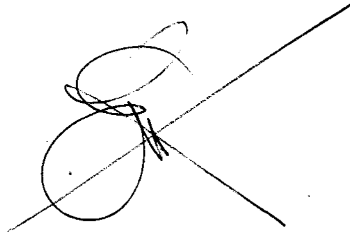
PASCAL 64

1st
FIRST PUBLISHING LTD



PASCAL-64

Complete Pascal Compiler for the COMMODORE 64

First Publishing Ltd Unit 20B, Horseshoe Road
Horseshoe Park
Pangbourne, Berks
Tel: 07357 5244

COPYRIGHT NOTICE

First Publishing Ltd makes this package available for use on a single computer only. It is unlawful to copy any portion of this software package onto any medium for any purpose other than backup. It is unlawful to give away or resell copies of any part of this package. Any unauthorized distribution of this product deprives the authors of their deserved royalties. For use on multiple computers, please contact First Publishing Ltd to make such arrangements.

WARRANTY

First Publishing Ltd makes no warranties, expressed or implied as to the fitness of this software product for any particular purpose. In no event will First Publishing Ltd be liable for consequential damages. First Publishing Ltd will replace any copy of the software which is unreadable if returned within 30 days of purchase. Thereafter, there will be a nominal charge for replacement.

First Printing September 1984
Printed in U.S.A. Translated by Greg Dykema
Copyright (C) 1984 Data Becker, GmbH
Merowinger str. 30
4000 Dusseldorf, W.Germany
Copyright (C) 1984 First Publishing Ltd
Unit 20B, Horseshoe Road
Horseshoe Park
Pangbourne, Berks
Tel: 07357 5244

ISBN 0-948015-071

PREFACE

What is Pascal? Pascal is a modern programming language developed by a Swiss professor (Dr. Niklaus Wirth) as a teaching language for structured programming. It quickly became popular for both teaching and commercial applications. Even among personal computers, Pascal has become a dominating language, loosening BASIC's hold as the 'standard' microcomputer langage. In many schools, Pascal is the standard for information processing and structured programming courses. Now Pascal is available for the most popular of home computers – the Commodore 64.

What has led to the tremendous popularity of this language?

One of the primary reasons is the efficient program structures of Pascal. A new method a programming, structured programming, is built into Pascal. Structured programs are far easier to understand and modify than unstructured programs. Pascal eliminates many programming errors simply through its use of local and global variable definitions in which a local variable name refers only to the program block in which it is declared. An additonal advantage is Pascal's complex data structures. In addition to the basic data types REAL, INTEGER, BOOLEAN, and CHAR data types, data fields (ARRAY), memory-optimized fields (PACKED ARRAY), character strings (STRING), mathematical sets (SETS), data record (RECORD), files (FILE), and pointers (POINTER) are all supported. These data types can be combined as desired to form more complex data structures. With the help of pointer, one can construct linked list, stacks, binary trees, and more. Pascal programs can be written with the built-in Commodore BASIC editor; you need not load in anything else or learn to use another editor.

Pascal 64 contains even more capabilities which make use of the special features of the Commodore 64. Procedures are included for working with high-resolution graphics and sprites. In addition, Pascal 64 allows you to make complete and efficient use of the relative data management capabilities of your disk drive. Pascal 64 allows you to use strings of any desired length. Programs created by Pascal 64 are 10-20 times faster than equivalent programs written in BASIC, or even 30 times faster in some cases because Pascal 64 creates actual machine code. Machine language programs may also be controlled with PEEK, POKE, SYS, and other special commands. Pascal 64 can be used in conjunction with the ASSEMBLER/MONITOR-64 software package available from First Publishing Ltd. Two programs can run concurrently – useful for video games (sprite control) or for controlling interface devices, for example. The possibilities which Pascal 64 offers you are virtually limitless.

TABLE OF CONTENTS

Chapter 1
1.1 Operation of the Pascal 64 compiler 1

Chapter 2
2.1 Organization of a Pascal program 4
2.2 Domain of variables..... 4
2.3 Defining a block..... 5
2 2.4 General syntax..... 5
2.5 Comments..... 5

Chapter 3
3.1 Variable types 6
3.2 Mathematic operations 6
3.3 Program header..... 8
3.4 Variable declaration..... 8
3.5 Constant declaration 9
3.6 Declaration of user-defined types 9
3.7 Arithmetic expressions..... 10
3.8 The Pascal assignment statement 11
3.9 Input and Output..... 11
3.10 Formatted numeric output 13

