the main body of the program.

* Arrays let you store elements of a single type of data.

* **Records** let you define complex arrangements of data types.

* **Sets** are used to manipulate data.

* **Files** store data sequentially.

* **Pointers** allow you to access dynamic variables and/or memory locations and control the values stored in those locations.

8. PROCEDURES

A **Procedure** is a subprogram or subroutine consisting of one command or a set of commands that perform a single task. For example, you might define a procedure that performs a task by executing a series of commands. The main program then executes this task, whenever it is needed, by calling the procedure. This saves you the trouble of repeatedly defining the operation every time you want the program to do the same task.

This gives the programmer a very powerful tool. You can write any number of separate procedures which are executed by the body of the program -- which itself is relatively small. This enables the programmer to demonstrate the logic of the main program without confusing it with the details of the subroutines. Have you ever read a BASIC program and wondered just where you were in it and what it was doing? A good Pascal program never lets that happen. Because the program is modular, it is also easier to isolate and fix any "bugs" that it has.

DECLARING A PROCEDURE

The form of a procedure is almost identical to that of a program. It is identified by the word **PROCEDURE** and given a name.

Important:

1. Do not use a name that is already defined by Pascal (i.e., a predefined word).

2. Do not choose a name with more than 256 characters.

3. Choose a name that clearly indicates the task it performs. This makes it easier to understand the program.

4. Any constants, variables, or user-defined data types, not declared in the main program must be declared in the procedure.

5. The sequence of instructions must be encased with a BEGIN/END pair.

There are, however, a few important differences between Programs and Procedures:

1. The END statement of a Procedure is followed by a semicolon (;) and not by a period.

2. A list of Parameters may follow the Name of the procedure. This list is enclosed in parentheses, and it is similar to the (Input,Output) declaration in a Program. This list tells Procedure what kind of information it will receive from the main program. Parameters are discussed in detail below.

Here is a sample Procedure that the main program might call to print a menu that the program frequently uses. Remember that it is not an entire program by itself.

```
(*-------------------------------------------------*)
PROCEDURE Menu;

BEGIN
  Writeln;
  Writeln;
  Writeln('MENU':15);
  Writeln;
  Writeln;
  Writeln('Press the number of your choice.');
  Writeln;
  Writeln('1.  Item A':  10,  '2.  Item B':20);
  Writeln;
  Writeln('3.  Item C':  10,  '4.  Item D':20);
  Writeln;
END;
(*-------------------------------------------------*)
```

Notice that there is no Parameter list in this Procedure. The main program simply calls **Menu** whenever it wants to print the choices. Also notice that the Writeln statements use the colon to indicate how much space to save for the text. It produces a cleaner output to the screen. Finally, after printing the menu, control automatically returns to the place in the main program where the procedure was called from. The next statements in the program would read the choice and take the appropriate action.

USING PROCEDURES

Pascal requires that any information used by the main program be defined before it is called. In general, the order of definitions following the Program or Procedure declaration is:

1. Labels (They are discussed in the next lesson)
2. Constants
3. User-defined data types
4. Variables
5. Procedures and Functions (Functions are the topic of the next lesson)
6. Main Program Body

This partial program below illustrates how to call the Procedure defined above. Notice that the Procedure receives no information from the main program, which simply calls it to print the menu. This is the simplest form of a Procedure. Once the Procedure runs, the main program reads the input and performs a specific task, which would be written in another Procedure.

```
PROGRAM CallMenu(Input,Output);

VAR
  Choice : Integer;

PROCEDURE Menu;       (* Begin Procedure *)

BEGIN
  Writeln;
  Writeln;
  Writeln('MENU':15);
  Writeln;
  Writeln;
  Writeln('Enter the number of your choice.');
  Writeln;
  Writeln('1.  Item A':  10,  '2.  Item B':20);
  Writeln;
  Writeln('3.  Item C':  10,  '4.  Item D':20);
  Writeln
END;                  (* End Procedure; note the semi-colon *)
```

```
BEGIN           (* Main Program *)
  Menu;
  Readln(Choice);
  (*statements which call a Procedure or Function indicated
        by the Choice selection..*)
END.
```

COMMENTS

1. The sample program first declares the Variable item, "Choice," which holds the value of the item selected from the menu.

2. After the Variable is declared as an integer for the main program, the Procedure is defined. Since all it does is print the menu, it contains no constants or variables of its own. But it could!

3. Notice the use of the Reserved positions after each item in the menu. This lets you control output to the screen and makes the menu more readable.

This simple Procedure can be very useful. Whenever you want to print the menu, have the main program simply list the command "Menu." But what if you want the main program to communicate some information to the Procedure or if you want the Procedure to return the results of some actions it has taken back to the main program? That involves **Parameters**, which is the topic of the next section.

PARAMETERS

When a Procedure is to receive information from or return information to the main program, a Parameter List follows the name of the Procedure. It is similar to the (Input,Output) segment in a program because it tells the Procedure what kind of information it will expect. This list is enclosed in parentheses. When the main program calls the Procedure, it must indicate what values it wishes to pass to the Procedure. It does this by stating the name of the Procedure and enclosing the information to be passed to it in parentheses.

An example should clarify the issues involved. You are writing a long program that prints text to the screen and you're tired of writing all those Writeln statements to make the output clear. Wouldn't it be easier to write a Procedure -- let's call it Skip -- that the program can execute, and in addition, indicate how many lines to leave blank? The Procedure must be told to expect an Integer from the program which, in turn, it will use to produce the desired number of blank lines on the screen.

Such a Procedure is listed below:

```
(*-----------------------------------------------*)
PROCEDURE Skip(Number:Integer);

VAR
  Count  :  Integer;

BEGIN
  FOR Count  :=  1 to Number DO
    Writeln
END;
(*-----------------------------------------------*)
```

In the program you could issue a statement like

```
Skip(5)
```

This statement calls the Skip Procedure and tells it that "Number" equals five. When the Procedure executes, it uses 5 as the Number which determines the end of the FOR loop. Simple, right? Well almost. This Procedure doesn't have to return any information to the main program. What if you want the Procedure to receive information, do something with the data, and return the new information to the main program. Then, you must declare another type of data in the parameter list -- a Variable Parameter.

VALUE AND VARIABLE PARAMETERS

The Skip Procedure received one item of data from the main program, i.e., the number of blank lines to print. It did not alter that value, and it did nothing to affect the main program. In the parameter list, you simply indicated that the Procedure should expect an integer that it calls "Number." This is a **Value Parameter**.

If you want the main program to feed data to the Procedure, have that data processed in some way, and then return it to the main program, you must declare a Variable Parameter. It's easy to do. Simply precede the name and type of data item the Procedure should expect with the variable indicator, Var.

The following Procedure will expect a value that the main program passes to it, and will associate that value with the name "Number" which it has been told is an Integer. The Procedure then doubles "Number." From now on, when the main program uses the item, "Number," it uses a value twice the original value.

```
(*-------------------------------------------------*)

PROCEDURE Double(VAR Number : Integer);

BEGIN
  Number := Number * 2
END;

(*-------------------------------------------------*)
```

If Number has been assigned the value 3 and the main program states:

```
        Double(Number)
```

the identifier "Number" will be equated to 6.

MULTIPLE PARAMETERS

A Procedure can receive any number of data items from the main program -- as long as it has been told in the Parameter List the name and type of items it should expect.

The only caution in using multiple parameters is that the values listed in the parentheses with the calling statement must correspond to the parameters declared in the Procedure's Parameter List. The following statement (procedure declaration) identifies a Procedure which expects four values from the statement that calls it.

PROCEDURE Mult(VAR X, Y: Real; Z: Real; VAR Number: Integer);

The statement in the program's body might be something like this:

```
Mult(item1, item2, item3, item4);
```

In this case, the Parameter List in the Procedure tells it that the first three pieces of data are real numbers and that the fourth is an integer. Since items 1, 2, and 4 are Variables, if their values are changed during the execution of the Procedure, the main program will now use those new values as it continues to run. The third value, however, is merely passed to the Procedure and is not altered after the Procedure is executed.

Note: When the main program passes a value to a Procedure, it is important that the program has initialized that value before it determines the actual value to be passed. For example, if the main program passes a variable named **Item2**, it should be assigned a value before it is passed to the Procedure. This insures that the variable does not contain any garbage left over from other parts of the program.

If the program has declared and initialized variables named "Item1," "Item2," and "Item 4," then the following call to a Procedure is just as valid as the previous Procedure call:

```
Mult(Item1, Item2, 30, Item4);
```

This statement passes the values currently represented by "Item1," "Item2," and "Item4," to the Procedure. Since the Procedure does not demand a variable in the third position, the call from the main program indicates a value (i.e., a number, a mathematical expression, or a variable name). After the Procedure

runs, Items 1, 2, and 4 may be changed. The value in the third position is unchanged.

Finally, the third position may contain any expression that consists entirely of values that the program already understands. The statement

```
Mult(Item1, Item2, Item2/10, Item3);
```

is just as acceptable to the Procedure as the previous example. Any arithmetic operator may be used in defining a Value Parameter.

FORMAL AND ACTUAL PARAMETERS

The procedure and the main program should identify the same variables by different names. The procedure identifies formal parameters. The calling statement from the main body of the program identifies the actual parameters.

Formal Parameters

The Procedure associates the values passed to it by the main program with the names identified in the parameter list. If a Procedure expects the three values:

```
(VAR Item1, Item2 : Real; Item3 : Char);
```

it will accept values from the main program that correspond to those data types and in the order in which they were declared. It associates those values with the names "Item1," "Item2," and "Item3." (Obviously, the data names will not be "Item," but something more significant.) The "names" listed in the Procedure's Parameter list are called the **Formal Parameter List.**

Actual Parameters

The main program might have identified certain values as variables with their own unique names. When a program calls a Procedure, the variables it passes to the Procedure are known as the **Actual Parameter List.** The values that result from any calculations the Procedure performs will be known to the main program by their **actual** names.

When using parameters, the following rules apply:

1. The number of actual parameters passed by the main program must exactly correspond to formal parameters declared in the procedure.

2. The types of data in the actual and the formal parameter list must be identical. Trying to pass a Real number to a procedure that expects an Integer will cause the program to crash.

CONCLUSION

This chapter has explained how to declare and manipulate Procedures. In general, a Procedure is like a Program. It may contain its own list of constants, variables, and, on occasion, other Procedures. Procedures may use a Parameter List to transmit information between the main program and the individual Procedure.

The next section explains another type of routine, the Function.

(This page left blank for your notes.)

9. FUNCTIONS

A **Function** is a subprogram, or subroutine, that receives values from the main program or procedure and returns a **single value** that is identified by the function's name.

When the program or procedure calls a function, it passes values, called **Arguments**, to that function. Arguments can be sent directly to the function or through variable or constant identifiers. The function performs its operations and assigns the result to a variable identified by the function's name. This variable can only be altered by the function itself and may not be changed outside the function.

For example, a Function named **Area** receives 2 values from the main program and calculates the area of a rectangle. If the main program called the function Area(a,b), and the values of a and b had previously been determined to be 3 and 4, the function Area would return the value 12 to the main program.

DECLARING A FUNCTION

A Function declaration resembles a Procedure declaration. Examine the following declaration of the Function **Area**.

```
(*-------------------------------------------------------*)

FUNCTION Area(Length, Width : Real): Real;

BEGIN
  Area := Length * Width
END;

(*-------------------------------------------------------*)
```

Although a Function declaration may resemble a Procedure, make certain that you understand the differences.

1. The label is FUNCTION.

2. The function's name is followed by the equivalent of a Parameter List. These items are called the **Arguments** of the function, and they are enclosed in parentheses. After the Arguments are named and their data types defined, the computer must be told what type of data the result of the calculations will yield. To do this, place a colon (:) after the parentheses enclosing the Arguments, and define the type of data that the function will yield.

   The declaration *FUNCTION Area(Length, Width : Real): Real*; means that the Function named Area will expect two Arguments. The first is known as "Length" and the second as "Width." Both of these values are Real numbers. After the function performs the calculations indicated in the Function body, the Function Area returns a Real number to the main program. This number is identified in the program by the Function name, **Area**.

3. While Procedures transmit many pieces of data between the main program and itself, Functions can only transmit one value back to the main program. That value is whatever the result of its calculations happens to be.

4. Functions, like Procedures, should be declared before the body of the main program, but after the definitions of Labels, Constants, and Variables that will be used in the main program.

5. The Arguments given to the Function in the main program are not altered by the Function--after all, it just uses them in the calculation. A Procedure, on the other hand, usually does change the parameters that were passed to it.

USING FUNCTIONS

Use Functions to define any set of operations that a program requires. This gives you a great deal of flexibility in the design of your program.

The sample program below asks the user to input the 3 dimensions of a solid object. It then uses the Area Function to calculate both the Area and the Volume.

```
PROGRAM Math (Input,Output);

VAR
  Length, Width, Height, Volume : Real;

FUNCTION Area(L, W : Real): Real;
BEGIN
  Area := L * W
END;

BEGIN (* Body of the main program *)
  Writeln;
  Writeln;
  Writeln;
  Writeln('This program calculates the Area');
  Writeln;
  Writeln('and Volume using values you enter.');
  Writeln;
  Writeln;
  Writeln('The length in inches is: ');
  Readln(Length);
  Writeln;
  Writeln('The width in inches is: ');
  Readln(Width);
  Writeln;
  Writeln('The height in inches is: ');
  Readln(Height);
  Writeln;
  Writeln('The Area is ', Area(Length, Width): 4:2,
      'square inches.');
  Volume := Height * Area(Length,Width);
  Writeln;
  Writeln('The Volume is ', Volume: 5:2,
    ' cubic inches.')
END.
```

Comments

1. The program first declares the Variables that it will use. It then declares the function **Area**. In parentheses, it indicates the two Arguments which **Area** expects to receive. It then declares the data type of the result. The body of the Function is obvious. Note, however, that the END statement of the Function is followed by a semicolon (;) not a period.

2. The body of the program prints the messages and requests the input. Notice the way the Program uses the Function. In the first instance, **Area** is part of a **Writeln** statement. In the second, it is part of the **Volume** calculation.

3. A function can be called as part of an arithmetic or relational statement. A Procedure, on the other hand, always requires a separate statement. This is because a Procedure often returns several values through its parameter list.

RELATED TOPICS

Predefined Functions

Kyan Pascal includes a number of frequently used mathematical functions. You don't have to define them to use them in a program. That has been done for you. If X is a Real number or an Integer, then the following values are defined:

Abs(X)	the Absolute value of X
Sqr(X)	the Square of X
Sqrt(X)	the Square Root of X
Sin(X)	the Sine of X (when X is in radians)
Cos(X)	the Cosine of X (when X is in radians)
Arctan(X)	the Arctangent of X (the result is in radians)
Ln(X)	the natural logarithm of X
Exp(X)	the result of e, the natural base, raised to the power of X

The Function Odd

The Odd function can be used to convert Integer data information into Boolean data. The Argument of the Odd function, however, must always be an Integer.

If the Argument of the Odd function is an odd number, the Function returns the Boolean value **True**. Otherwise, it returns **False**.

For example, if the Variable, "Number," currently represents the Integer 3, the statement **Odd(Number)** returns the Boolean value **True**. You can use this Function to make decisions based on the type of number the program is manipulating.

CONCLUSION

This section has introduced the concept of Functions and explained how to use them in your programs. It has explained the differences between a Function and a Procedure. The next section explains how to "Nest" Functions and Procedures and introduces the concept of "Scope."

(This page left blank for your notes.)

10. SCOPE AND NESTS

Procedures and Functions may be declared within other Procedures and Functions. This is called **Nesting**. In the tutorial lessons, for example, you learned how to nest IF conditions. If you nest Procedures or Functions in a program, it is important that you keep track of all the Variables being sent back and forth between the subprograms. These Variables possess an attribute known as **Scope**. A Variable's Scope indicates the range of the program for which the Variable contains a valid value.

* A Variable that has **Global** scope holds its value throughout the entire program.

* A Variable the has **Local** scope holds its value only during the part of the program that uses it.

Nesting and Scope are concepts used in a Pascal program. They are not statements or predefined words.

SCOPE

Scope refers to the extent of the program for which a Variable retains its value. There are two types of Scope:

Global Global Variables are those variables that the main program declares. They continue to represent values throughout the program. The actual value may change as a result of calls to Procedures or Functions. But whatever actual value they represent, the name that represents them remains the same.

Local Local Variables are those variables that are declared within a Procedure or Function and are meaningful only within that subprogram. The main program, for example, does not have access to a Local Variable. You can, however, assign the value of a global variable to a local variable (e.g., Local := Global).

EXAMPLES OF SCOPE

The following program illustrates the concept of Scope. It asks the user to enter two numbers and then calls a Procedure to reverse their order. The Global Variables "A" and "B" are used by the main program. The Local Variable "Y" has meaning only within the Procedure.

```
PROGRAM Exchange(Input,Output);

VAR                 (* Global Variables *)
  A, B : Real;

        PROCEDURE ExcVal (VAR X1,X2: Real);

        VAR         (* Local Variable *)
          Y: Real;

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

BEGIN               (* Main Program *)
  Writeln;
  Write('Enter the first number: ');
     Readln(A);
  Writeln;
  Write('Enter the second number: ');
     Readln(B);
  ExcVal(A,B);
  Writeln;
  Writeln;
  Writeln('Now ', A: 7: 2, ' is first');
  Writeln('and ', B: 7: 2, ' is second.')
END.
```

COMMENTS

1. The concept of Scope and the technique of communicating data between the main program and subprograms are the most difficult topics you will encounter when using Pascal. If necessary, reread these comments until you are certain you understand them.

2. The main program declares the Global Variables "A" and "B." When Pascal declares any Variable, it sets aside a location in memory and gives it the Variable's name. Since "A" and "B" were declared by the main program and are, therefore, Global Variables, any part of the Program has access to the values stored in those locations.

3. The Procedure, which is a subprogram, can identify its own memory locations to store the data that it needs. It establishes locations for "X1" and "X2" where it temporarily stores the values passed to it by the main program as "A" and "B." It also reserves a memory location named "Y" where it stores the value it knows as "X1."

   * "Y" is known as a dummy variable because it is set aside simply to hold the first value, X1, while the transfer of X2 to X1 takes place. If you didn't store X1 in Y, the first value, X1, would be lost when you loaded X2 into X1.

4. Since "X1," "X2," and "Y" are Local to the Procedure, they are meaningful only within that Procedure. If the main program tried to write "Y" or "X1," the program simply would not know what those values were.

5. The main program knows the values of "X1" and "X2" only by its identifiers of "A" and "B." When it writes those Global Variables after calling ExcVal, the values are reversed.

Bad Programming -- an example to avoid

You may have noticed an interesting fact about the declaration of the Procedure. Since "A" and "B" are Global Variables, why not use them in the Procedure's parameter list? That would avoid all the confusion about the "X1" "X2", "A", and "B" variables. In fact, this program would run just as well if you did substitute "A" for "X1" and "B" for "X2." Try it, the programs would be identical except that the Procedure would look like this.

```
PROCEDURE EncVal(VAR A,B: Real);

VAR
  Y : Real;

BEGIN
  Y := A;
  A := B;
  B := Y
END;
```

It works. But it can also make a real mess of things in a long program. If you have ever programmed in BASIC, used a variable, run the program, and found that it crashed or produced nonsense, you can appreciate why Pascal is very strict about Global and Local Variables. In BASIC all variables are Global. If you use a variable frequently, it may have been altered by another part of the program. Consequently, what you see when you read a line of BASIC code is not always what you will get. Pascal tries to avoid this.