Chapter 4
4.1 Control structures..... 14
4.2 Loops 14
4.2.1 The REPEAT statement..... 14
4.2.2 The WHILE statement..... 15
4.2.3 The FOR statement..... 15
4.3 Conditional statements 16
4.3.1 The IF statement 16
4.3.2 The CASE statement..... 17
4.4 The GOTO statement 18
4.5 Using control structures..... 19

Chapter 5
5.1 Procedures and functions 21
5.1.1 Procedure declaration 21
5.1.2 Calling procedures 22
5.1.3 Function declaration 23
5.2 Recursion..... 24
5.3 Exiting a procedure 24

Chapter 6
6.1 Data structures..... 26
6.2 ARRAYs..... 26
6.3 RECORDs 27

6.4 Combining RECORDs and ARRAYs 28
6.5 WITH statement 29
6.6 Working with strings 30
6.6.1 Subranges 30
6.6.2 Length of strings 31
6.6.3 Additional string operations 31
6.7 String functions 33
6.8 Packed arrays 34

Chapter 7
7.1 FILEs 36
7.2 Error channel and file status 37
7.3 Input/Output of data blocks 37
7.4 Relative files 38
7.5 Using relative files 39

Chapter 8
8.1 SETs 41
8.2 Representing sets 41
8.3 Set operations 42
8.4 Using sets 43

Chapter 9
9.1 Dynamic data structures (linked lists) 45
9.2 Using pointers 47
9.3 Additional list structures 49
9.4 Combinations of data structures 49

Chapter 10
10.1 Graphics statements 51
10.1.1 Initializing the graphics screen 51
10.1.2 Clearing the graphics screen 51
10.1.3 Setting a point – PLOT 51
10.1.4 Erasing a point – UNPLOT 52
10.1.5 The SPRITE statement 52
10.2 The POKE statement 53
10.3 Executing a machine language program 53
10.4 The high-speed mode – INTEGER 53
10.5 Simultaneous execution of two programs – INTERRUPT 55
10.6 Run-time error messages 57

Chapter 11
11.1 Symbol files for ASSEMBLER/MONITOR 64 58
11.2 Jump to monitor 58

Chapter 12
12.1 Drawing a line 60
12.2 Activating four-color graphics 60
12.3 Setting a point in multi-color mode 61

Chapter 13
13.1 Introduction to structured programming 62

Chapter 14
14.1 An overview of Pascal 64 64
14.1.1 Data types 64
14.1.2 Declarations 64
14.1.3 Procedures and functions 64
14.1.4 Structures 65
14.1.5 Input and Output statements 65
14.1.6 Other statements 66
14.1.7 Assigments 67
14.1.8 Arithmetic operations 67
14.1.9 Standard Functions 68
14.1.10 Access to data structures 68
14.1.11 Predeclared names 69
14.1.12 Special characters for I/O 69

15 BIBLIOGRAPHY 70

CHAPTER 1

1.1 Operation of the Pascal 64 Compiler

-1- Write your Pascal program as you are used to doing for BASIC programs. You must use line numbers as you do for BASIC programs. Correcting and inserting lines is done exactly as in BASIC. This means that you do not have to learn to use another editor, but the following exceptions must be noted:

In Pascal, it is possible that a number must be the first item on a line. This happens with the CASE statement, for example. The BASIC editor is unable to distiguish this number from the the preceding line number, however, and assumes that it is an extension of it. In order to avoid this, a shifted space can be used to separate this number from the line number. Whenever this line is edited, the shifted space must be typed in again because the LIST command prints a normal space. It is better if you do not put a number and the beginning of a line, but use a semicolon instead, followed by a space. This lone semicolon has no effect on the program and is read as a null statement. With the help of this trick, it is also possible to place several spaces at the beginning of a line, making the the structure of the program clearer.

The two BASIC commands REM and DATA affect the BASIC editor. Therefore they may not appear in any form in the program, not even as part of a variable name.

-2- When you are finished writing the program, save it to your work diskette with SAVE"program name",8. Do not under any circumstances use the Pascal 64 distribution diskette or it may be destroyed.

-3- To compile the program, load the compiler from the distribution diskette with

LOAD"0:*",8

and start the program with RUN. When the compiler has finished loading, take the Pascal 64 disk out of the drive and replace it with your work diskette. Should the compiler fail to load properly, your disk drive may not have been initialized properly, although this happens only very rarely. Turn the computer off and then back on and enter the following line:

OPEN 15,8,15,"I":CLOSE 15

After this, start again with the loader program. When the compiler is correctly loaded, it will ask for the name of the program you wish to

compile, followed by the nesting depth of the program. The nesting depth is the number of loops or subroutines which may be nested. The default value given by the compiler must be increased only for very deeply nested programs. Otherwise, simply press the RETURN key. Note: In BASIC, this value is limited to 9 for loops and 23 for subroutines.

-4- Now the compiled machine language program will be written to the work diskette. In its second pass, the compiler will display the line it is currently working on in the top-most screen line.

-5- If the compiler should find an uncompilable line, it responds with COMPILE ERROR IN XXXX and a short description of the error. Some error messages may be somewhat misleading. For example, if a variable name is misspelled, the compiler will respond with "Variable not declared," since it has no way of knowing whether you forgot to declare a variable or if you simply misspelled it. To correct an error, reload your program with LOAD"program name",8; re-edit the program; SAVE it to disk, and continue with step 2 after correcting the error.

-6- Once the program has been compiled successfully, the compiler will display a number representing the end of the program. Make note of this number, as we will need it later. If you want to compile another Pascal program, press the "Y" key. The compiler will automatically restart itself (the distribution diskette is not required). If you do not want to compile another program, press any other key.

-7- Load the loader from the distribution diskette with LOAD"LOADER",8 and start it with RUN 100. The program will ask for the program end (see point 6) and the program name and then loads the machine language program from your work diskette.

-8- The loader responds with "READY." and your Pascal program is completed. It can now be saved to diskette with SAVE; loaded in again with LOAD, or executed with RUN. When using the high-resolution graphics, the compiler reserves 8K of memory for storing the graphics screen. If you do not want to save this memory, enter SYS 3707 before the SAVE command. To later start such a program, use SYS 3748. The file created by the compiler can be deleted with:

OPEN 15,8,15,"S:P/program name":CLOSE 15

-9- It is often necessary to insert an existing routine into a Pascal program. The Pascal 64 command EXTERNAL is used to do this.

The format is:

EXTERNAL "routine name";

No other statements may be on this line. In order to compile a program which contains such a line, you must first use the linker. Load the linker

with LOAD"LINKER",8 and start it with RUN. Give it the name of the program and the name of the program to be created. The linker creates a program which, instead of the EXTERNAL, contains the actual routine so referenced. This routine must be on the same diskette. The line numbers of the routine will be renumbered and the line numbers of the main program will remain the same if there is enough room for the routine. The line numbers are irrelevant to Pascal.

If the routine accessed via EXTERNAL contains an EXTERNAL reference itself, this will NOT be executed and the linker will notify you. Such a routine must first be run through linker before the main program in order to process the nested EXTERNAL.

-10- Linker and loader are independent of the Pascal 64 distribution diskette. The compiler can be loaded from the distribution disk only, but linker and loader may be transferred to your work diskette.

-11- In order to get used to the way the compiler works, it would be a good idea to first read the following chapter and then link and compile the following program. The external routine necessary for this program is found in Chapter 12.

Main program:

```
10 CONST PI=3.14159265;
20 VAR X,Y,FX,FY,A,B,L,I,XL,YL,ST:REAL;
30 EXTERNAL "DRAW";
40 BEGIN A:= -6; B:= 1; ST:= 5; FX:= 12; FY:= 12;
50 ST:= PI/ST;
60 GRAPHIC 1; SCREENCLEAR;
70 L:= -3; WHILE L<=3 DO BEGIN
80 I:= 0; WHILE I<=2*PI DO BEGIN
90 X:= XL; Y:= YL;
100 XL:= (A+B)*COS(I)-L*B*COS((A+B)/B*I);
110 YL:= (A+B)*SIN(I)-L*B*SIN((A+B)/B*I);
120 IF I<>0 THEN DRAW
    (X*FX+160,Y*FY+100,XL*FX+160,YL*FY+100);
130 I:= I+ST; END;
140 I:= I+0.5; END;
150 REPEAT UNTIL PEEK(197)<>64; END;
```

CHAPTER 2

2.1 Organization of a Pascal program

A Pascal program consists of four parts:

-1- Program header
The program header contains the name of the program. For subroutines, the subroutine parameters are also located here. In Pascal 64, the main program header may be omitted.

-2- Variable declarations
Here all variables, constants, types, etc. are defined.

-3- Subroutine declarations
In this section, all of the procedures and functions called from the main program or from other subroutines are defined. The subroutine definition has the same structure as the main program definition, that is, each subroutine has its own variables and subroutines.