NOTE: As a general rule, do not use Global Variables to transfer information between the main program and subprograms. Use either Functions or procedures with Parameter Lists to convey data between the main program and the subprograms.

NESTING

As you have seen, a program can contain either a Procedure or a Function. Similarly, Procedures and Functions can contain any combination of other Procedures and Functions. This is called **Nesting**. The concept is simple, but its results can be very complex. The main program can call a Procedure which itself contains several functions. Or vice versa. The combinations are staggering.

Good programming doesn't require convoluted nestings. In fact, if you find you are writing bizarre, nested Procedures and Functions, you probably haven't thought out the problem you are trying to solve.

Nevertheless, it is important to realize that Procedures and Functions can be nested and that the nesting affects the Scope of the variables declared within those Procedures and Functions.

* Nesting affects the Global and Local quality of Variables.

Relative Global and Local Variables

The Variables declared in the main program are Global throughout it. That is, all parts of the program have access to the values represented by the Variables. All parts of the program can also change those values.

The Variables declared in a subprogram are local to the subprogram.

If a subprogram contains nested subprograms, the Variables declared in the main subprogram are Global to all the nested subprograms. Variables within nested subprograms, however, are not available to each other. They are Local to each nested subprogram.

Before discussing an example of nested programs and the relative Global or Local values of the variables, an old-fashioned outline should make the distinctions and limitations clear.

I. Main Program
    A. Procedure 1
        1. Function 1
        2. Function 2
    B. Procedure 2
        1. Function 3
        2. Function 4
    C. Function A
        1. Procedure a
        2. Function 5

Now for some complicated logic:

A variable declared in the Main Program is available to every Procedure and Function in the program.

A variable declared in Procedure 1 is available to Functions 1 and 2. It is not available to any other Procedure or Function.

A variable declared in Procedure 2 is available to Functions 3 and 4, but not to Procedure 1 or Functions 1, 2, A, or 5.

A variable declared in Function A is available to Procedure a and Function 5. It is not available to any other Procedure or Function.

A variable declared in Function 1 is not available to any other Function.

A variable declared in Procedure a is not available to any other Procedure or Function.

By now you get the picture. Pascal programmers use the concepts of Global and Local Identifiers to refer to these conditions.

A Variable or Constant in the main program is Global to the entire program.

A Variable or Constant in Procedure 1 is Global to Function 1 and Function 2, but Local in terms of the main program and the other subprograms.

A Variable or Constant in Procedure 2 is Global to Function 3 and Function 4, but Local in terms of the main program or any other subprograms.

A Variable or Constant in Function A is Global to Procedure A and Function 5, but Local with respect to all other Procedures and Functions.

That should settle the issue; and an example program should make everything clear.

An example of NESTING

The following program illustrates the use of nesting. It asks the user to decide whether to Add, Average, or Subtract two numbers. Next, it requests the two values. It then calls the Procedure "Choice," which determines the chosen function. Based upon the selection, the Procedure picks the right function from a group of nested functions to perform the appropriate calculations. Finally, the main program prints the results to the screen.

```
PROGRAM  Calc(Input,Output);

VAR
  Pick : Char;
  X, Y, Answer : Real;

   PROCEDURE  Choice (Select: Char; N1, N2: Real);

         FUNCTION Sum(Val1, Val2: Real) : Real;
            BEGIN
              Sum := Val1 + Val2
            END;

         FUNCTION Average(Val1, Val2 : Real) : Real;
            CONST
              D = 2;
            BEGIN
              AVERAGE := (VAL1 + VAL2)/D
            END;

         FUNCTION Difference(Val1, Val2 : Real) : Real;
            BEGIN
              Difference := Val1 - Val2
            END;

     BEGIN  (* of Procedure containing nested Functions *)
       CASE Select OF
           'S' : Answer := Sum(N1,N2);
           'A' : Answer := Average(N1,N2):
           'D' : Answer := Difference(N1,N2)
       END  (* of CASE *)
     END;  (* of PROCEDURE *)

BEGIN (* of Program *)
  Writeln;
  Writeln;
  Writeln('This program computes the Sum,');
  Writeln('Average, or Difference of two numbers.');
  Writeln;
  Writeln;
  Writeln('Enter S, A, or D :   .');
```

```
    Readln(Pick);
  Writeln;
  Write('Enter the first number: ');
    Readln(X);
  Writeln;
  Write('Enter the second number: ');
    Readln(Y);
  Writeln;
  Choice(Pick, X, Y);  (* Call Choice Procedure *)
  Writeln;
  IF Pick = 'S' THEN
    Writeln('The sum is ', Answer:5 :2);
  IF Pick = 'A' THEN
    Writeln('The Average is ', Answer:5 :2);
  IF Pick = 'D' THEN
    Writeln('The Difference is ', Answer:5 :2)
END.
```

The Procedure **Choice** contains three nested Functions. The calling statement from the main program passes the Choice selection through the variable, **Pick**. Choice, in turn, selects the appropriate Function and transmits the two values to the function, which performs the necessary calculations.

Rather than explain the obvious, a chart should indicate all the relationships between the Variables and Constants used in the program.

Table of Relative Scope

IDENTIFIER	CONTEXT	SCOPE in terms of	SUBPROGRAM
X	MAIN	G	Entire
Y	MAIN	G	Entire
Answer	MAIN	G	Entire
N1,N2	Procedure	G	All Functions
		L	Main
D	Average	L	Average

COMMENTS

1. This program uses a global variable, "Answer," to transmit the results of the Procedure and the Function it executes to the main program.

2. If the Functions called by the Procedure were not nested within the Procedure "Choice," they could not use the "Char" value which indicates the Function to be executed.

RELATED TOPICS

Types

Any type of data may be considered to be Global or Local. This includes scalar data types and Arrays.

Forward References

Kyan Pascal allows you to write programs that call and execute a Procedure or Function before it has been declared. This is called a **Forward** reference. Whenever a forward reference is used in a Pascal program, you just indicate it as such by including a semicolon and the term "FORWARD" after the Parameter or Argument list).

The sample program illustrates how to use this technique. It asks the user to enter 2 numbers.

```
PROGRAM COMPUTE (Input,Output);

VAR
  X,Y  :  Integer;

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

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

          FUNCTION Factor;

          CONST
            SmallNum  =  1;
          BEGIN
            Factor  :=  SmallNum MOD X + J;
            Y := Factor
        END;  (*  of FUNCTION  *)

BEGIN  (*  MAIN PROGRAM  *)
  Write('Enter an Integer: ');
    Readln(X);
  Writeln;
  Write('Enter another integer: ');
    Readln(Y);
  Bisect(X,Y);
  Writeln;
  Writeln('The Answer is ',  Y:  2)
END. (*  of PROGRAM  *)
```

COMMENTS

1. Note how the program is careful to declare the type of information that it expects: X and Y are integers. You must keep items like this straight when designing your program. If you don't and if you try to pass those values to Procedures or Functions, the program will crash.

2. The PROCEDURE "Bisect" is able to execute the FUNCTION "Factor" because the function has been declared as a FORWARD Reference before "Bisect" is declared. The FORWARD declaration must include the formal parameter list. Later, when the FUNCTION is defined, the parameters and FORWARD declaration are not repeated.

GOTO: Unconditional Branches

Although it should not be done regularly, Pascal allows you to use GOTO statements as long as you have labeled the line to GOTO. Many Pascal programmers will not use a GOTO statement because it violates the principle of modular, top-down programming. But the command is available if you need it.

Mis-using GOTO statements will not cause the Program to crash. You can use such statements whenever you want. GOTO statements, however, promote messy programming techniques and should be avoided. GOTO enables you to use unconditional branching. The GOTO command precedes the LABEL that identifies the line that will receive control.

The following rules govern the use of GOTO statements.

1. The LABEL is an Integer followed by a colon (:).

2. The maximum size of the LABEL is 4 digits.

3. The LABEL must be declared in the proper position immediately after the Program, procedure, or function declaration.

4. The LABEL must begin in column 1 or 2 of the program.

The following program uses GOTO statements to control the sequence of commands.

```
PROGRAM GoExample (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('This line is always skipped.');
35:  Writeln('The End.')
END.
```

Labels used in a Function or a Procedure must be declared locally. Do not try to declare a label in the main body of the program and then try to use that label in a subprogram. You can use a GOTO to jump forward or backwards within a subprogram; you can also use GOTO to leave a subprogram and return to the main program. Do not use the GOTO statement to jump from the main program to a Procedure or Function.

CONCLUSION

This chapter has introduced a number of principles that determine the Scope of a variable. It has also shown how nesting affects Scope. When you write complex programs, always make certain that you are aware of the Scope of the variables when you try to use them. Calling a variable that can not be accessed only leads to problems when you try to run the program.

11. ARRAYS

An ARRAY is a sequential collection of elements of the same data type. You can declare an ARRAY of Integers, an ARRAY of Characters, or even an ARRAY of an Array. Basically, you can declare an ARRAY of any data type.

This section explains the use of ARRAYS. It demonstrates:

* declaring multi-dimensional ARRAYS
* adding ARRAYS
* passing ARRAYS as parameters

Think of an ARRAY as a group of consecutive memory locations, each of which holds one item in the ARRAY. The ARRAY below, for example, holds 10 items. Notice that each item in the array has an identifying number below it. "c," for example, is the 3rd item in the ARRAY named String.

```
|T|e|c|h|n|i|q|u|e|s|  STRING
 1 2 3 4 5 6 7 8 9 10
```

DECLARING AN ARRAY

If an ARRAY is part of a User-Defined data type, it should be declared under the TYPE heading and it must indicate the data type of the elements. The form of the declaration is:

```
ArrayType = ARRAY[First .. Last] OF Element Type
```

A typical declaration of a String that will hold a number of characters is:

```
String = ARRAY[1..10] of Char;
```

1. In this example, **String** is the user-defined ARRAY type. Any name may be used as the ARRAY type identifier, but it should be relevant to the use of the ARRAY.

2. The **Subscripts**, which are often referred to as the Index values, indicate the number of elements in the Array. The first and last numbers are separated by an ellipsis (..) and the expression is enclosed in brackets. Each item in the ARRAY is identified by the appropriate subscript value.

3. The **element type** declares the type of data that comprises the ARRAY. Any predefined or user-defined data type may be used, but only one data type may compose a single array.

After the ARRAY has been defined as a String, you can declare a variable as the TYPE String which uses the ARRAY. This declaration appears under the VAR heading. The following is a typical declaration of a Variable that is a String.

```
VAR
  Line : String;
```

After the declaration, the body of the program may use the Variable "Line" to contain up to 10 characters.

USING AN ARRAY: Strings

As noted above, before you use an Array type, a variable must be identified as of that type. The Variable name then represents the array, and that name is used in the program. For an example of single dimension ARRAYS, refer back to Lesson 5. One of the most common uses of Arrays is defining a string. The two points to remember about Strings are:

* If you define the String as an ARRAY of Char, the String must contain the exact number of items defined in the ARRAY declaration.

* If the String entered by the user does not fill the array , the rest of the array will be filled with spaces. If the string is bigger than the array, the last characters in the string will be lost. (Refer to Lesson 5 if you don't understand what these items mean.)

The following program declares a TYPE identified as "String," and a Variable named "Line," as a String Type. It asks the user to enter a String of data. It then asks for an index value and prints the corresponding letter from the String.

```
PROGRAM Locate (Input, Output);

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

VAR
  Line  :  String;
  X, Count  :  Integer;

BEGIN
  Writeln('Enter a line of no more');
  Writeln('than 15 characters.');
     Readln(Line);
  Writeln;
  Writeln;
  FOR Count  :=  1  TO  3  DO
   BEGIN
         Writeln;
         Writeln('Enter an Index value from 1 to 15.');
         Readln(X);
         Writeln(Line[X],  '  is the  ',  X,  '  Character in the string.')
   END
END.
```

Comments

1. First declare the data TYPE "String" as an ARRAY of 15 characters.

2. Declare **Line** as a String type, and "X" and "Count" as Integer variables.

   * X will hold the Index value entered by the user.

   * Count keeps track of the number of Index requests that are made.

3. **Line** is used to determine the corresponding character in the String. Single dimension arrays use one index value, enclosed in brackets, to isolate an element in the string.

MULTI-DIMENSIONAL ARRAYS

ARRAYS OF ARRAYS

Multi-dimensional ARRAYS are easier to illustrate than to explain. Perhaps the best way to think of a multi-dimensional ARRAY is as a table that has a number of rows, with each row consisting of a number of columns. The figure below illustrates a two-dimensional array. Each row in the ARRAY consists of three data items. One item is loaded in each of the three columns. There are four rows in the ARRAY.

		COLUMN #		
		A	B	C
R	1	2	7	5
O	2	3	12	41
W	3	9	1	15
#	4	11	3	71

A Multi-dimensional Array

Declaring Multi-dimensional ARRAYS

There are several ways to declare multi-dimensional Arrays. The first is to declare the ARRAY characteristics of the ROW itself, and then to declare the numbers of ROWS in the table. The following declares a row of three Real numbers. Then, it declares a matrix of four ROWS that will form the table.

Finally, it assigns the Variable "Table" to represent the entire matrix defined by the TYPE "TableType."

```
TYPE
 Row = ARRAY[1..3] OF Real;
 TableType = ARRAY[1..4] of ROW;

VAR
 Table : TableType;
```

The Variable "Table" now represents an empty table that is similar in outline to the table described above. There are four ROWS of data, and each ROW can contain three different Real numbers. Note that in the table each item or value is uniquely identified by the Row and the Column number of its space. Also note that, at this point, the table itself is empty. It only consists of an empty four row by three column grid of spaces.

An alternative method of defining two-dimensional ARRAYS combines the two TYPE declarations into one statement. The values of the number of the ROW Array are simply indicated before the number of items in each ROW. In other words, you are declaring a table of X ROWS when each ROW consists of Y columns or items of data.

Notes: 1. The data type of the items must be the same.

2. When using a matrix that consists of two values, the first refers to the ROW, the second to the COLUMN.

In other words, a declaration that consists of:

```
TYPE
 TableType = ARRAY[1..10, 1..7] of Integer;

VAR
 Table : TableType;
```

would set aside, in the computer's memory, an empty table that is ten ROWS long with each Row consisting of seven columns. Each column, obviously, can hold one item of data.

Subscripts or Index Values

When using multi-dimensional ARRAYS, it is often necessary to access a specific item in the ARRAY. If you have understood the discussion in the previous paragraphs, you already realize that you identify any entry in the two dimensional table by using the ARRAY's identifier and its ROW and COLUMN coordinates in brackets. In the sample two-dimensional array that was illustrated previously, the statement:

```
Writeln(Table [2,1]);
```

prints 3 on the screen. First, it determines the ROW and then the COLUMN of the table. Then it gets the value stored in that location. Finally, it prints the item on the screen.

The numbers used in the brackets to indicate the ROW and COLUMN matrix locations can themselves be variables. This allows you to set up loops to either clear the memory locations, enter new data into each location, or read and write the information that is already there.

A typical example of using loops to control the entry of data into an Array is the following program which creates a table of values. It uses the Variables "Subrow" and "Subcol" to indicate the position in the Array that will hold each number the user enters.

```
PROGRAM Matrix(Input,Output);

TYPE
 MaxType = ARRAY[1..4,1..3] OF Real;

VAR
 Matrix : MaxType;
 Subrow, Subcol : Integer;

BEGIN
 FOR Subrow := 1 TO 4 DO
  FOR Subcol := 1 TO 3 DO
   BEGIN
    Write('Matrix element ', Subrow: 3, Subcol: 3, ' is: ');
     Readln(Matrix[Subrow,Subcol])
   END;
```

```
    FOR Subrow := 1 TO 4 DO
      BEGIN
        Writeln;
        FOR Subcol := 1 TO 3 DO
            BEGIN
              Write('Matrix element in ', Subrow, ' ',
              Subcol, 'is ', Matrix[Subrow, Subcol] :7 :3)
          END (* Subcol FOR loop *)
       END (* Subrow FOR loop *)
END. (* Main Program *)
```

COMMENTS

1. Nested FOR loops determine the locations within the Table. The outer FOR loop keeps track of the current row. The inner FOR loop keeps track of the number of items in each row. The inner loop gets three items before passing control to the outer loop which increments the Row number and begins the process again.

2. The Variables "Subrow" and "Subcol" are used in both the loop control statements and the current location in the ARRAY "Table." The expression Matrix[Subrow,Subcol] indicates the element of the Array that is represented by the current values of "Subrow" and "Subcol."

3. The punctuation of the program body may seem confusing. Let's review the rules:

   * Every BEGIN statement must have a corresponding END statement.

   * The END statement that concludes a part of the Program is followed by a semicolon (;).

   * The line that precedes an END statement is **NOT** punctuated.

   * END followed by a period indicates the conclusion of the Program itself.

With these rules in mind, look at the punctuation in the sample program. You should be able to see the logic of the three closing END statements.

* The "END" statement closes the BEGIN statement in the "FOR Subcol" loop.

* The next "END" closes the "FOR Subrow" loop. It requires no punctuation because it precedes the final "END." statement.

4. The output of the program should confirm the distinction between rows and columns in your mind.

5. The ":7 :3" statement in the final Writeln assures that each value will have the necessary space to make the output visually readable in decimal form.

Adding Multi-dimensional ARRAYS

You can add the values located in Arrays by first determining the value in each specific location in the Array, enter or determine the corresponding value in the second Array, and store the result in a third Array of like proportions.

The following program asks the user to enter the elements of the first Array. It then requests the corresponding elements of the second Array and immediately performs the calculations that produce the third. This program could write the first array as well as the final array. Note, however, that it could never print the second Array because it never exists as a complete entity. When the program identifies the values in the first Array, it adds the value the user enters and immediately constructs the third Array.

The program uses the same techniques that you learned in the previous example to create and write an Array. It then returns to each value and requests the number to be added to that value. Finally, it produces the resultant table.

```
PROGRAM AddMatrix(Input,Output);

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

VAR
 Matrix, BigMatrix : MatxType;
 Subrow, Subcol : Integer;
 AddEle : Real;

BEGIN
 FOR Subrow := 1 TO 3 DO
  FOR Subcol := 1 TO 3 DO
       BEGIN
        Write('Matrix1 element ', Subrow: 3, Subcol: 3, ' is: ');
          Readln(Matrix[Subrow,Subcol])
       END; (* Subcol FOR loop *)
 FOR Subrow := 1 TO 3 DO
    FOR Subcol := 1 TO 3 DO
       BEGIN
        Writeln('Matrix2 element ", Subrow: 3, Subcol: 3, ' is: ');
          Readln(AddEle);
        BigMatrix[Subrow,Subcol] := AddEle +
                        Matrix[Subrow,Subcol]
       END; (* Subcol FOR Loop *)
 Writeln;
 Writeln('The Sum of the two matrices is: ');
 Writeln;
 FOR Subrow := 1 TO 3 DO
  BEGIN
       Writeln;
       FOR Subcol := 1 TO 3 DO
        Write(BigMatrix[Subrow, Subcol]: 7: 3)
  END (* Subrow FOR Loops *)
END. (* Main Program *)
```

COMMENTS

1. This program is essentially the same as the previous one. One difference, however, is the use of the Variable "AddEle" which holds the value to be added to the item in the first Array.

2. The Array "BigMatrix" is formed by reading the corresponding value in the Array "Matrix" and adding the user-entered value "AddEle" to it.

3. The use and punctuation of END statements may seem confusing. Every BEGIN requires an END. If a calling statement contains only one command, as in the last "FOR Subcol" command, no BEGIN statement is needed. Consequently, only one END statement is needed to end the loop.