-4- Program block
The program itself is located in this part. The program may make use of all of the previously declared global variables (as opposed to variables defined within a subroutine, called local variables) and subroutines. The program must start with BEGIN and end with END
The program block of the main program is called at the start of the program. This block then calls the subroutines. After the END., control is returned to the BASIC interpreter.

2.2 Domain of variables

Because a subroutine is organized the same way the main program is, it can have its own variables and its own subroutines. The variables of a subroutine cannot be used by the main program or a higher-level (outer-nested) subroutine. The subroutine however can make use of the variables of the main program or higher-level subroutine.

In general, we can say that a variable declared at the top of a main program or subroutine becomes undefined or invalid following the END; of that block.

If a variable in a subroutine has the same name as that of another variable in a higher-level subroutine, the higher-level global variable loses its accessability to that variable for the duration of the subroutine in which the conflict occurs and will regain it again once the local variable loses its definition. The value of the global variable is not affected. Therefore you

need not worry about choosing unique names for each of the variables in all of your subroutines. Thus this shortcoming of BASIC is addressed by the Pascal language.

2.3 Defining a block

The program-block portion of a main program or subroutine is delimited by the statemenmts BEGIN and END; (END. for a main program). A block consists of several statements and/or structures (loops, etc.), each separated from the other by a semicolon. Within the structures are one or more blocks. A block consisting of only one statement does not need a BEGIN and END;, as long as this construction makes sense (it does not for program blocks, for instance).

2.4 General syntax

In Pascal, each reserved word (keyword) must be followed by a space. If a keyword requires parameters, they follow this space. A semicolon signals the end of a statement and serves to separate one statement from the next. If a statement does not require a parameter list, the semicolon may be placed immediately following the keyword. No spaces may appear within a list element of parameters, variables, or formulas. List elements are separated from each other by commas, not by spaces. Spaces may not be placed before a semicolon or the semicolon is treated as a null statement and not as a statement separater. the end of a line (carriage return) is treated as a space for the purposes of syntax. Naturally, where one space is allowed, more may be used if desired, in order to show the structure of the program graphically.

Example:

```
READLN (VARIABLE1,VARIABLE2); WRITELN (VARIABLE1,VARIABLE2);
WRITELN;
```

The most common error made by beginners is the improper use of spaces, but the rules can be learned quite quickly because they follow a logical pattern. Pascal 64 is more strict in its syntax than is standard Pascal but is more tolerant in the use of Pascal capabilities.

2.5 Comments

To place a comment in a Pascal program, set it off from the rest of the program text as follows:

```
(* comment *)
```

The compiler will ignore any and all text between the "(*" and the ")*".

The information presented in this chapter will be explained in greater detail in the following chapters.

CHAPTER 3

3.1 Variable types

In Pascal, four basic data types are available:

-1- Whole numbers are denoted by INTEGER and may be values ranging from −32768 to 32767. A decimal point and any digits after will be ignored. INTEGER variables use only a small amount of memory (2 bytes) and can be manipulated quickly. Variables should be of type INTEGER whenever possible to save time and memory.

-2- Real numbers are denoted by REAL and have the same value range and number of significant digits as BASIC floating-point variables. A floating-point number may be in the range +/− 1.70141183E+38. The number of significant digits is limited to 9. Each floating-point number occupies five bytes in memory. In contrast to BASIC, the first character of a real number may not be a decimal point. If the absolute value of the number is less than one, a zero must precede the decimal point:

.00315 must be written as 0.00315

The conversion from INTEGER to REAL and back again is done automatically, if it is required in a calculation and is possible. When converting from REAL to INTEGER, the places after the decimal are simply ignored, as is the case with the DIV and MOD operators and the assignment of a floating-point result to an INTEGER variable.

-3- Alphanumeric characters are denoted by CHAR and CHAR variables may contain any of the 64's printable characters, including color and cursor control characters. Alphanumeric characters are enclosed in quotation marks. The CHAR type may not contain strings; the data structure string is used for this purpose and is explained in chapter 6.

-4- Variables of type BOOLEAN may contain either of two values: TRUE or FALSE. All comparison operations yield a BOOLEAN value. A BOOLEAN value is a statement of the truth of the result of a mathematical operation. Such a result is generally tested by an IF statement. The values TRUE and FALSE may also be assigned to a variable and later tested. This is useful for flags, for instance. The two values TRUE and FALSE are reserved words in Pascal 64.

3.2 Mathematical operations

The following mathematical operations are possible in Pascal 64:

+, −, *, / These operations work the same way as in BASIC and may

be used with either REAL or INTEGER values and variables.

**	This operator corresponds to the BASIC operator ^ and is used for exponentiation. The character ^ is reserved for use with pointers in Pascal (Chapter 9). Standard Pascal does not have an exponentiation operator.
DIV	This operator performs a division but yields an INTEGER value and is therefore executed faster.
MOD	This operation is the counter part of DIV. It yields the mathematical remainder of an integer division.
AND, OR	These operators are the same as their BASIC counterparts, yielding an INTEGER value, and can be used for logical or binary operations, that is, they can be used with two BOOLEAN values to yield a BOOLEAN value. They can also be used with two integers and help with POKE and PEEK operations for setting or testing individual bits, as in BASIC.

The comparison operators >, <, =, >=, <=, and <> also operate as in BASIC and may be used for all three data types. The result of these operations is either −1 = TRUE or 0 = FALSE (a boolean value). Two such results may be combined with AND or OR and yield a result of true or untrue. Loops (REPEAT, WHILE) and IF make use of this conditional. Such a result can also be assigned to a boolean variable and tested later. The function NOT(X) inverts a boolean value, changing TRUE to FALSE or FALSE to TRUE.

In addition, Pascal has the following pre-defined functions:

SIN(X), COS(X), TAN(X), EXP(X), ABS(X), PEEK(X), NOT(X)
correspond to the BASIC functions with the same names. The argument may be of type REAL or INTEGER.

TRUNC(X)	yields the integer portion of a floating-point number and corresponds to the BASIC function INT.
SQR(X)	returns the square of a number.
SQRT(X)	returns the square root of a BASIC function SQR.
LN(X)	returns the logarithm base e of a number.
ARCTAN(X)	returns the inverse tangent of a number and corresponds to the BASIC ATN.
ORD(X)	converts a character to the corresponding number code. Corresponds to the ASC function in BASIC.

CHR(X)	converts a number into a character is the opposite of ORD. It performs the same operation as BASIC's CHR$.
SUCC(X)	yields the successor of a variable which is the value plus one for INTEGER variables.
PRED(X)	Predecessor of a value. SUCC and PRED are generally used for user-defined data types.
ROUND(X)	rounds off a real number to the nearest whole number, is equivalent to TRUNC(X+0.5).
RND(X)	returns a random number between 0 and 1.

Function operands must be enclosed in parentheses; this applies even to the NOT function.

If your favorite function is missing, it can be implemented through a function or formula library:

```
COT(X) :       1/TAN(X)
ARCSIN(X) :    ARCTAN(X/SQRT(1-X*X))
COSH(X) :      (EXP(X)+EXP(-X))/2
LOG10 :        LN(X)/LN(10)
```

In Pascal, it is possible to define your own functions. Chapter 5 explains how to do this.

All four data types can appear as a variable, variable field, constant, or direct number in mathematical expression.

All variables used in a program or subroutine must be declared before the start of the program (see Chapter 2).

3.3 Program header

The program header is the first part of a program and may be omitted in Pascal 64. Most of the programs in this manual do not have headers.

Format:

```
PROGRAM program name;
```

3.4 Variable declaration

The keyword VAR tells the compiler that you wish to declare variables. The variable names are separated from each other by commas; their type is

indicated after a colon, and a semicolon followed by a space separates this from the next variables.

Example:

```
VAR    A,B,TEST:REAL; X,COUNTER:INTEGER; A:REAL; M,N:CHAR;
```

A declared variable has no default value, not even zero. You must assign a value to each variable in your program before its contents are accessed, otherwise you will receive a random value. The length of the variable names and the number of variables is limited only by memory.

To define an array of a certain type, the variable type is simply replaced with ARRAY[number..number] OF variabletype;

```
VAR A,B,TEST:ARRAY[-5..1000] OF REAL;
```

This line defines the arrays A, B, and TEST with element number running from −5 to 1000. This corresponds to the DIM command in BASIC. The indices of the array must be placed within square brackets. Note that arrays of arrays are allowed and can be used to define multi-dimensional arrays (up to four dimensions). Note also that the indices need not start at zero but can start and end with any integer. This is described in Chapter 6 in greater detail.