4. Remember that no punctuation is necessary before an END statement--even if the statement is itself an END statement.

COPYING ARRAYS

If you define two ARRAYS that have the same subscript types and the same element types, the values of one Array may be copied into the other with a simple assignment statement.

If the following TYPE and Variables are declared:

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

VAR
  Matrix1, Matrix2: MatxType;
```

Matrix 1 may be copied into Matrix 2 by the statement:

```
Matrix2 := Matrix1
```

Values may be added to a String Array by indicating the index of the String where the element should be placed and the element to be added. The following program copies one string into another, alters the second string, and then prints both.

```
PROGRAM AddStrings(Input,Output);

TYPE
  String = ARRAY[1..10] OF Char;

VAR
  Word1, Word2 : String;

BEGIN
  Word1 := 'Experience';
  Word2 := Word1;
  Word2[7] := 'm';
  Word2[8] := 'e';
  Word2[9] := 'n';
  Word2[10] := 't';
  Writeln(Word1);
  Writeln;
  Writeln(Word2)
END.
```

COMMENTS

The first word is copied into the second, and then the second is altered by using index values to change specific letters. The letters are enclosed in single quotes.

Using ARRAYS in Parameters

Often you want to pass information from an Array in the body of the program to a Function or a Procedure. Simply include the Variable that indicates the Array in the Parameter or Argument List. You can pass elements of the Array or the entire Array itself.

* If you pass an element of an Array, the Argument List in the Function or the Parameter List in the Procedure must indicate the data type of the element being passed.

* If you pass an entire Array, you must indicate the name the subprogram will know the Array by, declare it as a Variable, and indicate its TYPE.

The sample program below illustrates how Array values may be exchanged between the main program and subprograms. It creates a single-dimensioned Array of numbers called "BigArray," and uses the variable "Subscript" to identify items within the Array.

The Program also defines two Procedures. The first, called "Exchg," receives two items from the Array and reverses their order. Note that the procedure's Parameter List contains the Variables A and B. which are declared to be Real numbers. The declaration is similar to Parameter Lists that you have seen before:

```
PROCEDURE Exchg(VAR A,B: Real);
```

The second Procedure, named **SortOrder**, communicates those values to the first procedure by passing two elements of the Array at a time. It uses the variable **NumIndex** to indicate which two elements of the Array to transmit. It does this with the statement:

```
Excg(SubArray[NumbIndex],SubArray[NumbIndex+1]);
```

The second Procedure can transmit items from the Array in the main program because it receives the entire Array in its Parameter List which is:

```
PROCEDURE SortOrder(First,Last: Integer; VAR SubArray
    :NumbArray);
```

The Procedure's Parameter List tells it to expect to receive the first and last items in the sort, as well as the entire Array which it knows as "SubArray." The calling statement in the main program calls the Array, "BigArray." It passes all this information by the statement:

```
SortOrder(First, Last, BigArray);
```

The program allows the user to enter numbers. It then asks the user to indicate the entry number where it should begin the sort and the entry number which ends the sort.

Note: This program is purely for instructional purposes. If you indicate more than 6 items in the list to be sorted, the program takes a long time to run. In addition, when you indicate the first and last entries, use the "Entry Number" as an index to the Array. The actual number may be bigger than the number of items that the Array has indexed. If you enter the actual number, you will confuse the program.

```
PROGRAM ParamArray(Input,Output);

CONST
 MaxNumbs = 150;

TYPE
 NumArray = ARRAY[1..MaxNumbs] OF Real;

VAR
 First, Last, Subscript : Integer;
 BigArray : NumArray;

        PROCEDURE Exchg(VAR A,B: Real);

        VAR
          C : Real;

        BEGIN
          C := A;
          A := B;
          B := C
        END; (* of Exchange Procedure *)

        PROCEDURE SortOrder(First, Last: Integer;
                            VAR SubArray: NumArray);

        VAR
          NumbIndex : Integer;
          Exchanged : Boolean;
```

```
        BEGIN
         REPEAT
          Exchanged = FALSE;
          FOR NumbIndex := First TO (Last-1)DO
                IF SubArray[NumbIndex] > SubArray[NumbIndex+1]
                THEN
                 BEGIN
                    Exchg(SubArray[NumbIndex],
                        SubArray[NumbIndex+1]);
                        Exchanged := TRUE
                 END; (* of IF..THEN loop *)
         UNTIL Exchanged := FALSE
        END; (* of SortOrder Procedure= *)

BEGIN (* MAIN PROGRAM *)
 Writeln;
 Writeln;
 Writeln('Enter a list of numbers to be ordered.');
 Writeln('After each number press RETURN.');
 Writeln('Press 0 and RETURN to end.');
 Subscript := 0;
 REPEAT
  Subscript := Subscript + 1;
  WRITE('Entry Number ', Subscript: 3, ' is :');
    Readln(BigArray[Subscript]);
 UNTIL BigArray[Subscript] = 0.0;

 Writeln('Order this list between which entries?');
 Writeln('Use the Index number, not the value.');
 Write('First : ');
   Readln(First);
 Writeln;
 Write('Second : ');
   Readln(Last);
 SortOrder(First, Last, BigArray);

 Writeln;
 FOR Subscript := First TO Last DO
  Writeln(BigArray[Subscript]:7 :3, ' Entry Number ', Subscript:3)
END.
```

Comments

1. The main program creates an Array of up to 150 elements. It then asks the user to determine the boundaries of the values it will sort. It calls the Procedure "SortOrder" to conduct the sort, passing the entire Array as a parameter.

2. "SortOrder" compares each element in the part of the Array to be sorted to the item that follows it in the "BigArray." If the first item is greater than the second, it calls the Procedure "Exchg" which reverses the items. It continues to do this until the Boolean Variable, "Exchanged," is FALSE.

3. The Boolean Variable, "Exchanged," is originally set to FALSE. If the Procedure "SortOrder" determines that the first item is greater than the second, it calls the Procedure, "Exchg," and sets "Exchanged" to TRUE. This forces the Procedure to continue switching elements of the subarray until all of its elements are less than the succeeding element. When no further reversals are necessary, "Exchanged" is allowed to remain TRUE and the Procedure "SortOrder" ends.

4. Pay special attention to the use of the Subscript variable in the main Program. It enables the program to access individual items in the Array. The NumbIndex variable is used in the Procedure "SortOrder" for the same purpose.

RELATED TOPICS

End of Line

When the user presses the <RETURN> key, the computer reads a value that signals the end of the line. Pascal labels this value, **EOLN**. **EOLN** is a Boolean Variable and it remains FALSE until the **RETURN** key is pressed. It then remains TRUE until additional data is entered with a Read or Readln Statement. Consequently, you can use the **EOLN** value to control input from the keyboard.

The following program illustrates how to use the **EOLN** value to control the input of data. The program will accept four words of up to 15 characters in length. It stores each word in a matrix of Arrays, getting each word one

element at a time until the **RETURN** key is pressed. The **EOLN** value tells the program that the end of a word has been reached. When the **RETURN** key is pressed, the program prints the number of characters entered.

```
PROGRAM GetWord(Input,Output);

TYPE
  WordType  =  ARRAY[1..15] OF Char;
  TableType = ARRAY[1..4] OF WordType;

VAR
  WordIndex, LetterIndex : Integer;
  WordMatrix : TableType;

BEGIN
  Writeln;
  Writeln('Enter 4 words.  End each word');
  Writeln('by pressing the <RETURN> key;.');
  FOR WordIndex := 1 TO 4 DO
    BEGIN
          LetterIndex := 0;
          WHILE NOT EOLN DO
           BEGIN
            LetterIndex := Letter Index + 1;
            Read(WordMatrix[WordIndex,LetterIndex])
           END  (*  Of WHILE  *)
    END  (*  of FOR loop  *)
  Writeln('The preceding word had ', LetterIndex: 3, ' letters.');
     Readln
END. (* of Main Program *)
```

COMMENTS

The program counts each character as it is entered until an End Of Line (EOLN). Words of less than 15 characters are filled with spaces. Any characters over 15 are ignored.

Recursion

Pascal allows you to define a Function or Procedure which calls itself. This is known as **Recursion.** If you use a Recursive subprogram, make certain that it contains a condition that will allow the subprogram to return to the main program.

Recursion is used when:

1. logical decisions occur repetitively, or

2. computing a function requires repeating a series of identical commands, such as

   N! = N*(N-1)*(N-2)*...*(N-(N-1))

The following Procedure, "SortAlpha," is used to sort words in an Array. If the first word is greater than the second, it calls another Procedure, "Exchg," to reverse their order. It then continues to call itself until each element of the ARRAY is less than the subsequent element.

```
PROCEDURE  SortAlpha(VAR WordMatrix : WordArray);

VAR
 WordIndex  :  Integer;

BEGIN
 FOR WordIndex  := 1 to Maxword-1 DO
  IF  (WordMatrix[WordIndex] >
      WordMatrix[WordIndex+1]
  THEN
      BEGIN
      Exchg(WordMatrix,WordIndex);
      SortAlpha(WordMatrix)
      END
END;
```

CONCLUSION

This very long section has introduced a data type, the ARRAY, that allows you to manipulated complex forms of data. It has also shown you how to:

* declare an ARRAY
* declare a multi-dimensional ARRAY
* add ARRAYS
* copy ARRAYS
* pass ARRAYS as parameters
* use the End Of Line value to control input
* use RECURSIVE calls within a subprogram

The next section introduces the concept of a RECORD and shows how to construct ARRAYS of RECORDS.

12. RECORDS

Some units of data are really mixtures of Pascal data types. A date, for example, is a combination of two different data types and three different elements. The date "January 1, 1986" is one String of characters, followed by an integer, followed by a character, and followed by a group of Integers. Pascal allows the programmer to define mixed data types as **RECORDS**.

This section explains:

* Creating and using records
* Accessing records using the WITH statement
* Arrays of records
* Variant Records

DECLARING A RECORD

A **RECORD** is a user-defined data type. Consequently, it must be defined under the TYPE heading in a Pascal program. First, declare an identifier as a RECORD. Then, declare the names of the items in the record and indicate their data types. Conclude the RECORD with the "END;" statement.

The following RECORD, named "DateType," contains three items that comprise a date. After declaring the data type RECORD, a Variable, "DateRec," is assigned to that type.

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

VAR
 DateRec : DateType;
```

Note that the Record Identifier is followed by an equal sign (=) and the declaration of the data type RECORD. Semicolons indicate the end of each

item in the Record -- with the usual exception of the last item that precedes the END statement. The entire declaration ends with a semicolon.

The items in a Record are called "fields." The "DateType" Record contains three fields -- Month, Day, and Year. The general format of a RECORD is:

```
TYPE
 Identifier = RECORD
  Field1 = DataType;
  Field 2 = DataType;
  etc.
 END;
```

A RECORD may define one of its fields as another RECORD. If you do this, however, the Record that is a field in the main Record must already be defined. The following RECORD contains a field that is itself a Record as defined in the previous example.

```
TYPE
 EmployType = RECORD
  LName : ARRAY[1..15] Of Char;
  FName : ARRAY[1..10] Of Char;
  Address: ARRAY[1..20] Of Char;
  City : ARRAY[1..10] Of Char;
  State : ARRAY[1..10] Of Char;
  Birth : DateType
 END;

VAR
 EmployRec : EmployType;
```

The Variable "EmployRec" contains 6 fields. The first 5 are Arrays that hold identifying the employee. The sixth field is a previously defined Record, "DateType." It holds the birthdate of the employee.

Using Records

When a program reads or writes records or fields in record, it must be told the name of the record and the specific field to be addressed. If the program uses a record type like the one defined above as an employee record, the following commands would write the employee's last name and birthday.

```
Writeln(EmployRec.LName);
Writeln(EmployRec.Birth);
```

As you can imagine, addressing multiple fields in a Record requires a great deal of repetitive programming. Pascal uses a special command, **WITH**, which simplifies addressing fields in a Record.

The WITH..DO Statement

The **WITH..DO** statement allows you to indicate a record identifier. Once the record has been identified, the program can locate specific fields in the Record by the field name.

The **WITH** statement can be used with Records or ARRAYS of Records. Fields within a Record that are themselves ARRAYS, can also be addressed using the **WITH** statement.

The following statements illustrate the use of **WITH** to access fields in the record **EmployRec** and write information contained in that record.

```
WITH EmployRec  DO
 BEGIN
  Writeln(LName);
  Writeln(FName);
  Writeln(Birth)
 END;
```

The WITH statement saves a great deal of programming time when you want to access fields within a Record.

Copying Records

If two records are defined as the same type, it is possible to use a simple assignment statement to copy one record into another. For example, after defining the TYPE of RECORD as "DateType," the following lines copy the first record into the second.

```
VAR
 DateRec1, DateRec2, : DateType;

BEGIN
 DateRec2 := DateRec1;
```

A Sample Program

The following program illustrates the use of Records. It calculates the approximate number of days that have elapsed since January 1, 1980.

The program defines a RECORD that consists of the day, the month, and the year. The day and month are defined as subranges. (See Section IV, Part I, Lesson 7 if you forgot what a subrange is.) The Variable "Day" can equal any number from 1 to 31. "Month" can equal any number between zero and twelve.

The program asks the user to enter a date and calculates the elapsed time since January 1, 1980.

```
PROGRAM Elapsed (Input,Output);

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

TYPE
 DateType = RECORD
  Day   : 1..31;
  Month : 0..12;
  Year  : Integer
 END;
```

```
VAR
  B       : Integer;
  DateRec : DateType;
  InMonth  :  ARRAY[1..3]  OF Char;

BEGIN
  Writeln('Enter MONTH -- upper case, first 3 letters.  ');
  Readln(InMonth);
  WITH DateRec DO
    BEGIN
          Write('DAY  =  ');
            Readln(Day);
          Writeln;
          Write('Year  =  ');
            Readln(Year)
    END;
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.
```

COMMENTS

1. The record consists of three groups of integers. "Day" and "Month" are subrange types, i.e., the numbers indicate the subrange of Integers that the value can equal. Year is a regular integer.

2. The WITH statement identifies the record that will be used to save the data. Once the record has been identified, the program can address the individual fields simply by using their names, i.e., "Day," "Month," and "Year."

3 Note that the subrange for the "Month" field contains the value of zero. The program uses this value to check the user's entry in the "Month" field. It originally sets the value of "Month" to zero; it then determines the name of the month entered and assigns the appropriate number to the "Month" field. If the name does not equal one of the valid month abbreviations, the value in the "Month" field remains 0.

4. The final IF test examines the "Month" field. If it determines a 0, it prints the error message. Otherwise, it calculates and prints the elapsed time since the entered date.

5. Note that the Month field is declared to be a subrange of Integers; yet, the user enters it as a String. The IF statements then convert the user-entry to an Integer value. The program could have declared Month as a scalar collection of the names of the months, but then you could not directly compare the user's entry and its Integer value. This is because the String, "JAN" is not equivalent to JAN, the element in a scalar list.

Arrays of Records

A Pascal program can declare Arrays of any type of data, including RECORD data types. To create an ARRAY of RECORDS, define the RECORD TYPE, then define the array Variable as:

Identifier = ARRAY[subscript range] OF RECORD TYPE.

The format of the declaration is:

```
TYPE
 RecIdentifier = RECORD
  Field1 : type;
  Field2 : type;
   ...
 END;

VAR
 ArrIdentifier : ARRAY[subscripts] OF RecIdentifier;
```

Once the array of records has been formed, you may access individual records by indicating the appropriate index after the ARRAY identifier. You may identify fields by adding the field name extension. If the program has declared a Record Type known as DateRec, the following lines create an ARRAY of DateRec and accesses fields within the first and second records.

```
VAR
 List : ARRAY[1..10] OF DateRec;

BEGIN
 Writeln(List[1].Year);
 Writeln(List[2].Year);
```

Since the Array, "List," contains an Index or Subscript value, it cannot be identified by using the **CASE..OF** statement.

A Sample Program

The following program might be used to automate your address book. It allows you to write and read records which contain name and address information. The program consists of two procedures -- WritePages and ViewPages.

In WritePages the user enters the number of pages which will be written to. Data is then entered with a FOR..DO loop keeping track of the array of records.

In ViewPages the user enters the number of pages which he or she wishes to see. The procedure writes the specified pages using a FOR..DO loop for the array of records.

```
PROGRAM AddressBook(Input, Output);
TYPE
        String = ARRAY[1..60] OF Char;
        PageType =  RECORD    (* Declare Page Record Type *)
          Name,
          Address,
          ZIP,
          Phone: String
        END;
VAR
        Page: ARRAY[1..10] OF PageType;
        Start, Ending: Integer;

PROCEDURE WritePages;
VAR Loop : Integer;
BEGIN
 Writeln('Enter the names/addresses to be written to the pages of the book');
 Write('Enter starting page, space, ending page [i.e., 1 10]? ');
  Readln(Start, Ending);
FOR Loop := Start TO Ending DO
 BEGIN
  Writeln;
  Writeln('Page #', Loop);
  Write('Name:  ');
   Readln(Page[Loop].Name);
  Write('Address:  ');
   Readln(Page[Loop].Address);
  Write('ZIP:  ');
   Readln(Page[Loop].ZIP);
  Write('Phone:  ');
   Readln(Page[Loop].Phone)
 END
END;
```

```
PROCEDURE ViewPages;
VAR Loop : Integer;
BEGIN
  Writeln('Look at the names/addresses on pages of the book');
  Write('Enter starting page, space, ending page [i.e., 1 10]? ');
   Readln(Start, Ending);
  Writeln;
  FOR Loop:= Start TO Ending DO
   BEGIN
     Writeln('Page Number ', Loop);
     Writeln(Page[Loop].Name);
     Writeln(Page[Loop].Address);
     Writeln(Page[Loop].ZIP);
     Writeln(Page[Loop].Phone)
   END
END;

BEGIN
  WritePages;
  Writeln;
  ViewPages
END.
```

Variant Records

When you design a program using records, you often find yourself defining several different record types that have most, but not all, fields in common. You could define separate records to handle each situation.

For example, an auto repair shop owner wishes to keep a record of each repair in order to bill his customers. Unfortunately for the bookkeeper, clients may be either individuals or companies. In either case, he needs to know the labor and parts used as well as the invoice number, the customer's name and address. If the client is a company, he needs to know their requisition number. If the client is an individual, he needs to know the clients Social Security numbers. The two records below could handle those conditions.

```
TYPE

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

  Invoice1 = RECORD
    InvoiceNum,
    Labor,
    Parts        : Integer;
    CusName,
    CusAddr      : String;
    ReqNumb      : Integer
  END;

  Invoice2 = RECORD
    InvoiceNum,
    Labor,
    Parts        : Integer;
    CusName,
    CusAddr      : String;
    SocSec       : String
END;
```

NOTE: Similar data types within the RECORD may be listed if they are separated by commas. You don't need to place each item on a single line, but it makes the Record Declaration easier to read.

While there is nothing wrong with declaring as many records as you want, it does become time consuming. Pascal, however, allows you to use the **CASE..OF** statement to include fields that are defined one way in one case and a different way in another.

The following RECORD declaration illustrates how to combine alternative data types within a field. It allow the bookkeeper to combine the two different records into a single data item.

```
TYPE
  Invoice = RECORD
    InvoiceNum,
    Labor,
    Parts : Integer;
```

```
    CusName,
    CusAddr : String;
    CASE Custmr : Integer OF
        1 :(ReqNum : Integer);
        2 :(SocSec : String)
  END;
```

The value after the CASE statement is called a **tag Field**. It can be any simple data type, either a character or an integer. The tag field allows you to indicate the different record fields that can be included in the RECORD. Use CASE..OF statements to access the variant fields when either writing to or reading from those fields.

The following program stores a record that is similar to our sample record. It records the customer's name and address. It then determines if the customer is a company or an individual by asking the user to indicate the tag identifier. If the customer is a company, it requests the requisition number of the order. If the customer is an individual, it requests the customer's social security number. To access the variant fields, it uses the CASE..OF statement.

```
PROGRAM VariantRec(Input,Output);

TYPE
 InvType = RECORD
   CustName,
   CustAddr : ARRAY[1..20] of Char;
   CASE Custmr : Integer OF
     1: (ReqNum : Integer);
     2: (SocSec : ARRAY[1..11] OF Char)
 END;

VAR
 Invoice : InvType;

BEGIN
 WITH Invoice DO
 BEGIN
    Write('Enter Name : ');
     Readln(CustName);
    Writeln;
    Write('Enter Address : ');
```

```
        Readln(CustAddr);
      Writeln;
      Writeln('Enter Customer Type');
      Writeln(' 1.  Company');
      Write(' 2.  Individual  :  ');
       Readln(Custmr);
      CASE Custmr OF
       1  :  BEGIN
                Write('Enter Requisition  Number: ');
                 Readln(ReqNum)
             END;
       2  :  BEGIN
                Write('Enter Social Security Number: ');
                 Readln(SocSec)
             END
      END
  END;
  Writeln('RECORD':  20);
  Writeln('_______':  20);
  Writeln;
  Writeln;
  Writeln;
  Writeln;
 WITH Invoice DO
   BEGIN
      Writeln('Name   :  ', CustName);
      Writeln('Address:  ', CustAddr);
      CASE Custmr OF
        1:  Writeln('Req.  No  :  ', ReqNum);
        2:  Writeln('Soc. Sec. No.  :  ', SocSec)
      END
   END;
  Writeln;
  Writeln;
  Writeln;
  Writeln('End Of Program.')
END.
```

COMMENTS

1. The declaration of the RECORD illustrates another way of formatting the declaration. Since Pascal ignores spaces, you can separate each item in a list by a space or a line. The data type must be indicated at the end of the list which is indicated by a colon (:). The items "CustName" and "CustAddr" could follow each other on a single line; but printing the items on separate lines makes the declaration more readable.

2. "Invoice" is declared to be the user-defined type, "InvType."

3. The program uses a WITH Invoice DO statement to identify the Record. After the BEGIN statement, you need to refer to fields within the Record only by the field name. Close the WITH statement with an END statement.

4. The user enters the variable, "Custmr" which is the tag field. A "1" indicates that the customer is a company and the program requests the requisition number. A "2" indicates that the customer is an individual, so the program requests a Social Security number.

5. CASE..OF statements control the input of information for the variant field. Note that the CASE statement must be terminated by its own END statement. This END statement encloses the END statement that marks the conclusion of the nested BEGIN/END statements which are associated with each variable field. You must use BEGIN/END statements when more than one command line follows the CASE condition.

6. Writeln statements control the format of the output.

7. Another CASE..OF statement controls the information printed to the screen. Again note that the CASE statement requires its own END statement. Since there is only one command line associated with each CASE condition, a BEGIN/END statement is not required for the individual CASE conditions.

8. The program prints output that depends upon the type of record input during the program requests.

Conclusion

You should now be fairly comfortable with declaring and using RECORD data types. Variant Records allow you to perform extremely sophisticated operations on complex data elements. The next section demonstrates how to use another Pascal data type, **SET**, to access and manipulate data.

13. SETS

A SET is a collection of items, called Members. These Members can be integers or groups of characters. The set can contain up to 256 members. The general format for a Set declaration is:

```
TYPE
  Identifier = SET OF base type;
```

The base type can be any scalar type: a list of names, a subrange, etc.

Some typical sets are illustrated below.

```
TYPE
     NumSet = SET OF 1..50;

     Months = (JAN, FEB, MAR, APR, MAY, JUN, JUL,
          AUG, SEP, OCT, NOV, DEC);

   YearSet = SET OF Months;

VAR
     Number : NumSet;
     Calendar : YearSet;
```

The Variable "Number" can contain any numbers from 1 to 50. The Variable "Calendar" can contain any of the designations for the months of the year. The actual contents of the two sets are declared in the body of the program. The original declarations indicate what the set **can** contain. The program defines what they **do** contain.

The following sample program illustrates the declaration and use of a set of numbers ranging from 10 - 25. It uses a new statement, **IN** to determine if a value is included within the Set. (A further discussion of **IN** follows below.)

```
PROGRAM SetDemo(Input,Output);

TYPE
  NumSet  =  SET OF 10..25;

VAR
  Prime,  NotPrime  :  NumSet;
  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 Number.')
  ELSE
    IF N IN NotPrime THEN
      Writeln('That is not a Prime Number.')
    ELSE
      Writeln('That is not between 10 and 25.')
END.
```

COMMENTS

1. The TYPE declaration defines the set type as "NumSet."

2. The Variable declarations define "Prime" and "NotPrime" as "NumSet" types. Each of these variables can contain integers from 10 to 25.

3. The program then defines the 2 sets, "Prime" and "NotPrime." Prime contains the prime numbers between 10 and 25. NotPrime contains the others.

4. Values assigned to the set within the program are contained within brackets [] and separated by commas (,).

5. The IN statement determines if the value entered by the user is "in" either of the defined sets.

6. Nested IF loops determine which type of number is entered by the user. The second Else statement declares an invalid entry if the user enters a number that is not between 10 and 25.

Sets, Arrays, and Scalar Variables

A Set is actually a collection of data, much like an Array. The difference is that the items in an Array are identified by their position in the Array. Remember that you can indicate the elements of an Array by using an Index value to identify the item. You can't do that with Sets.

A Set is also similar to a Scalar Type in that it is not a list of elements. The difference, however, is that you cannot use subrange declarations to indicate parts of the set as you can for a Scalar variable.

A Set, unlike Arrays and Scalar Variables, can hold, at any point in the program, several values.

A Set can be manipulated by using the IN statement and set operators. (See below.)

The IN Statement

The IN statement allows the programmer to determine whether a value is included in the defined set. You used it in the first sample program. The general format for an IN statement is

```
BEGIN
 IF variable IN setname THEN
  BEGIN sequence of commands
        Command1;
        Command2;
          ...
  END;
```

IN determines whether the variable is actually included in the set name indicated. You can use a TRUE condition to define one set of actions or a FALSE condition to determine another.

Operations Using Sets

There are three operations that can be performed on Sets. They are indicated by the + (or Union), * (or Intersection), and - (or Difference) symbols.

Union	returns the total list of elements contained in the sets.
Intersection	returns the values that the two sets have in common.
Difference	returns the elements that are not shared by the sets.

In addition to the three set operators which manipulate sets, there are seven relational operators that allow you to compare sets. These operators produce either TRUE or FALSE Boolean Values that are exactly parallel to the arithmetic operators that have already been discussed.

Equality	Set1 = Set2
Inequality	Set1 <> Set2
Subset	Set1 <= Set2
Superset	Set1 => Set2
Member IN	Set1 IN Set2

Note that the "Set IN" operator returns the Boolean Value of TRUE only if Set1 is a member of Set2.

Using Sets

It is not necessary to declare a Set before you use it in a program. You can simply declare the Set within the program by including the elements within brackets. The following program declares the Set of elements [F,NP]. The contents of the Set, however, must be identified in a Scalar list before the body of the program. The Set has no identifier; it is simply defined in the program.

The program requests grades for each student in a class. If the grade is F or NP, the screen displays an admonition; otherwise, it displays congratulations.

```
PROGRAM  Finals (Input,Output);

CONST
  ClassSize = 30;

TYPE
  GradeType = (A,B,C,D,F,P,NP,I);

  StuGrade = RECORD
   StudentID : Integer;
   Grade    : GradeType
  END;              (* of Record declaration *)

VAR
  ClassGrade : ARRAY[1..ClassSize] OF StuGrade;
  N : Integer;
  LetterGrade : ARRAY[1..2] OF Char;

BEGIN
  FOR N := 1 TO ClassSize DO
   BEGIN
    Write('Input student ID :');
      Readln(ClassGrade[N].StudentID);
    Writeln;
    Write('Input grade : ');
      Readln(LetterGrade);
    IF LetterGrade = 'F ' THEN
     ClassGrade[N].Grade := F;
    IF LetterGrade = 'NP' THEN
     ClassGrade[N].Grade := NP;
    IF ClassGrade[N].Grade IN [F, NP] THEN
     Writeln('Too Bad.  Try Again!')
     ELSE
     Writeln('Way to Go!')
  END (* of N FOR loop*)
END. (* of main program *)
```

COMMENTS

1. The GradeType is a Scalar list of elements.

2. The Student Record contains two fields: the Student ID and the actual grade.

3. ClassGrades is an Array of Student Records.

4. The variable N identifies each element in the Array of records.

5. The IF statements assign a value to the Grade field. The value depends upon the letter grade entered by the user.

6. The grade entered is compared to the SET of characters [F, NP]. If the grade is in that SET, the message "Too Bad" is printed on the screen. Otherwise, the message "Good" is displayed.

SETS AND ARRAYS

Sets are often used to examine the members in an ARRAY. If a specific item in the ARRAY is included in the set, one action can be taken. If the item in the ARRAY is not in the set, another action can be indicated.

This program uses the IN statement to compare each grade to the set of Failed (F), Not Passed (NP), or Incomplete (I) grade types. Finally, it totals and prints the number of items that fall into any of these categories.

```
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;
   Gr: GradeSet;

BEGIN
   I := 0;
   Gr := [F,NP,I];
   FOR N := 1 TO ClassSize DO FOR M := 1 TO 25 DO
   IF ClassGrade[N].Grades[M] IN Gr THEN
   I := I + 1;
   Writeln('In this class ', I:3, 'tests were ');
   Writeln('either failed, not passed or incomplete')
END.
```

COMMENTS

1. The program sets up an array of 25 test scores for each student in a class of 30.

2. The complete list will contain 30 sets of grades containing 25 scores.

3. Subscripts identify the current record. They are also used to determine the position within each array of grades. "N" indicates the number of the Record; "M" indicates the individual grade.

4. "I" keeps track of the number of test grades that fall within the SET of unacceptable scores. If a grade is "F," "NP," or "I," the count of unacceptable grades is increased by 1. The total is then printed to the screen.

CONCLUSION

This section has covered most of the important information that you need to know in order to use SETS. Obviously, this topic is very large in scope; if you want to learn more, consult a book that contains an extensive description of all the commands available in the Pascal programming language.

The next section introduces the concept of FILES. A file enables your program to save and retrieve information on a storage device. This greatly increases the amount of data that your Pascal program can process.

14. FILES

Files allow you to redirect input and output to a storage device -- usually a disk. This section demonstrates how to:

* WRITE files to a disk
* READ files from a disk
* Create files of records
* Manipulate Random Access Files
* Create TEXT files

GENERAL COMMENTS ON FILES

As you have probably already realized, all this power to manipulate data isn't worth much if you can't store the information in a disk file. Pascal uses files to control the input and output of data. The statement after the Program Name contains the declarations "Input" and "Output" to indicate the device that reads the information and the device that prints it. Ordinarily, the input device is the keyboard and the output device is the monitor screen. A **Read** or **Readln** statement calls for input from the keyboard. A **Write** or **Writeln** statement prints information to the screen.

You can use file designations to redirect input and output to a disk file. You simply have to declare a file and its data type. You can then write to or read from the file. The only disadvantage of using disk files is that it slows the program down. It takes a great deal of time to access, read, and write information from a disk file.

If you use disk files, remember that they save data in a strictly sequential format. The first piece of data entered is the first saved. Consequently, the first piece of data read by a program is the first piece of data that was entered.

Declaring a File

When you want to use a disk file for input or output, you must declare it in the PROGRAM statement after the "Input" and "Output" declarations. You can use any name at this point since you will equate the actual file with its disk identifier in the body of the program. The following PROGRAM declaration identifies a program that will get input from the keyboard, print it to the screen, and record it on a disk.

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

You can use any file identifier to replace the term "F." Just remember that it must be defined in the body of the program as indicated below.

The program declaration tells the computer that another file is available for input and output other than the usual Input and Output files.

To enable the program to use the data stored in the disk file, you must indicate that there is a Variable that contains the information. The following PROGRAM and VARIABLE declarations enable the program to access a file, **F**.

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

VAR
  F : FILE OF Integer;
```

In this case, all the elements of the file **F** are integers. A FILE may also contain Characters, Real numbers, Arrays, Sets, and Records.

Writing to a File

To store data in a disk file, you must first open the file. You do this by using the **Rewrite** statement. **Rewrite** tells the computer to redirect the output to another file. The **Rewrite** command takes two parameters. The first names the identifier of the file it will write to. The second defines the device and the filename it will be saved under.

Note: **Rewrite** also clears the disk file of any existing data.

A typical **Rewrite** statement requires that you enter the identifier **name** of the file as well as the device and filename it is stored on. They are often not the same and, to avoid confusion, it is a good idea not to make them the same. The following declaration indicates the program's identifying name for the file variable and then the disk device and filename.

```
Rewrite(F, 'D1:LST');
```

This statement tells the computer to prepare a user's data file, **F**, which it will write to a file named **LST** on the disk device **D1:**. To put the data into the disk file, the program must contain two more statements. The first stores the data in a file buffer area; the second writes the buffer contents to the disk file.

```
F^ := DataItem;
Put(F);
```

The caret or ^ symbol is executed by pressing <**SHIFT**> and the * key.

The variable **F^** is actually a file buffer variable. The variable **DataItem** refers to the data element that will be placed in this buffer. The **DataItem** can be any type of information you want to use: an integer, a string, or even an entire record. Before the value of an element can be put into a file, the program must assign a temporary file buffer variable which holds the data until it is written to the disk.

The Put(F) statement writes the contents of the buffer to the device:filename indicated in the Rewrite statement. NOTE: Make certain that the device:filename exists in the system. If the program runs and cannot locate the it, the system will crash.

Remember that when writing to the disk, the first element is stored in the first position, the second element in the second position, and so forth.

The only memory space reserved for file variables is for the file buffer variable. This is because the file itself exists outside the memory space of the computer which is merely transferring the data to the disk file. (If the file is a FILE OF Integer, the file buffer is assigned two bytes of memory to accommodate large integer values. Integers use more memory than real numbers or other types of data. The size of the buffer, however, has no direct bearing on the program or how the programmer writes it.)

In summary, the steps for writing a file are:

1. Declare that the program uses an external (disk) file as well as the standard Input and Output files.
2. Define the filetype in the Variable list as a FILE OF some data type.
3. Declare an Identifier as the filetype defined in the Variable list.
4. Open the file with the Rewrite statement which equates the file identifier with a filename.
5. Load the data into the file buffer, which is indicated by the ^ symbol after the file identifier.
6. Write the data to the disk file with the Put(FileIdentifier) statement.

Reading a File

Reading a file is similar to writing one. First, the file must be opened for reading. The command is:

```
Reset(F,'D1:Filename')
```

This equates the filename with a variable that the program uses to identify the file.

Before the file can be read from the disk, the computer must be told to reserve memory space for the read buffer. Like the write buffer, the read buffer holds the data until the program is ready to use it. As you might have assumed, a read buffer has exactly the opposite format of the write buffer. The statement is.

```
DataItem := F^;
```

After you declare the buffer, use the Get command to retrieve the data on the disk file. The full statement is:

```
Get(F);
```

The complete sequence of commands is:

```
DataItem := F^;
Get(F);
```

The following diagram illustrates the commands used to write and read files. The related commands are placed side by side.

Comparison of Write and Read

WRITE	READ
Rewrite(FileIdentifier, Device:Filename)	Reset(FileIdentifier, Device:Filename)
FileIdentifier^ := DataItem	DataItem := FileIdentifier^
Put(FileIdentifier)	Get(FileIdentifier)

Once you know how to retrieve information from a disk file, you have to be able to tell the program when to stop reading the disk. Pascal uses an End Of File marker to tell the computer when the file ends.

The END OF FILE Marker

The disk uses a specific value to indicate the end of a file. Pascal recognizes that value as **EOF**. EOF is a standard Boolean function that becomes true when the end of a file is reached.

When you are reading from a file, you often don't know how many items the program should **Get**. Consequently, you should use the EOF value as a test to determine whether the program should continue reading data from the file. The full sequence of instructions for reading to the end of file is:

```
Reset(F,'D1:Filename')
WHILE NOT EOF (F)  DO
 BEGIN
   DataItem :=  F^;
```

```
        Writeln(DataItem);
        Get(F)
      END;
```

The following program illustrates how to write to a disk file and then read that information back into the program. It asks the user to enter 10 numbers which it stores in the file named, "LIST." The program then reads the file back into its memory and prints the output on the screen.

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

VAR
 List1 : FILE OF Integer;
 Count : Integer;
 J : Integer;

BEGIN
 Rewrite(List1, 'D1:LIST');
 FOR Count := 1 to 10 DO
  BEGIN
   Write('Enter a number: ');
   Readln(J);
   List1^ := J;
   Put(List1)
  END;  (* FOR loop *)
 Writeln;
 Writeln('Here is the output from the disk.');
 Reset(List1, 'D1:LIST');
 WHILE NOT EOF(List1) DO
 BEGIN
   J := List1^;
   Write(J: 5);
   Get(List1)
 END  (* WHILE loop *)
END.
```

COMMENTS

1. The program declaration contains a third file, **List1**. This tells the program that information will be used from a source other than the normal Input/Output files.

2. The file identifier is declared as a variable, **List1**. Since Integer is a predefined data type, the program does not need to define the type. **Count** regulates the number of FOR loops that control the user's entries. **J** holds the value the user enters during each loop.

3. The program opens the disk file for writing with the **Rewrite** statement. The first item in the parentheses is the file identifier, i.e., the name the program uses to the identify the file. The second item is the device:filename which actually stores the data.

4. A simple FOR loop allows the user to enter the 10 numbers.

5. Readln(J) assigns the number entered to the variable J. The value of **J** is then assigned to the file buffer, **List1^**. Once the data is in the buffer, the number is written to the disk file with the **Put** statement. Note that you use the file identifier to indicate what the buffer should write to the file. You read and write files, the buffer merely holds the data in the process.

6. Once the file is stored on the disk, the program reads it back and prints it to the screen.

7. Reset opens the file for reading. In effect, it forces the position indicator back to the beginning of the file. It also reidentifies the disk with the program's file identifier.

8. The WHILE loop tests for the End Of File marker. If it does not register the EOF marker, it retrieves the value stored in the next disk position and puts it in the file-identifier buffer. That value is then transferred to the variable **J**. After writing **J** (allocating 5 spaces for the number on the monitor), it executes the **Get** statement to retrieve the next value. This process continues until the program senses the EOF marker (i.e., the Boolean EOF condition becomes true).

9. Note that the control loop regulates how much data the user can enter; the EOF marker regulates how much data is read back into the system.

FILES OF RECORDS

Most often, you use disk files to store files of records. The procedure for creating such files is almost identical to the sample program described in the previous section. The only difference is that you must define the record before you declare the filename variable as a FILE OF the record type you have defined. A beginning Pascal programmer can find all this naming confusing, but it is strictly logical.

If you keep the following definition in mind, the discussion of files of records should be easier to comprehend.

An IDENTIFIER is the name the program uses to label the data element--whatever that type of element is. Furthermore, the Variable identifier of a user-defined data-type must be defined as the type indicated by the Identifier of that type.

With this principle in mind, the following procedure for declaring a file of records should not be too confusing. When you want to write a file of records:

1. Declare the file identifier in the Program declaration.

```
PROGRAM RecDemo (Input,Output,ClassFile);
```

2. Define the structure of the record under the TYPE heading.

```
TYPE
  RecType = RECORD
    Name : ARRAY[1..15] OF Char;
    Grade : Integer
END;
```

3. Declare another identifier under the TYPE heading to represent a FILE OF the record-type identifier.

```
TYPE
  StudentFile = FILE OF RecType;
```

4. In the list of variables, declare the file's identifier as a type indicated by the type-identifier.

```
VAR
  ClassFile : StudentFile
```

You can now refer to **ClassFile** in the body of your program. You can also give the record an identifier. You could declare **StudentRec** as the identifier of each record. This allows you to access individual fields in the record by the usual statements, "StudentRec.Name" and "StudentRec.Grade".

To write the records to the file, put the individual record into the file-identifier's buffer and then write the buffer to the disk file. To write an entire record to the disk, use the following statements.

```
ClassFile^ := StudentRec;
Put(ClassFile);
```

To read stored records, simply indicate the file-identifier's buffer. For example, to write the student's name in each record, you could use the following statement:

```
Writeln(ClassFile^.Name);
```

Note that the read statement allows you to include the buffer as part of the variable. This saves one step. You could, however, also read the value into a variable and then write the variable. But you don't have to.

The following program uses these principles to write a file of student names and their grades. It continues to request names and grades until you tell it to "END." A WHILE loop controls the input. A nested IF loop stops the input as soon as it senses an "END" indicator. Without the IF loop exit from the sequence, you would still have to enter a grade before the controlling WHILE loop realized that "END" had been entered. This would add an extra record that assigned END a grade.

```
PROGRAM  RecDemo(Input,Output,ClassFile);

TYPE
  RecType = RECORD
    Name : ARRAY[1..15] OF Char;
    Grade : Integer
  END;

  StudentFile = FILE OF RecType;

VAR
  ClassFile : StudentFile;
  StudentRec: RecType;

BEGIN
  Rewrite(ClassFile, 'D1:Class');
  Writeln('Building A File');
  Writeln('_________________');
  Writeln;
  Writeln('Enter "END" to stop the list.');
  Writeln;
  WHILE StudentRec.Name <> 'END          ' DO
   BEGIN
     WITH StudentRec DO
      BEGIN
           Writeln('Enter Name : ');
             Readln(Name);
             IF Name <> 'END          ' THEN
              BEGIN
                   Writeln('Enter Grade : ');
                     Readln(Grade);
                   ClassFile^:= StudentRec;
                   Put(ClassFile)
              END (* of IF *)
     END (* of WITH *)
   END;  (* of WHILE *)
 Writeln;
 Writeln('Record Entry Complete.');

 Reset(ClassFile, 'D1:Class');
 WHILE NOT EOF (ClassFile) DO
   BEGIN
```

```
        Writeln;
        Writeln;
        Writeln('Name  : ', ClassFile^.Name);
        Writeln;
        Writeln('Grade : ', ClassFile^.Grade: 3);
        Get(ClassFile)
      END     (* of WHILE *)
END.
```

COMMENTS

1. The program opens the file, "ClassFile" and identifies the file that the file identifier refers to.

2. The WHILE loop continues to execute until "END" is entered at the Name request. Note that the "END" constant must contain exactly 15 spaces. Since this is the declared length of the Name field in the Record definition, any other number of spaces will prevent the program from compiling.

3. The WITH statement identifies the record name, **StudentRec**, that the rest of the loop uses.

4. The variable, **Name**, contains the student name entered by the user.

5. The IF loop determines the proper action to take. If the Name field equals anything but "END," the program requests the grade. If the name field does equal "END," the program exits the IF loop and terminates the entry sequence.

6. After reading the grade, the program loads the file buffer, **ClassFile^**, with the entire student record. It then writes the record to the disk with the Put statement.

7. Once all the names and grades have been entered, the program Resets the disk file, again equating the file identifier, **ClassFile**, with the device:filename.

8. Finally, using the Writeln statements to format the output, the program enters the WHILE loop which continues to execute until it reads the End Of File marker.

9. For each record in ClassFile, the program loads the fields into the ClassFile buffer and prints the information. Note that the data is loaded into the buffer and printed directly to the screen. The format of the statement is:

```
Writeln(Filename^.Fieldname);
```

10. The Get(Filename) statement forces the program to continue checking records until it senses the EOF marker.

READING AND MANIPULATING DATA FILES

As you may have realized if you tried writing your own file storage programs, writing to a file is destructive. In other words, the Rewrite statement that opens the disk file clears the file of any pre-existing data. Actually, it just resets the position that the program writes to at the beginning of the file. Consequently, when you begin writing, you over-write the existing data.

Because of the destructive nature of writing to a file, it is usual to write a separate program that reads the file you have written. This program can also manipulate any of the information without destroying the original file.

The following program illustrates how to read data from an existing data file and then manipulate the data. With only a few differences, it is identical to the second part of the RecDemo program. It reads the file, ClassFile, and prints the data on the screen. In addition, it flags any grade less than 65 and prints the message "Failure." Finally, after keeping a running total of the grades and the number of grades entered, it computes and displays the class average.

```
PROGRAM  RecRead(Input,Output,ClassFile);

TYPE
 RecType = RECORD
     Name : ARRAY [1..15] OF Char;
     Grade: Integer
 END;
 StudentFile = FILE OF RecType;
VAR
 ClassFile : StudentFile;
 StudentRec: RecType;
 Count : Integer;
 Average : Real;

BEGIN
 Count := 0;
 Average := 0.0;
 Writeln;
 Writeln('Reading the file': 25);
 Writeln('_____': 25);
 Reset(ClassFile, 'D1:CLASS');
 WHILE NOT EOF (ClassFile) DO
  BEGIN
     Writeln;
     Writeln;
     Writeln('Name   : ', ClassFile^.Name);
     Writeln;
     Writeln('Grade  : ', ClassFile^.Grade: 3);
     IF ClassFile^.Grade < 65 THEN
      Writeln('FAILURE');
     Count := Count+1;
     Average := Average + ClassFile^.Grade;
     GET (ClassFile)
  END; (* of WHILE loop *)
 Writeln;
 Writeln;
 Average := Average/Count;
 Writeln('The class average is: ', Average: 3: 2)
END.
```

COMMENTS

1. The record and file declarations must be identical to those that declared the disk file.

2. **Count** keeps track of the how many records are read.

3. **Average** holds the total value of the individual grades. It is divided by **Count** to calculate the class average.

4. Note that the grade is declared to be an Integer. The **Average**, however, must be a Real number since the sum of the integers is divided by a number that may result in a fraction.

5. The program reads each record in the file and prints the contents of the ClassFile buffer, **ClassFile^**, as determined by the field name.

RANDOM ACCESS FILES

The files discussed so far are known as Sequential Files. Each piece of data is recorded and read in the order it was entered. As you saw, this means that every time you run a program that writes records, it erases any information that already exists in the file. It also means that if you want to read the sixth record in a file, the program must first read records 1 through 5.

Kyan Pascal includes a non-standard statement that allows you to access any record in the file directly. Such files are called **Random Access Files.** The **Seek** statement allows you to create and read Random Access Files.

SEEK

When records are saved to a disk, the disk keeps track of where each record begins and ends. It identifies the records by Record0, Record1, and so forth, until the End Of File. The SEEK command allows the progam to read those record delimiters without having to read the entire record. This allows the program to scan the record numbers and identify the exact record you want to

read. The format of the SEEK command is:

Seek(F,N);

where **F** is the file identifier and **N** is the number of the element in the file

The Seek statement should be followed by a command that writes data to or reads data from the file. After Seeking the element **N** in the file-identifier **F**, **Put** or **Get** the file identifier.

Put(F)	writes the contents of the file buffer, which has already been loaded with the data by the Seek Command, to the disk file at the position indicated by the Seek command.
Get(F)	reads the contents of the record located at the position indicated by the Seek statement into the file buffer.

The following program, **SeekDemo**, illustrates how to use the Seek command to either write or read a specific record in a file. It declares a FILE OF Strings, requests the record number of the string to be entered, and then asks the user to input the string. Note that this program allows you to change existing records or append records to an already existing file. You access the desired record by indicating its record number. Obviously, you can not read a record that has not yet been written.

The body of the program uses the **Reset** command to open the file. If you **Reset** a file, you reset the record counter to zero. You can then either read or write additional records. If the program had used a **Rewrite** statement to open the file, the contents of the file would have been over-written and lost. Consequently, use **Rewrite** only when you enter data for the first time. This use of **Rewrite** insures that no unwanted data remains in the file if it has been used by previous programs.

```
PROGRAM  SeekDemo(Input,Output,StrFile);

TYPE
  StringType = ARRAY[1..30] OF Char;

VAR
  StrFile : FILE OF StringType;
```

```
    C :  Char;

  PROCEDURE RdRec;  (* Read a Record *)
  VAR
   i : Integer;
  BEGIN
   Write('Enter Record Number : ');
      Readln(i);
   Seek(StrFile,i);
   Get(StrFile);
   IF NOT EOF(StrFile) THEN  (*EOF true if element empty*)
    Writeln(StrFile^)
  END; (* of RdRec PROCEDURE *)

  PROCEDURE  WrRec;  (*Write a Record*)
  VAR
   i : Integer;
  BEGIN
   Write('Enter Record Number:  ');
      Readln(i);
   Writeln;
   Writeln('Enter the string of data : ');
     Readln(StrFile^);  (*Assign data to file buffer*)
   Seek(StrFile, i);
   Put(StrFile)
  END;  (* of WrRec procedure *)

  BEGIN
   Reset(StrFile, 'D1:Class');
   REPEAT
    Writeln;
    Writeln;
    Writeln('Enter your selection ': 25);
    Writeln('R-Read  W-Write  Q-Quit': 24);
       Readln(C);
    IF C = 'R' THEN
     RdRec;
    IF C = 'W' THEN
     WrRec;
   UNTIL C = 'Q'
  END.
```

COMMENTS

1. The program declares a disk file which it knows as **StrFile**. The disk file is identified later in the program.

2. The program includes two procedures. The first handles reading the file; the second, writing to the file.

3. After declaring the String TYPE, the file identifier is declared in the Variable list.

4. The variable C holds the selection of reading, writing, or quitting.

5. The main body of the program asks the user to select the desired procedure. Two IF conditions, nested inside the REPEAT..UNTIL loop, enable the user to continue writing and reading records until the Quit command is entered. Remember that a REPEAT..UNTIL loop continues to execute until a final condition exists. In this program, the condition is that C equals **Q**.

6. If C equals **R**, the program executes the read record procedure. RdRec requests the record number and stores it in the variable **i**. It then locates the position **i** in the file **StrFile**. Next, it Gets the record indicated. If the position does not indicate the End Of File marker, the program puts the data into the file buffer and then writes the buffer to the screen. After writing the record, it returns control to the REPEAT..UNTIL loop in the main program.

7. If C equals **W**, the program executes the write record procedure. **WrRec** requests the number of the record to be written. (Remember that the first record is always 0.) After reading the record number to be written, it requests the string input. The string the user enters is immediately stored in the file buffer by the **ReadIn(StrFile^)** statement. The program then **Seeks** position **i** on the disk file and Puts the string on the disk.

8. One final note: Remember that the file buffer is always indicated by the carat (^) symbol. When reading a file, you write the file buffer to the screen with a **WriteIn(FileName^)** command. When writing a file, you read the data from the keyboard into the buffer and then Put the file on the disk with a **ReadIn(FileName^)** command followed by a **Seek** and then a **Put(FileName)** statement.

TEXT FILES

Because files often consist entirely of text, Pascal has a standard type of file called Text. It is predefined as *Text = FILE OF Char*

To use a Text FILE, include the filename in the program declaration and then declare its data type in the Variable list as Text. The following statements create a text file named Word.

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

VAR
 Word : Text;
```

The advantage of using Text files is that the input and output commands are simpler than those used for other types of files. After declaring the file's identifier and associating it with a device:filename with the statement

```
Rewrite(FileIdentifier, 'D1:FileName');
```

you can write to the disk file with the command

```
Write(FileIdentifier, TextIdentifier);
```

For example, to write a string of text that has been identified as **Comments** to a file named **Word**, use the statement

```
Write(Word, Comments);
```

This simple command replaces the two statements which ordinarily put the contents of the string into the buffer and then write the buffer to the disk file:

```
Word^ = Comments;
Put(Word);
```

Just as it is easier to write text files to a disk file, it is also easier to read them. The read command is

```
Read(FileIdentifier, TextIdentifier);
```

To read a file named **Word** into a text variable named **Comments**, use the statement

```
Read(Word, Comments);
```

This single command replaces the two commands that put the file contents into the buffer and then print the buffer to the screen:

```
Comment = Word^;
Get(Word);
```

The following program lets the user enter a string of 100 characters and saves the data in a disk file. The structure of the program is obvious; it declares the text file and the string that holds the user's input. It then opens the file for writing, requests and reads the input, and then writes it to the disk file. To show what it has written, it reopens the disk file for reading, reads the file, and then displays it on the screen.

```
PROGRAM  WordProc(Input,Output,Word);

TYPE
  StringType = ARRAY[1..100] OF Char;

VAR
  Word : Text;
  Comments : StringType;
BEGIN
  Writeln;
  Writeln;
  Rewrite(Word, 'D1:Word1');
  Writeln('Enter text :');
  Writeln;
     Readln(Comments);
  Writeln('Saving the file');
  Write(Word,Comments);
  Writeln;
  Writeln('Now reading the file');
  Reset(Word, 'D1:Word1');
  Read(Word, Comments);
  Writeln(Comments)
END.
```

COMMENTS

1. Note that the **Writeln** statement still writes to the screen and that **Readln** gets data from the keyboard.

2. The **Write(Word, Comments)** statement writes the text, identified by **Comments** to the disk file, identified in the program as **Word**.

3. The **Read(Word, Comments)** statement gets the data from the disk file identified as **Word** and loads it into the string **Comments**.

4. A simple **Writeln(Comments)** statement prints the text on the screen.

CONCLUSION

This section provides only an introduction to the use of files in a Pascal program. It has illustrated how to write and read a few types of files. For more detailed explanations and illustrations of the use of files, consult a Pascal textbook. It should explain different techniques for updating existing files. Just keep in mind the unique capabilities of Kyan Pascal when you read the text.

The next chapter explains how to use Pointers to access locations in memory. Pointers let you Peek and Poke information into the computer's memory locations; they also allow you to create lists that can keep track of the location of each element in the list.

15. POINTERS

A Pointer is a variable that points to a location in memory. Because of this special property, pointers can be used to create larger and more flexible data representations (data-structures) than have previously been discussed.

This chapter will explain the concept of pointers and how they are used in the memory of the computer. This chapter will also show how to use pointers to:

* Read values directly from memory locations
* Write values directly to memory locations
* Create new pointers using the NEW command
* Create a linked lists to form a data base
* Clear Pointers using the Dispose command

POINTERS AND MEMORY -- AN EXPLANATION

Normally, the computer keeps track of where it stores information, so that you don't have to. For example, if you write the lines of code

```
VAR
    Count : Integer;

BEGIN
    Count := 54;
```

the computer sets aside a place in memory named, Count, and stores the number 54 in that location. With these lines, we told the computer: "Find an unused portion of memory that will hold an integer, then give that portion of memory the name Count, then put the value 54 in the area named Count."

In this example, we could say that the variable name Count is a pointer, since it points to an area of memory. The reason we dont't call a variable name a pointer is two-fold; it points to only one location in memory, and the variable's contents are a value rather than an address of a location in memory.

So what's an address of memory location? Good question. Think of the memory locations of the computer as a long line of boxes. Each box is identified by a number, and the boxes are numbered so that the next box has a number that is one higher than the last box. Then the address is the number of the box. For example, if one box was labeled 10001, then the next box would be labeled 10002, and one after that would be 10003. In this example, the addresses are 10001, 10002 and 10003. In summary, an address is the identifying number of a memory location (box).

So, instead of holding a value, pointers hold the address of a memory location. Because pointers hold addresses, we have some control over the area of memory which pointers will point to.

To illustrate the concept of pointers, let's create one. We know that we can define a pointer which points the computer to a location in memory. But, to be useful, we must tell the computer what our pointer is pointing at. We can have our pointer point to an integer, a character, a record field, or any other defined data type. For this example, lets have our pointer point to an integer. To declare our pointer to the computer, we would say

```
VAR
    Ourpointer : ^Integer;
```

By doing this, we are assured that no matter what address we give Ourpointer, the contents of the memory locations at that address are interpreted as an integer.

Please note that Ourpointer actually looks at two memory locations. This is because we told the computer that we are pointing to an integer, and all integers are two memory locations (bytes) long. Therefore, if the computer is to make sense of what it sees in the memory location pointed to by Ourpointer, it must consider both bytes. Expanding on this, any time a pointer points to a data-type which is more than one byte in length, the computer must look at the pointer location plus all the following memory locations which are required to store an item of that particular data type. (Note: Please refer to Chapter V, Page 11, for more information on the storage requirements for different data types).

In our example, Ourpointer points to an integer. If Ourpointer has the address 10001, then the computer will look at locations 10001 and 10002 (two bytes). If Ourpointer were pointing to a record which was ten bytes long at location 10001, then the computer would look at locations 10001 through 10010. If

Ourpointer were pointing to a character at location 10001, then the computer would look at only location 10001 (i.e., character data types are only one byte long). You may wish to look at the figures below.

Var Ourpointer : ^Integer (Two bytes long)

Ourpointer ===>	10001:	The computer looks at two memory locations
	10002:	

Var Ourpointer2 : ^Record (Four bytes long).

Ourpointer ===>	10001:	
	10002:	The computer looks at four memory locations.
	10003:	
	10004:	

Var Ourpointer3 : ^Character (One byte long).

Ourpointer ===>	10001:	The computer looks at only one location.

POINTER

Now you know what pointers do and how they work. In this section we will explain how to use pointers, and why they are particularly useful in reading from or writing to specific memory locations.

Standard Pascal contains no provisions for allowing a programmer to create a pointer which points to a specific memory location. However, Kyan Pascal contains an extension to the standard Pascal syntax which allows you to do this. This extension is called POINTER and it is used in the following syntax.

```
PointerIdentifier := POINTER(MemoryLocation);
```

All memory locations must be given in their decimal representations. Once a pointer is given the address of a memory location, that location can be read from or written to.

Using Ourpointer from last section, here is what Ourpointer looks like after it is first declared:

Ourpointer ===> ????

Ourpointer points nowhere in particular because it has not yet been given any value. But, after we say:

Ourpointer := POINTER(300);

Ourpointer looks like this:

Ourpointer ===> 300

Now that Ourpointer points to a specific location in the computer's memory, we can read or write to that location.

Reading from Memory Locations

In the example below, we declare the pointer to be pointing to an integer. Next, using the POINTER command, we point the pointer at location 10001. Then, using the WRITE command, we display the contents of the memory locations 10001 and 10002 in integer form (don't forget integers are two bytes long).

```
Program OurExample1 (output);

    VAR
        Locate : ^Integer;

    BEGIN
        LOCATE := POINTER(10001) ;
        WRITE (Locate^)
    END.
```

If you are familar with the PEEK command used in BASIC, then you will recognize the similarity between this program and the PEEK command. Both read the contents of specific memory locations.

Writing to Memory Locations

The following program declares Locate to be a pointer to an integer. Using the POINTER command, Locate is once again given the address 10001. But, unlike the last example, the statement

```
Locate^ := 65;
```

changes the contents of locations 10001 and 10002 instead of reading them.

```
Program OurExample2;

    VAR
        Locate : ^Integer;

    BEGIN
        Locate := POINTER(10001) ;
            Locate^  := 65
    END.
```

This program performs the same function.

Compare this program with OurExample! and the way the Locate^ variable is used (notice the caret at the end of the pointer name). Locate^ is the way the computer represents the contents of the memory locations pointed to by Locate. Thus if in the WRITE statement in the first example was replaced by

```
Count  := Locate^;
WRITE (Count) ;
```

not only would the contents of locations 10001 and 10002 have been displayed, a copy of those contents would have been assigned to the variable Count (assuming of course, we also declared Count as a variable).

Using Special Memory Locations

In the Atari location 710 holds the number that corresponds to a particular color of the screen background. Using the predefined Pointer function, you can change the color of the background to suit your needs. The following program cycles through 255 background colors and then restores the background to the original color.

```
PROGRAM Change_Color;
        (*Cycle through screen background colors *)
VAR
          Color,Orig: ^Integer;

PROCEDURE Cycle;
VAR
          Loop: Integer;
BEGIN
   FOR Loop := 0 TO 255 DO
          Color^ := Loop
END;

BEGIN
   Color := Pointer(710);
   Orig^ := Color^;
      Cycle;
   Color^ := Orig
END.
```

Comments

1. After declaring the global variables, the cycle procedure and its local variable are declared.

2. The program stores the original color of the background, then cycles through the other 254 colors with a FOR loop, and finally returns the background to its original color.

Memory Locations Beyond the Size of MaxInt

Depending upon the size of your computer, you may want to address memory locations that are greater than the actual Integer value that Pascal can use.

This value, a predefined constant known as MaxInt, is 32767 with Kyan Pascal. If you want to address locations greater than that number, the equivalent of that address is a negative number calculated by subtracting 65536 from the desired memory location. The formula is

Equivalent address = Memory Location - 65536

For example, if you want to Assign the Pointer Variable Locate with memory address, 40000, you need to first find the negative equivalent of 40000 with the formual. The resulting value is -25536. This number would then be used in place of 40000. The following statement assigns this value to the Pointer Variable Locate:

```
Locate := POINTER(-25536);
```

NEW

Imagine how difficult it could be trying to keep track of each pointer, and making sure that none of the variables those pointers pointed at overlapped in memory. Well, relax, you don't have to! We don't have to be concerned with exactly wher in memory a pointer points, as long as it points to the data-type we are concerned with. In situations like these, we use the NEW command. Here is the syntax:

```
NEW (PointerIdentifier): ;
```

The NEW command works by finding an unused area of memory that is large enough to hold the data type the pointer points to. It then gives the address of that area to the pointer. Finally, it sets aside this memory so that it is not overlapped by any variables or pointers. The memory set aside by the computer for storage of pointer variables is commonly called the "HEAP". Consider the following lines of code:

```
VAR
    Locate : ^Integer;
    Letter : ^Char;

BEGIN
    NEW (Locate) ;
    NEW (Letter) ;
```

After the pointers are declared, they look like this:

```
Locate ===> ????
Letter  ===> ????
```

This is because they are uninitialized and could point to any location. However, after the NEW commands, the pointers look like this:

```
Locate ===>

Letter  ===>
```

Notice that we don't know where in memory these pointers are. But we do know that Locate points to an integer (two Bytes, remember) and that Letter points to a character.

Because of the way the NEW command works, we cannot use the Letter pointer to look at the locations Locate is pointing to. Note also that these fields are uninitialized; they most likely contain garbage until they are given a value. In the next section, we will show you how a linked list can be made using the NEW command and a record field.

POINTERS AND LINKED LISTS

In addition to using Pointers to access memory locations, you can also use them to identify the position of items in a data base list. This allows you to create lists of different types of data, and have each item in the list contain a pointer to the next or previous item. You can then access each item and decide what the program should do with it. The example used in this section is a very elementary list, using a simple record data-type; but you can imagine how complex the data base can become.

When using linked lists, remember a few important points:

1. Since a Pointer is used to indicate the location of each item in the list, you must use the New statement to load the Pointer with an unused location.

2. Linked lists are entered into memory as they occur in the program. This means that the first item entered will be the last item read when the program retrieves the list.

3. To indicate the first item entered on the list as the last item that will be found when reading the list, you must use a statement called **NIL**. It is like a reverse End Of Line or End Of File indicator. You can tell the program to read back through the list until it finds a **NIL** statement.

AN EXAMPLE OF A LINKED LIST

This sample program creates a linked list of Names and appointment dates. Each entry in the list is linked by a Pointer to the next item. The record itself is also located by a Pointer.

The Pointers are identified as Pointer-Types, and two variables are declared as Pointers. The first, **Appointm**, points to the memory location that stores the record; the second, **Pt**, stores the memory location of the linking Pointer. NIL is used to identify the first item in the list since it is the last item read when the program returns to examine the list.

When the program reads the list of appointments, it reads the last entry first. The pointer in that entry indicates the location of the previous entry, and so forth, until the pointer contains the NIL value.

```
PROGRAM Points(Input,Output);

TYPE
 String = ARRAY[1..15] OF Char;
 Appointer = ^AppointRec;   (* Pointer Type *)

 AppointRec = RECORD
  Person : String;
  Date : String;
  Link : Appointer
 END;

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

BEGIN
 Writeln;
 Writeln;
 Pt := NIL;
 New(Appointm);
 Write('Enter Name:  ');
   Readln(Appointm^.Person);
 Writeln;
 Write('Enter Date ');
   Readln(Appointm^.Date);
 Appointm^.Link := Pt;
 Pt := Appointm;
 New(Appointm);
 Appointm^.Person := 'Bob           '; (* Note:  There must be exactly
 Appointm^.Date := '02 - 05 - 86';       15 characters between quotes or
 Appointm^.Link := Pt;                 a "Wrong Type" error will result *)
 Pt := Appointm;
 New(Appointm);
 Appointm^.Person := 'David Brandes  ';
 Appointm^.Date := '02 - 06 - 86';
 Appointm^.Link := Pt;
 Writeln;
 Writeln;
 Writeln('The Appointments are:  ');
 Writeln;     (* program completed on page 166 *)
```

COMMENTS

1. The program declares a String data-type which holds the name of the person in the Record.

2. The Pointer data-type is identified as Appointer. It is declared to be an **^AppointRec** data-type. Remember that Pointers do not have to be declared as Variables; they can also be declared as data-types, with variables defined by that data-type.

3. A Record is declared which contains the fields: Person, Date, and Link.

4. The Variables, **Appointm** and **Pt**, identify variables of the Pointer type.

5. The program uses the Node identifier, **Appointm^**, to indicate the memory location where the actual record is stored. It appends the field identifier to that name to specify the actual record item. Because it uses the Node identifier, the program does not need to indicate the Record Identifier.

6. The format of each record first sets the variable, PT to the previous **Pt** value. It then defines the Person and Date fields. Each of these must contain a specific number of places because of the length of the Strings defined in the TYPE declarations.

7. Each record first identifies the value of the Pointer, **Pt**, which points to the previously saved record. Since the first record can not point to a previous record, the variable **Pt** in that record equals **NIL**. Successive Pointers are equated to the previous Link field.

8. After defining the Pointer value of each record as the location indicated by the previous Pointer in the variable, **Appointm**, the record assigns a new memory location for the next record. It then defines the elements of that record and equates the Link field with the existing value of **Appointm**.

To read the list defined in this program, use the following statements which read the list until the program finds a NIL Link-indicator. When reading the list, the Pointer, **Appointm**, will always indicate the location of the previous record until it reads the **NIL** indicator. To read the previous record,

equate the variable, Pt, with the Appoint^.Link field; then equate that field with the previous Pt value. The result is that each equation identifies the next field to be read.

The following lines read each record in the list back into memory. Once the record is read, it is printed to the screen. The read sequence continues until the NIL value is read in the Pt field.

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

Once you can read and write information to specific locations in memory, you need to be able to tell the computer to forget that information. The Dispose command performs that task.

DISPOSE

When you reserve space in memory to store Pointers and Nodes, those locations continue to hold the values stored in them even after the items they refer to are no longer used by the program. Consequently, when you are finished with Pointers, you should clear the memory locations that holds their values. The Dispose command performs this task.

The syntax of the Dispose statement is:

```
Dispose(PointerIdentifier);
```

If you forget to Dispose of saved Pointer values, you occupy memory spaces that can be allocated to other variables. The Dispose statement thus frees space for the user program.

CONCLUSION

This section has explained the use of Pointer Variables access memory locations. Pointers are used to read and write values directly into memory. They are also used to locate elements in a data-base list of items. For more information on using Pointers, refer to a standard Pascal manual.

The discussion of linked lists has barely scratched the surface of this important topic. We recommend that you refer to a textbook on structured data types for a thorough investigation of this powerful programming tool.

If you have mastered all the concepts and techniques illustrated in the Tutorial, you should be fairly comfortable with programming in the Pascal language. From now on, use your intuition to solve problems. Experiment. If you need to refresh your memory about the structure of specific commands or statements, consult the reference section of this manual.

V ASSEMBLY LANGUAGE PROGRAMMING

This section describes how to:

* Use the Kyan Pascal assembler ("AS")
* Include assembly code routines in a Pascal program
* Access parameters in Procedures
* Access values returned by Functions

This section does not explain how to write assembly language programs. It is intended primarily for programmers who already know how to write assembly code and wish to use it in their Pascal programs. (NOTE: If you have previously written assembly language routines with Version 1.- of Kyan Pascal, please refer to the end of this section for instructions on how to convert your routines to this version).

USE OF THE KYAN PASCAL ASSEMBLER

Kyan Pascal features a special purpose macro assembler called "AS". This assembler is optimized for maximum speed of assembly by limiting features. Symbol table listings, cross reference listings, nested macros, identifiers of greater than 6 characters, and linkable object code modules are not supported by "AS".

The Kyan Pascal compiler normally pipes its assembly language output to the assembler, which in turn produces an executable object file. If the -S option is used on the compiler, the output of the compiler will not be assembled. Instead, the compiler will generate a text file of assembler macros to disk with the filename **P.OUT**. The expansion of these macros can be found in the file **STDLIB.S** which is located on the Kyan Pascal disk. To obtain an assembly language listing with the macros expanded, add the " MEX ON" directive to line 1 of the macro file and assemble the file using the "-l" option.

The output will be an assembly language text file. This assembly language file can then be modified as required for special applications. When modifications are complete, the assembler can then be used as described below to generate an executable file.

For advanced programmers, Kyan Software offers a Code Optimizer Toolkit which includes the source code of the Kyan Pascal Runtime Library. When used with this source library, the assembler will conditionally assemble only the library routines required in the compiled application program. This feature saves memory and permits large applications to run in one program segment. It also allows the programmer to modify standard Pascal procedures and functions to meet special programming requirements.

RUNNING THE ASSEMBLER

The assembler "AS" can be run in two ways.

Option 1: Type "AS" at the prompt(%) and type <RETURN>. Then, enter the filename of the file to be assembled along with any of the options listed on the Help Screen.

Option 2: Type "AS" at the prompt(%), the filename, and the desired options on the same line. Then press <RETURN>.

The assembler has two options which are listed below:

- L	Produces a listing.
- O device:filename	Renames the output file.

The assembler listing and error messages can be redirected in two ways as shown below:

> Device:filename	Redirects output to a file
> P:	Redirects output to a printer

An example of assembler options and output redirection would be:

```
% AS_XYZ_-L_-O_MNO_>D1:ABC
```

With this command, the assembler would: (1) assemble the file named "XYZ"; (2) produce an executable file named "MNO"; and, (3) produce a listing of the program with the filename "ABC" on device 1.

You can stop the assembler at any time during assembly by pressing the <ESC> key. You will be returned to the system prompt (%).

NAMING AN EXECUTABLE FILE

The assembler has several features which control the name generated for an executable file.

Option 1:	Anytime the "-O" option is used, the executable file is given the name assigned in the option statement.
Option 2:	Any source filename that ends in ".s" will result in an executable file with the same filename with the ".s" stripped off.
Option 3:	If neither of the above options apply, then the executable file will be named "A.OUT".

RELOCATING PROGRAM ORIGINS

Under default conditions the compiler generates a binary file which starts at location $2000 in memory. If it is necessary to change starting locations, the following instructions should be inserted before the first line of the Pascal program.

```
#A
_Origin EQU $XXXX
#
```

where $XXXX is the new location (e.g., for graphics, $XXXX = $4000).

ASSEMBLY LANGUAGE ROUTINES

Kyan Pascal accepts assembly language routines as part of the Pascal program. This enables the programmer to include routines that are not restricted by the structure of standard Pascal. Five rules govern the use of assembly language in a Pascal program.

1. Assembly language routines must appear within the body of a Program, Procedure, or Function. That is, they must appear between a BEGIN/END block.

2. To indicate the difference between the Pascal program and the assembly code, assembler routines must begin with a #a label and end with a # label. The # sign must be placed in the first column, and the a must appear in the second column. Lines contained between these labels are left untouched by the Pascal compiler and are integrated into the final assembly language output file exactly as they are written.

3. All labels used in assembly language must begin in column 1.

4. Labels used as part of the assembler routine must not begin with an underscore character (_). The compiler uses labels with the format "_xxxxx". Consequently, if you use labels beginning with an underscore, the program may fail.

5. If the X register is used by an assembly language routine, it must first be saved and then later restored. The X register is used by the compiler as the system stack pointer. If it is used and not restored, the compiler will lose track of all variable references (i.e., your program will crash).

In summary, to use assembly language statements:

1. Place all code between BEGIN/END statements
2. Include all code between #a and # labels
3. Begin all labels in column 1
4. Do not use labels that begin with an underscore (_).
5. Save and restore the X register

ASSEMBLER DIRECTIVES

Kyan Pascal supports 25 assembler directives. Directives are also known as pseudo-code since they appear in the assembly language listing but are not part of the language of the microprocessor. Instead they are terms understood by the assembler itself. The 25 directives are:

Kyan Pascal Assembler Directives

Symbol	Description
ORG	Origin -- indicates that the assembled code should start at the specified location in memory (ORG $4000; start at $4000)
EQU	Equate a label with a value which will be assigned to the label whenever it appears in the program. In effect, EQU defines a constant (e.g., A EQU $FF; define A to be $FF)
DB DW	Define Byte and Define Word are used to build tables and strings that reside in other parts of the assembly program. When the program executes, it sets the index register to the values identified by these directives. These values indicate where the table or string resides in memory. (e.g., DW $FF00; put $00 in next byte and $FF in following byte).

>	Least Significant Byte (LSB) is used with a label or specific value to indicate the least significant byte of a 2-byte hexadecimal number (e.g., >$FF01 = $0001 or, if WLABEL = $11EE, then >WLABEL = 00EE).
<	Most Significant Byte (MSB) is used with a label or specific value to indicate the most significant byte of a 2-byte hexadecimal number (e.g., <FF01 = $00FF, or, if WLABEL = $11EE, then <WLABEL = $0011).
DS	Define Storage saves space for the number of bytes in the expression field (e.g., ds 5;reserves 5 bytes).
STR	String counts the number of characters in the expression fields and puts that number in the first byte followed by the ASCII values of each character in the string (e.g., str 'abc'; first byte is 3 followed by ascii values of a, b, and c).
IFDEF	If Defined assembles the code following the directive if the identifier in the expression field is defined (e.g., ifdef abc; assemble what follows if abc is defined).
IFNDEF	If Not Defined assembles the code following the directive if the identifier in the expression is not defined (e.g., ifndef abc; assemble what follows if abc is not defined).
IFEQ	If Equal assembles the code following the directive if the expression is equal to zero (e.g., ifeq x-1; assemble what follows if x was previously defined to be 1).
IFNE	If Not Equal assembles the code following the directive if the expression is not equal to zero (e.g., ifne x; assemble what follows if x was previously defined and not equal to zero).
ELSE	Else optionally follows one of the IF- directive and reverses the conditional assembly (e.g., ifeq y .... else .... endif; assemble what follows "else" if y is defined as not equal to 0, otherwise don't assemble what follows).

ENDIF End If directive ends the conditional assembly associated with IF- or IF- ELSE directives (e.g., ifdef abc .... endif; end the conditional assembly associated with the IFDEF).

INCLUDE Include file in expression field (e.g., include xyz ; include the file xyz).

LST ON Listing On turns on the listing at that point.

LST OFF Listing Off turns off the listing at that point.

DSECT Data Section defines an area of memory for data only. For example, define a data area in high memory:

```
dsect
ram equ $b000
abc ds 2000
dfg ds 1000
dend
```

DEND Data End ends the definition of the area in memory reserved for data only.

MACRO Macro definition follows. For example, define the macro chrout:

```
macro chrout
ora $80
jsr cout
endm
```

ENDM Macro definition ends.

MEX ON Macro EXpansion ON for listing.

MEX OFF Macro EXpansion OFF for listing.

ASC ASCii is used to put the ASCII values to the string in the expression field in the bytes following the ASC directive (e.g.,asc 'ok'; put ASCII value of '0' and 'k' in next 2 bytes).

DFLAG Define FLAG is used to define or redefine the variable in the expression field. The value of the definition has no meaning. The DFLAG directive is used with the IFDEF and IFNDEF directives to assemble code required by one or more already assembled macros or code segments.

Do not use parentheses in assembler directives. Expressions are evaluated from left to right, and no one operator takes precedence over another.

ASSEMBLY CODE AND PROCEDURES

Data is normally transmitted to a Procedure in the form of parameters. The formal parameter list that is part of the Procedure declaration defines the data the Procedure expects to receive from the main program or the calling routine. The main program transmits the actual data in the Actual Parameter List that accompanies the call to the Procedure.

For example, the Procedure declaration *AddXY(X,Y: Real);* expects to receive two real numbers from the calling routine. The main program might call this procedure with the statement *AddXY(3.5,4);*.

If the Procedure is an assembly code routine, it must read the parameters from the stack which contains the list of parameters passed to the Procedure.

Every time a Procedure is called by another routine, Pascal creates a call frame on the stack to hold the parameters being passed to the Procedure. The zero page location Stack Pointer (_SP) and the local variable pointer (_Local) are used to reference the stack.

The Assembler can identify parameters only by their position in the Stack; but, since the Assembler cannot locate those values by itself, the programmer must identify each parameter by telling the Assembler its location in the Stack. This means that the position of each parameter must be calculated manually before it is used in an assembly language routine.

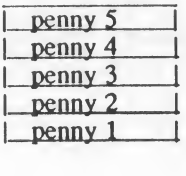
To calculate the position of each parameter in the Stack you need to understand the structure of the stack and the number of bytes used to store different types of parameters.

THE STACK

When parameters are passed to a Procedure, they are placed in a stack. Each value entered on the Stack pushes a preceding value down the Stack. It's like putting pennies in a coin-change holder with a counter. The first penny you put in is pushed down by the next one, and the counter records how many pennies you have saved.

If you have saved 5 pennies, the first one is at the bottom of the coin-holder and the counter indicates that you have saved 5 cents. Note that penny number one is at the bottom of your stack .

The following diagram illustrates this structure.

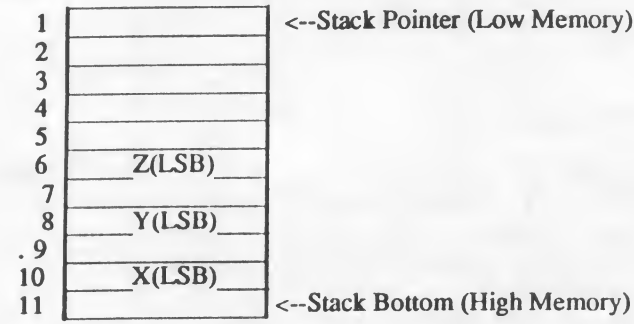


The Stack passed to the Procedure is identical in structure to the coin holder. The first parameter passed to the Procedure is located at the bottom of the stack. The others are pushed on top of it as they are passed to the Procedure.

Examine the following Procedure declaration and the corresponding diagram of the Stack that would hold the parameters passed to it from the calling routine. Remember that the Stack is set aside in memory and that _SP is used to keep track of the current value of the stack pointer.

```
PROCEDURE StackSample(X,Y,Z: Integer);
```

The parameter list passed to this Procedure would be stored in the Stack as



The five empty locations at the top of the Stack are used by the computer to store information about the Stack itself. When using the Stack, however, these positions must always be included in calculating the position of the parameters stored in the Stack.

For now, just make sure you understand that the first parameter, X, is stored in positions #10 and #11, that Y is in positions #8 and #9, and Z in positions #6 and #7. The Least Significant Byte (LSB) is the lower number and the Most Significant Byte (MSB) is the higher number.

To locate a specific item in the parameter list, calculate how many bytes of memory separate the Stack pointer from the value you want to locate. Since Pascal programs and subroutines always declare the parameters before the body of the program or subroutine, it is simple to calculate positions of parameters

on the stack. The only problem with calculating the position of the parameter is that each type of parameter takes up a different amount of space.

The following list indicates how many bytes of memory are required to store a type of data.

MEMORY STORAGE ALLOCATION

DATA TYPE	BYTES ALLOTTED
I. Real	8
Integer	2
Char	1
Boolean	1
Pointer	2
II. ARRAY[1..n] OF Integer	2*n
ARRAY[1..n] OF Char	n
ARRAY[1..n] OF Boolean	n
III. Value Parameter (Real)	8
Value Parameter (Integer)	2
Value Parameter (Char, Boolean)	1
Value Parameter (ARRAY[1..n] OF Integer)	2*n
Value Parameter (ARRAY[1..n] OF Char or Boolean)	n
IV. Variable Parameter (Address of parameter) (All types)	2

Make sure you understand the amount of memory required by each data type before you try calculating positions of parameters in the Stack. There are four groups of data types. The first consists of the predefined data types. The second group consists of ARRAYs of the predefined types. The third contains the data types when they are passed as Values to a Procedure or Function. The last indicates the size allotted for any Variable passed to a Procedure or Function.

If a Procedure or Function uses a Real number, that number takes up 8 bytes of memory. If a Procedure or Function uses an ARRAY [1..10] OF Integer,

the array uses 20 bytes of memory (2*n). If the Procedure or Function is passed an Integer Value in the Parameter list, the Value requires 2 bytes of memory. If a Procedure or Function is passed any type of Variable, each Variable uses 2 bytes of memory.

Storage of Real Numbers

Kyan Pascal stores Real numbers as Binary Coded Decimals (BCD) and allots them 8 bytes of storage. The 8 bits contained in each byte are divided into 2 parts which are called NIBBLES. The part with bits 0 through 3 is called the Low Order Nibble. The part with bits 4 through 7 is called the High Order Nibble. For example:

```
                    (1 Byte = 8 Bits)
|<-------------------------------------------------------------->|
7       6       5       4       3       2       1       0
|<------High Order Nibble---------|--------- Low Order Nibble ------>|
```

When Real numbers are stored, they are organized as follows:

Byte	Low Order Nibble	High Order Nibble
0	1st significant digit	Bit 4: sign of exponent 0 = + / 1 = - Bit 5: sign of number 0 = + / 1 = - Bits 6 & 7: always zero
1	3rd significant digit	2nd significant digit
2	5th " "	4th " "
3	7th " "	6th " "
4	9th " "	8th " "
5	11th " "	10th " "
6	13th " "	12th " "
7	2nd digit of exponent	1st digit of exponent

The decimal point for the Real number is automatically placed between 0 and 1.

A Sample Stack Calculation

A sample stack calculation should clarify the concepts explained above. The following Procedure receives an Integer from the calling routine. Imagine that the Procedure is simply going to double that value.

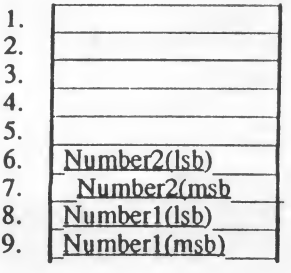
```
PROCEDURE Double(Number: Integer);
```

The calling routine might contain a statement like *Double(10);* If the Procedure contains assembly code, you must read the value passed to Number by calculating Number's position in the Stack. After pushing the Number on the Stack, 5 more bytes are added which contain information for using the call frame. Therefore, to calculate the location of Number, add 5 to the position of the Stack Pointer.

If the Procedure Double used 2 Values, its declaration would be

```
PROCEDURE Double(Number1, Number2: Integer);
```

The Stack containing the Parameter Values Number1 and Number2 would look like



Since you know that Number1 and Number2 are Integers and that Integers takes 2 bytes of memory, you should be able to calculate that the least significant byte (lsb) of Number1 is 8 bytes brom the top of the Stack. The lsb of Number2 is 6 bytes from the top of the Stack.

RULE: Each parameter is entered in the Stack as it is encountered. Successive parameters are put on top of previous parameters.

The Stack Pointer and the Label "LOCAL"

You may have wondered about the 5 bytes pushed on the top of the Stack. The first two are the subroutine return linkage. They contain the address that points to the memory location with the next executable instruction. The next 2 bytes contain the address of the previous stack pointer. The last byte is the lexical level of the current procedure or function.

The Kyan Assembler uses 2 predefined labels, _SP and _Local, which always contain the address of the current and previous Variables Stacks. A third predefined label, _T, contains the address of temporary zero page memory that the assembly routine may use as workspace. There can be up to 15 temporary bytes beginning at location _T and continuing to _T+14.

The absolute locations of these labels are

```
_SP       EQU   4
_LOCAL    EQU   2
_T        EQU   16
```

These labels can be used by the assembly routine to access values placed on the Stack.

ASSEMBLY LANGUAGE PROCEDURES AND VALUE PARAMETERS

To access Value Parameters passed to an assembly code Procedure, determine the offset from the Stack Pointer to the value parameter being passed. First, load the accumulator with the least significant byte of the Value and store it in workspace _T. Then repeat the process to get and store the most significant byte of the Value. Repeat the process for each Value you want to access. You must access Values by loading their least and then most significant bytes because the 6502 processor can only handle one 8-bit piece of data at a time.

The following Procedure is passed three integer parameters by a calling routine. It also uses two integer variables that are local to the Procedure. The assembly routine accesses the third Value Parameter, C, and loads it into _T and _T+1.

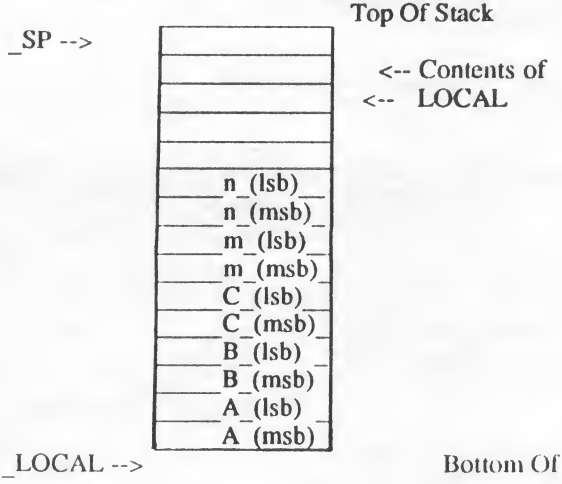
```
PROCEDURE  Access(A, B, C: Integer);
VAR
  m,n : Integer;
BEGIN
#a
     LDY #9     (* the offset from _SP to "C" *)
     LDA  (_SP),Y
     STA  _T
     INY
     LDA  (_SP),Y
     STA  _T+1
#
END;
*********************************************************
```

COMMENTS

1. Three Integer values, A, B, and C are passed to the Procedure by the calling routine.

2. The Procedure contains two local integer values, **m** and **n**.

3. The Parameters Stack has the following structure.



4. Since all of the Values and Local variables are Integers, each element uses 2 bytes of memory. (Refer to the List in the previous section for memory size used by the different data types.)

5. The top 5 stack positions contain the Stack Pointer and the address of the Bottom of the Stack.

6. The offset from _SP to C is the total of the 5 bytes at the top of the Stack, plus 2 bytes each for the integer variable, m and n; i.e., 9 bytes.

7. The statement LDY#9 loads the offset value to C in the Y register.

8. The statement LDA (_SP),Y uses the offset from the _SP to load the least significant byte of the Integer Value being passed, i.e. C, into the accumulator.

9. That value is then stored in the temporary workspace, _T, for use within the Procedure.

10. Finally, the Y register is incremented and then used to provide the offset to the most significant byte of the Value C.

The general rule for calculating the position of a Value Parameter in the Stack is

Offset (from _SP) = 5 bytes + bytes of memory used by values above the Value desired

The following example shows another example of passing parameters by value and accounting for local variable definitions.

```
Procedure (x: integer);
Var b: boolean;
BEGIN
  ....
  ....
#A
  ....
#
END;
```

_SP +5	Add 5 bytes for overhead
+1	Add 1 byte for local boolean, b
+6	LSB of X
+7	MSB of X

The number of bytes used for the local variables must be added to calculate the offset from _SP for a passed parameter.

LOCAL

You can also use the Label _LOCAL to calculate the position of a Value Parameter in the Stack. Remember that the predefined, absolute value of _LOCAL is 2. _LOCAL is the address of the bottom of the stack.

The formula for calculating the position of a Value Parameter using the Label _LOCAL is

Offset(from _LOCAL) = Bytes used by the desired Value
+
Bytes below theValue on Stack

The offset must be subtracted from the value of _LOCAL to reference the parameter.

Passing Value Parameters: Summary

So far, we have discussed only Values passed in the Parameter list. Assembly code, as we have noted, cannot identify the Pascal concept of SCOPE that is usually involved with Parameters.

Consequently, when a Pascal routine calls a subroutine, it passes the values in a Stack. Since all Pascal programs and subroutines list the declaration before the body statements, the position of the passed values is easily calculated.

A **word of caution**: never try to calculate the stack location of values based upon their relative positions in the program. That stack only contains the values passed to the subroutine. It cannot be used to access other values or variables used by the main program.

PROCEDURES AND VARIABLE PARAMETERS

An Assembly language Procedure reads Variable parameters from the stack in much the same way that it reads Values. The only difference is that when Variables are passed to a subroutine, the Stack contains only the address of the Variables being passed. In effect, the position in the stack that contains the Variable actually contains a pointer to that variable. Since all pointers require 2 bytes of memory, all Variable parameters in the Stack require 2 bytes.

Once the Variable's address is determined, use the temporary workspace to store the actual variable. If data manipulations within the Procedure change the value of the Variable, the new value can be stored back into memory by referencing the address stored in the temporary workspace.

The following sample procedure expects to receive the identifiers, or names, of 3 integer Variables. It then reads the address of Variable C, which is the third Variable in the Parameter List. Once the address is identified, the actual value of the Variable is read. The calling routine might execute a command such as

```
XYZ(Length, Width, Height);
```

The 3 Variables will be read into the Parameter Stack as A, B, and C.

```
**********************************************************
PROCEDURE XYZ(Var A, B, C : Integer);

BEGIN
#A
  LDY  #5          (* Load offset to C *)
  LDA  (_SP),Y     (* Get LSB of ADDRESS of C *)
  STA  _T          (* Save ADDRESS in _T *)
  INY
  LDA  (_SP),Y     (* Get MSB of ADDRESS of C *)
  STA  _T+1        (* Save ADDRESS in _T+1 *)
  LDY  #0
  LDA  (_T),Y      (* Get LSB of C *)
  STA ...
  ...
#
END;
```

COMMENTS

1. Since C is the last Variable passed to the Stack, it is 5 bytes from the top of the stack.

2. LDY #5 sets the offset in the Y register.

3. The accumulator is then loaded with the least significant byte of the address that refers to the Variable C.

4. The LSB of C's address is loaded into the temporary workspace, _T.

5. The Y register is incremented, and the most significant byte of C's address is read.

6. The MSB of C's address is then read and stored in _T+1.

7. Once the address of C has been determined, the value addressed by _T is loaded into the accumulator.

The offset to C is calculated in the usual manner:

+ 5 bytes	offset from the Stack Pointer
+ 0	local Variables
5 bytes	total offset to LSB of C's address

The offset to the MSB of C is 6.

ASSEMBLER ROUTINES AND FUNCTIONS

Like a Procedure, a Function also receives data in the parameter list. It transmits the value calculated by the Function, however, in a slightly different manner. The manner is determined by the structure of the Function subroutine.

Remember that a Function receives data in the Parameter List, and then, after performing calculations that may include local variables, stores the resultant value in the location identified by the Function's name. The Parameter Stack for a Function, therefore, contains the Function's identifier.

As a rule, the Function's identifier is placed on the Parameter Stack after the passed parameters and before the local variables.

To calculate the position of the Function's identifier, use the same memory allotment guidelines that you used for Procedures.

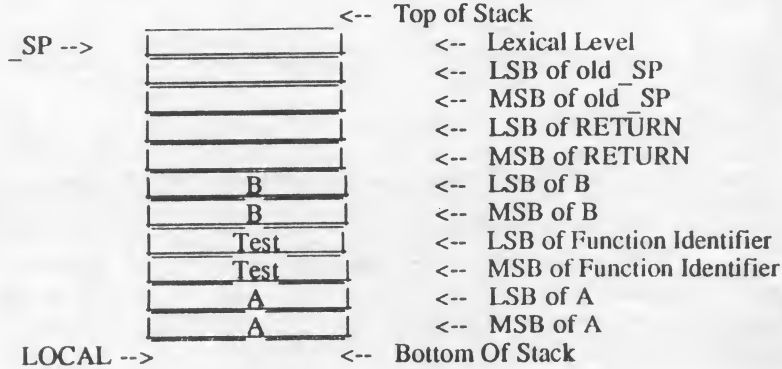
The following sample Function receives an Integer Value from the calling routine and returns an Integer which is identified by the Function's identifier. It also uses a local variable which is also an Integer.

```
************************************************
FUNCTION Test(A: Integer): Integer;

VAR
 B : Integer;

BEGIN
Test := 0;  (* Assignment required for ISO compatibility *)
...
END;
************************************************
```

The Stack for this Function is diagrammed below. It is more detailed than previous Stack illustrations because it illustrates the byte structure of each Parameter item. Remember that the Function's identifier is passed to the Stack after the parameters and before the local variables.



A Sample Function in Assembly Language

The following Function, XYZ, is passed three Integer Values and returns a Boolean value with the Function's name. Note that a Boolean requires only 1 byte of memory, so locations on the Stack are calculated accordingly. It also declares a local Variable, X.

The code in the sample Function reads the Value of B. After performing a series of undetermined calculations, it theoretically determines a Boolean value. That value is stored in the accumulator. The accumulator is then stored in the Stack location associated with the Function's identifier.

NOTE: Conformance to the ISO standard dictates that the compiler return an error whenever it encounters a FUNCTION written entirely in assembly language. This occurs because ISO requires any function identifier to be explicitly assigned a return value. For this reason, a dummy assignment using the function identifier should be included as the first line of a function.

```
*****************************************************************
FUNCTION  XYZ(A, B, C: Integer): Boolean;

VAR
 X: Integer;

BEGIN
   XYZ := TRUE;          (* Required for ISO compatibility *)
#a
  LDY  #10       (* The offset to B *)
  LDA  (_SP),Y           (* Put LSB of B in Accumulator *)
  STA   _T       (* Store LSB of B in workspace *)
  INY
  LDA  (_SP),Y           (* Put MSB of B in Accumulator *)
  STA   _T+1     (* Put MSB of B in workspace *)
   .
   .
   .
  LDA ...        (* Put Boolean value in Accumulator*)
  LDY  #7        (* The offset to Function Identifier*)
  STA  (_SP),Y           (* Put Boolean value in Identifier, XYZ*)
   .
#
END;
```

COMMENTS

1. The FUNCTION XYZ is assigned a dummy value to conform to ISO requirement.

2. The offset from the Stack Pointer to B is 10:

+5 bytes	offset to first stack parameter
+2	local Integer Variable
+1	Boolean XYZ (the Function identifier)
+2	Integer Variable C
10 bytes	total offset to B

3. The program then reads the least and most significant bytes of the Value B, storing each in the temporary workspaces, _T and _T+1.

4. Finally, a Boolean value is loaded into the accumulator. That value is then loaded into the Stack location reserved for the Function Identifier, XYZ, which is offset from the Stack Pointer by 7 bytes.

+5 bytes	offset to first stack parameter
+2	local Integer Variable X
7 bytes	total offset to XYZ

Remember that the only difference between Functions and Procedures is that the Function Identifier is placed on the Parameter Stack. It is located after the passed parameters and before the local variables.

MISCELLANEOUS OPERATIONS

The following subprograms illustrate how to use assembly code to Peek and Poke memory locations. A Poke statement is a Procedure since it enters a value into a specific memory location. A Peek statement is a Function since it returns the value determined by the addressed memory location.

POKE

The following Procedure Pokes the value, Val, into memory location, Loc. The rules for locating parameters on the Stack indicate that Loc is offset from the Stack Pointer by 7 bytes and that Val is offset by 5 bytes. The Procedure first reads the Parameter List to determine the memory location to be poked. It

then reads the value to be entered. Finally, after clearing the Y register, it stores the value to be poked into the address contained in _T.

```
*************************************************************
PROCEDURE Poke(Loc, Val: Integer);

BEGIN
#a
 LDY  #7            ; Offset from _SP to Loc;
 LDA  (_SP),Y       ; Get LSB of Loc;
 STA  _T            ; Save LSB of Loc;
 INY
 LDA  (_SP),Y       ; Get MSB of Loc;
 STA  _T+1          ; Save MSB of Loc;
 LDY  #5             ; Offset from _SP to Val;
 LDA  (_SP),Y       ; Load Val into Accumulator;
 LDY  #0            ; Clear Y register;
 STA  (_T),Y        ; Store the value in the Accumulator
                    ; in memory location _T;
#
END;
*************************************************************
```

PEEK

The Peek statement is actually a Function since it returns a single value that is based upon the Location parameter which indicates the memory address to be read.

That value is stored in the memory location reserved for the Function's identifier.

The following assembly code subroutine returns the value contained in the memory location, **Loc**.

```
*************************************************************
FUNCTION  Peek(Loc: Integer): Integer;

BEGIN
   Peek := 0;
#a
 LDY  #7            ; Offset to Loc ;
 LDA  (_SP),Y       ; Get LSB of Loc;
 STA  _T            ; Save LSB of Loc in workspace;
 INY
 LDA  (_SP),Y       ; Get MSB of Loc;
 STA  _T+1          ; Save MSB of Loc in workspace;
 LDY  #0            ; Clear Y register;
 LDA  (_T),Y        ; Load Accumulator with the
                    ; Address being Peeked;
 LDY  #5            ; Offset to Function Identifier
 STA  (_SP),Y       ; Store contents of Accumulator
                    ; in LSB of Function Identifier
 INY
 LDA  #0            ; Load Accumulator with 0 for MSB
                     of return integer
 STA  (_SP),Y       ; Store contents of Accumulator;
                    ; in MSB of Function Identifier;
#
END;
*************************************************************
```

COMMENTS

1. Peek is the reverse of Poke.

2. Read and store the memory location you want to examine in temporary workspace.

3. Read the memory location of the Function Identifier and write the value stored in the temporary workspace into the Function Identifier's location.

4. Note that Peek returns an integer and Poke writes to memory only a byte.

Converting Assembly Language Routines from Kyan Pascal (Version 1.-)

Assembly language routines written for Version 1.- of Kyan Pascal must be converted to run under the new compiler environment. The following changes are required to your old routines.

1. The local variable stack now begins allocating storage at SP+5 instead of SP+3.

2. The predefined labels, T, SP, and LOCAL are now redefined as _T, _SP, and _LOCAL. Note the underscore character preceding the label.

3. The new compiler uses the underscore convention to distinguish compiler-assembler system labels from user labels. Do not use the underscore in your own labels.

CONCLUSION

If you know Assembly Language programming, you should now be able to include Assembler routines in your Pascal programs. You should also have learned how to pass data items through parameter lists.

Since all data items are similar to Values or Variables, only those types of data were covered in this section. Values may be passed by any of the data types itemized in the list at the beginning of this section. Variables are always put on the Parameter Stack in terms of their pointers--i.e. they always occupy 2 bytes of memory allocation.

(This page left blank for your notes.)

VI WORKING WITH KIX

OVERVIEW

This manual has explained Kyan Pascal in terms of the file procedures used by the DOS 2.5 disk operating system. We assumed that most users were familiar with it, and we felt that the fewer issues forced upon the beginning user, the easier it would be to learn Pascal. Actually, Kyan Pascal uses the KIX programming environment. KIX gives you the functionality of DOS 2.5 from a command line prompt, thereby eliminating the time required to use the DOS 2.5 menu.

When you boot Kyan Pascal, the KIX prompt % appears and awaits your commands. One of the powerful features of KIX is that you can issue KIX commands whenever you have the KIX prompt. This eliminates the need to access the DOS 2.5 menu whenever you want to use a file management function.

This section presents an overview of the KIX system. It then explains the different groups of KIX commands.

* Device Control
* Listing Disk Directories and File contents
* Manipulating Files, Devices, and Disks
* Searching Files and Devices

NOTE: You can enter a KIX command whenever you have the % prompt.

THE KIX ENVIRONMENT

When you boot your Kyan Pascal disk, KIX is automatically loaded into memory. It stays there throughout the programming session. It assigns **D1:** as the default device so that you can specify filenames without device prefixes. KIX allows you to change this default device using the **CD** KIX command.

When you are writing and running programs using the KIX system, it is not always necessary to specify a device name before a Kyan system filename (i.e., ED, PC, AS, STDLIB.S, LIB). KIX automatically searches for the Kyan files in the devices on line. First, it looks in the RAMdisk; if the file isn't there, KIX looks in the current working directory (e.g., "D2:"). If not found there, KIX looks in the system device (e.g., D1:). If the file is not found in any of these devices, KIX will return an error message. Thus you don't have to worry about specifying correct device names each time you call a system file.

KIX Command Structure

Each KIX command has the same command line format:

% command_[-options]_[Device:Filename]

At the KIX prompt (**%**), you can enter any KIX command. KIX also allows you to use both upper and lower case letters.

Device:Filenames

To access a file, identify its device prefix and its filename. KIX does not mind if you use lower case characters. It will convert these to upper case automatically. As previously mentioned, you do not always need to use the device name. KIX will automatically assume the default device if no other is specified. Other than these exceptions (for prefixes and lower case characters), KIX has the same filename rules that apply to DOS 2.5.

DEVICE CONTROL

There are two KIX commands that are used for device control.

PWD Print Working Device
CD Change Device

PWD: Print Working Device

PWD, the Print Working Device command, prints the name of the current default device on the screen. It is extremely useful when you can't remember where you are in the system's structure.

KIX automatically assigns D1: as the default device when Kyan Pascal is booted. Thus, if you invoke the PWD command when you first boot the system, it will print

D1:

CD: Change Device

The Change Device command lets you change from one default device to another.

For example,

% CD_D8: Sets the system default device to device **D8:**, the RAMdisk. If you execute a PWD command now, D8: will be shown as the default device.

LISTING DISK DIRECTORIES AND FILE CONTENTS

Whenever you have the KIX prompt (%), you can execute two different commands that return information about the device or file you are working with. They are

LS	**List Directory**
CAT	**Concatenate**

LS: List Directory

The LS command lists all of the files that reside in a particular disk's directory and the number of free sectors on that disk. The following notes apply to the LS command.

- o Wildcard characters are supported, so you can limit the list to specific files.
- o If you do not specify a device, KIX will assume the default device.
- o Redirection to an output device other than the screen is supported with the > symbol.

Some examples of LS are:

LS_D1:	List all files on drive 1 to the screen.
LS	List all files on the default device to the screen.
LS_D2:*.P_>P	Direct a listing of all files with a ".P" extension on drive 2 to the printer.
LS_D8:_>dir.1st	Direct a listing of all files on drive 8 (RAMdisk) to a file on the default device named DIR.LST.

CAT: Concatenate

The CAT command lets you print the contents of a file and/or copy many files into a single file.

For example,

```
CAT_d2:myfile.p
```

displays the contents of MYFILE.P on the screen.

Normally, CAT prints to the screen but you can redirect output to other devices using the > symbol. For example,

```
CAT_d1:myfile.p_>p:
```

will send the contents of MYFILE to the printer.

Another example,

```
CAT_myfile.p_>d2:myfile.p
```

will send MYFILE.P from the default device to drive 2, using the same filename.

CAT can also be used to merge a number of small files into a larger file. The following example merges three chapter files into one large file named Book.

```
CAT_Chapt1_Chapt2_Chapt3_>d2:Book
```

MANIPULATING FILES, DEVICES, AND DISKS

These commands provide you with an alternative to the file management functions found on the DOS 2.5 menu.

CP	**Copies files**
MV	**Renames files**
RM	**Removes files**
CHMOD	**Changes file protection status**
FORMAT	**Formats a disk**
SD	**Screen Dump**

The functions of these commands are obvious. The important point to remember is that you must use valid filenames to identify the source and destination files.

CP: Copy

The copy command produces a replica of the source file in the destination file. The syntax is:

CP_Source_Destination

Here are some examples of copy commands.

CP_myfile.p_d2:xyz.p	Copy MYFILE.P from the default device to drive 2; name the new file XYZ.
CP_d1:*.S_d8:StdLib.S	Copy the first file encountered that has a ".S" extension to D8: (the RAMdisk) and give it a new name of StdLib.S.
CP_d8:MyProg.P_d1:MyProg.P	Copy MyProg.P (a Pascal source code file) from drive 8 to drive 1.

MV: Move

The Move command allows you to rename a file. When a file is renamed, the original file is deleted.

The syntax of the Move command is:

MV_old.nam_new.nam

Here are two examples of the Move command:

MV_d2:myfile_yourprog	Changes the name MYFILE on drive 2 to YOURPROG.
MV_*.p_*.txt	Uses a wildcard to replace the ".P" extension on all files on the default device with a ".TXT" extension.

RM: Remove

The remove command deletes files.

The syntax of the Remove command is:

RM_Device:Filename

Some examples of the Remove command are:

RM_d2:yourprog	Removes the file YOURPROG from drive 2.
RM_myfile.*	Uses a wildcard to remove all files named MYFILE, regardless of their extensions, from the default device.

CHMOD: Change Protection Mode

Files can be locked so that they cannot be written to. Conversely, locked filed can be unlocked so that they can be written to. The CHMOD command is used to lock and unlock files. It locks files when followed by a -W extension and unlocks files when followed by a +W extension

The following examples demonstrate the use of CHMOD.

CHMOD_-W_myfile	Locks the file MYFILE on the default device.
CHMOD_+W_d8:*.p	Unlocks all files with a ".P" extension on drive 8.

FORMAT: Format A Blank Disk

The Format command lets you format a disk in either single or enhanced density while still remaining in the system. It is a valuable tool when you suddenly find yourself in need of a formatted disk. The syntax of the FORMAT command is:

FORMAT_(device number)_(density)

where the density parameter is S for single and D for enhanced.

Some examples are:

FORMAT_2_S	Format drive 2 disk in single density.
FORMAT_1_D	Format drive 1.in enhanced density

Always be sure to include a space between the device number and the density parameter.

Note: It is not necessary to format the RAMdisk (D8:).

SD: Screen Dump

The screen dump command outputs the current screen display to the printer. If no printer is present, the command is ignored.

Wildcards

KIX supports an Asterisk (*) wildcard that can be used to replace strings and characters in filenames. It can represent any string of characters, including a null or empty string.

This wildcard makes KIX commands even more powerful. For example, if you can't remember the exact name of a file, you can use a wildcard in the filename to help you find it.

Exclamation (!) Command

Typing an exclamation point ("!") and <RETURN> at the KIX prompt will re-execute the last command or program entered. This command is useful for running a program more than once without having to reload it.

Returning to KIX from DOS 2.5

When you are at the DOS 2.5 menu, you may want to return to the KIX environment. You can do this by Using the DOS "L" option to Binary load the "AutoRun.SYS" from Side 1 of your Kyan Pascal disk.

APPENDIX A

GUIDE TO ISO STANDARD PASCAL

DATA TYPES

POINTERS

STRUCTURED: Array, File, Set, Record

SIMPLE: Real
Ordinal
..Enumerated
..Predefined (Boolean, Integer, Char)
..Subrange

STANDARD IDENTIFIERS

CONSTANTS: False, MaxInt, True

TYPES: Boolean, Char, Integer, Real, Text

VARIABLES: Input, Output

FUNCTIONS: Abs, ArcTan, Chr, Cos, Eof, Eoln, Exp, Ln, Odd, Ord, Pred, Round, Sin, Sqr, Sqrt, Succ, Trunc

PROCEDURES: Dispose, Get, New, Pack, Page, Put, Read, Readln, Reset, Rewrite, Unpack, Write, Writeln

TABLE OF SYMBOLS

SPECIAL SYMBOLS

+	-	*	/	=
<	>	<=	>=	<>
.	,	:	;	:=
(	)	[	]	|
..	(*	*)	{	}

WORD SYMBOLS (RESERVED WORDS)

and	end	nil	set
array	file	not	then
begin	for	of	to
case	function	or	type
const	goto	packed	until
div	if	procedure	var
do	in	program	while
downto	label	record	with
else	mod	repeat	

DIRECTIVE

forward

APPENDIX B

TECHNICAL SPECIFICATIONS

Runtime Memory Map

Programs are loaded at $2000 and relocated if a graphics mode is used.
See Chapter III for more information on program relocation for graphics.

0	$ 7FF	Atari system overhead
$ 800	$1FFF	File Management System
$2000	$8BFF	User Program
$8C00	$BBFF	Kyan Runtime Library (LIB)
$BC00	$BFFF	Screen Area and display list
$C000	$FFFF	System ROM

Technical Data

OPERATING SYSTEM:	DOS 2.5
INTEGER RANGE:	-32768 to +32767
REAL RANGE:	-1.00E+99 to +1.00E+99
CHARACTERS:	ASCII character set with corresponding ordinal values
SET:	Maximum 256 members
POINTER:	Represented by 16-bit integers
CUT BUFFER SIZE:	2K
BCD MATH PRECISION:	13 digits
MIN. RAM REQUIRED:	48 K
MAX IDENTIFIER LENGTH:	256 characters

APPENDIX C

COMPILER ERROR MESSAGES

NUMBER	DESCRIPTION
1	error in simple type
2	identifier expected
3	'Program' expected
4	')' expected
5	':' expected
6	illegal symbol
7	error in parameter list
8	'of' expected
9	'(' expected
10	error in type
11	'[' expected
12	']' expected
13	'end' expected
14	';' expected
15	integer expected
16	'=' expected
17	'begin' expected
18	error in declaration part
19	error in field-list
20	'.' expected
21	'*' expected
47	Function cannot return a structure in ISO Pascal
50	error in constant
51	':=' expected
52	'then' expected
53	'until' expected
54	'do' expected
55	'to'/'downto' expected
56	'if' expected
57	'file' expected

58	error in factor
59	error in variable
101	identifier declared twice
102	low bound exceeds high bound
103	identifier is not of appropriate class
104	identifier not declared
105	sign not allowed
106	number expected
107	incompatible subrange types
108	file not allowed here
109	type must not be real
110	tagfield type must be scalar or subrange
111	incompatible with tagfield type
112	index type must not be real
113	index type must be scalar or subrange
114	base type must not be real
115	base type must be scalar or subrange
116	error in type of standard procedure parameter
117	unsatisfied forward reference
118	forward reference type identifier in variable declaration
119	forward declared; repetition of parameter list not allowed
120	function result type must be scalar, subrange, or pointer
121	file value parameter not allowed
122	forward declared function; repetition of result type not allowed
123	missing result type in function declaration
124	F-format for real only
125	error in type of standard function parameter
126	number of parameters does not agree with declaration
127	illegal parameter substitution
128	result type of parameter function does not agree with declaration
129	type conflict of operands
130	expression is not of set type
131	test on equality allowed only
132	strict inclusion not allowed
133	file comparison not allowed

134 illegal type operands
135 type of operand must be Boolean
136 set element type must be scalar or subrange
137 set element types not compatible
138 type of variable is not array
139 index type is not compatible with declaration
140 type of variable is not record
141 type of variable must be file or pointer
142 illegal parameter substitution
143 illegal type of loop control variable
144 illegal type of expression
145 type conflict
146 assignment of files not allowed
147 label type incompatible with selecting expression
148 subrange bounds must be scalar
149 index type must not be integer
150 assignment to standard function is not allowed
151 assignment to formal function is not allowed
152 no such field in this record
153 type error in read
154 actual parameter must be a variable
155 control variable must not be declared on intermediate level
156 multidefined case label
157 too many cases in case statement
158 missing corresponding variant declaration
159 real or string tagfields not allowed
160 previous declaration was not forward
161 again forward declared
162 parameter size must be constant
163 missing variant in declaration
164 substitution of standard proc/func not allowed
165 multidefined label
166 multideclared label
167 undelared label
168 undefined label
169 error in base set
170 value parameter expected
171 standard file was redeclared
172 undeclared external file
173 Fortan procedure or function expected
174 Pascal procedure or function expected

175	missing file "input" in program heading
176	missing file "output" in program heading
177	assignment to function identifier not allowed here
178	multidefined record variant
179	X-opt of actual proc/func does not match formal declaration
180	control variable must not be formal
181	constant part of address out of range
201	error in real constant : digit expected
202	string constant must not exceed source line
203	integer constant exceeds range
204	8 or 9 in octal number
205	zero string not allowed
206	integer part of real constant exceeds range
250	too many nested scopes of identifiers
251	too many nested procedures and/or functions
252	too many forward references of procedure entries
253	procedure too long
254	too many long constants in this procedure
255	too many errors on this source line
256	too many external references
257	too many externals
258	too many local files
259	expression too complicated
260	too many exit labels
300	division by zero
301	no case provided for this value
302	index expression out of bounds
303	value to be assigned is out of bounds
304	element expression out of range
398	implementation restriction
399	variable dimension arrays not implemented

APPENDIX D

DOS 2.5 ERROR MESSAGES

Show below are the know CIO STATUS BYTE values.

NUMBER	DESCRIPTION
01 (001)	Operation complete (No errors)
80 (128)	[BREAK] key abort
81 (129)	IOCB already in use (OPEN)
82 (130)	Non-existent device
83 (131)	Opened for write only
84 (132)	Invalid command
85 (133)	Device or file not open
86 (134)	Invalid IOCB number (Y reg only)
87 (135)	Opened for read only
88 (136)	End of file
89 (137)	Truncated record
8A (138)	Device timeout (doesn't respond)
8B (139)	Device NAK
8C (140)	Serial bus input framing error
8D (141)	Cursor out of range
8E (142)	Serial bus data frame overrun error
8F (143)	Serial bus data frame checksum error
90 (144)	Device done error
91 (145)	Bad screen mode
92 (146)	Function not supported by handler
93 (147)	Insufficient memory for screen mode

NUMBER	DESCRIPTION
AO (160)	Disk drive # error
A1 (161)	Too many open disk files
A2 (162)	Disk full
A3 (163)	Fatal disk I/O error
A4 (164)	Internal file # mismatch
A5 (165)	File name error
A6 (166)	Point data length error
A7 (167)	File locked
A8 (168)	Command invalid for disk
A9 (169)	Directory full (64 files)
AA (170)	File not found
AB (171)	Point invalid

APPENDIX E

ASSEMBLER ERROR MESSAGES

NUMBER	DESCRIPTION
	Syntax Errors
1	Address Error
2	Cannot Include File
3	Format Error
4	Forward Reference in Expression
5	Illegal Use of Conditional Assembly Directive before or
6	Misplaced Else Operator
7	Identifier Expected as Operand
8	Label Required
9	Multiply Defined Symbol
10	Nesting Error
11	Invalid Op-Code
12	Phase Error
13	Questionable Syntax
14	Undefined Symbol
15	Illegal Argument for Conditional Assembly
16	Symbol not in Macro Call Parameter List
17	Directive Requires "on" or "off"
	Fatal Assembler Errors
20	Unknown Error
21	Symbol Table Overflow
22	Lost Label
23	End of File During Macro Definition
24	End of File During Conditional Assembly

APPENDIX F

RUNTIME ERROR MESSAGES

NUMBER	DESCRIPTION
1	Case Index Error
2	Array Index Error
3	Input Error
4	DOS Error
5	Range Error
6	Arithmetic Overflow
7	End of File

INDEX

INDEX

L

M

N

O

P

INDEX

S

T

U

V

W

X

Suggestion Box

We do our best to provide you with complete, bug-free software and documentation. With products as complex as compilers and programming utilities, this is difficult to do. If you find any bugs or areas where the documentation is unclear, please let us know. We will do our best to correct the problem in the next revision of the software. We would also like to hear from you if have any comments or suggestions regarding our product.

To help us better understand your comments please use the following form in your correspondence and mail it to:

Kyan Software Inc., 1850 Union Street #183, San Francisco, CA 94123.

Name _____
Address _____
City _____ State _____ ZIP _____
Telephone:
(day) _____ (evening) _____

Kind of Problem
__ Software Bug
__ Documentation Error
__ Suggestions
__ Other _____

Software Description
Product Name _____
Version No. _____
Date Purchased _____

Kyan Software Products You Use
__ Kyan Pascal
__ Advanced Graphics Toolkit
__ System Utilities Toolkit
__ Other _____

Your Hardware Configuration
Type/Model of Computer _____
How many and what kind of disk drives _____
What is your screen capability: ____ 40 Column ___80 Column
What kind of 80-column adapter? _____
How much RAM? _____K (what kind of RAM Board? _____)
What kind of printer and interface do you use? _____
_____
What kind of modem & interface? _____
Other information about your computer system: _____
_____

What do you use this software for?

___ Education (I am a ___ teacher ___ student)
___ Hobby
___ Professional Software Development
___ Other _____

Problem Description (if appropriate, please include a disk or program listing).

_____
_____
_____
_____
_____
_____
_____
_____

Suggestions

_____
_____
_____
_____
_____
_____
_____
_____
_____
_____
_____
_____