3.5 Constant declaration

The keyword CONST is placed before a list of constants that are to be used in a program. A constant is a name that is assigned a fixed value for the duration of the program. The value of a constant cannot be changed during program execution. Examples are:

```
CONST PI=3.14159265; YES=-1; NO=0; PAREN="(";
```

The data type is automatically determined by the compiler. A constant may not be assigned a new value within the program.

3.6 Declaration of user-defined types

In addition to the pre-defined data types, Pascal allows you to define your own, using the TYPE command:

Example:

```
TYPE COLOR=(BLUE,RED,GREEN,YELLOW,WHITE,BLACK)
```

Now variable of this type can be declared:

```
VAR CARPETCOLOR:COLOR;
```

and values assigned to them:

CARPETCOLOR:= GREEN;

The functions SUCC and PRED can also be used with these data types.

SUCC(GREEN)=YELLOW is a TRUE statement.

These newly-defined data can appear anywhere within a formula other numbers or constants may appear.

3.7 Arithmetic expressions

Writing an expression in Pascal is done the same way as in BASIC. A set of operations, using algebraic hierarchy, all variables and data types and unlimited use of parentheses permitted in Pascal. The following exceptions must be noted:

- A space must be placed before and after the operators AND, OR, MOD, DIV, and IN so that the compiler does not think these operators are part of a variable name. Variable names are permitted to contain these operators meaning COLOR is a legal name although it contains the OR operator. Because the space in front of these operators may be replaced by a carriage return, an expression may span several lines. The space after these operators may not be replaced by a carriage return, however.
- No other spaces may appear within an expression.
- The index of an array is placed within square brackets and may be an expression. Indices may also be negative values if such has been declared.
- User-defined functions may be used as pre-defined functions; function arguments may be constants, variables, or expressions.
- The negation of a variable using the unary minus operator is not allowed:

  B:= -B; must be replaced by B:= B*-1;

  The calculation hierarchy in Pascal is slightly different than that in BASIC:

- The parentheses have the highest priority. This also applies for function parameters, which must be enclosed in parentheses.
- Then the operators *, /, **, DIV, MOD, and AND are executed.
- Next, the operators +, -, and OR have precedence.
- Last, the comparison operators (<, >, =, <>, <=, >=) and the set operator IN are executed. Chapter 8 contains more information about sets.

The logical operators AND and OR do not have the equal priority in this hierarchy. This is according to the Pascal standard, however, and is therefore implemented in Pascal 64. It is recommended that parentheses be used in expressions involving these operators to avoid errors.

3.8 The Pascal assignment statement

In order to assign a value to a variable, one uses the following assignment instruction:

```
variable:= expression;
```

or for arrays:

```
variable[expression1]:= expression2;
```

The variables on the left side are assigned the result the of the expression on the right side. This assignment is similar to the assignment in BASIC. The primary difference is that in Pascal, a colon is required before the equal sign in order to distiguish it from the equality comparison operator. In addition, a space must follow the equal sign because the assignment operator is a Pascal command and a reserved word (see Chapter 2).

Example:

```
FIELD[3+4]:= SIN(8);
```

The sine of 8 is assigned to the seventh element of the array, assuming the array is of type REAL.

3.9 Input and Output

In order to read a number or character from the keyboard, one uses the READ command:

```
READ (variable1,variable2,...,variableN);
```

This command reads values into the variables separated by commas. When reading numbers, the values must be separated from each other by a space or RETURN. If the variable is of type CHAR, only one character is read.

The READLN statement may be used instead of the READ statement. The READLN statement waits for a RETURN at the end of the input:

READLN (X,Y);

If the following input is entered:

12 32.5 4 <RETURN>

then the first two numbers are assigned to the variables X and Y, respectively, the number 4 is ignored, and the cursor moves to the next line. The READ statement saves the number 4 for the next input.

To output results, Pascal has the WRITE statement:

WRITE (variable,expression,"text",...);

It is possible to output the result of any desired expression with the WRITE statement. In addition, text placed within double quote marks will also be printed. Text may be comprised of any characters which can be represented within the double quote marks (such as cursor control characters). If an array of type CHAR is output without specifying the index, the entire array is printed as a string (Chapter 6). With values of type BOOLEAN or of a user-defined type, the corresponding number is printed.

The WRITELN statement is just like the WRITE statement, except that it places a RETURN at the end of the output. To print just a RETURN, use the WRITELN statement without parameters:

WRITELN;

Corresponding BASIC commands are:

```
READLN          : INPUT
WRITELN         : PRINT
WRITE           : PRINT ... ;
```

When entering multiple numbers, they must be separated by spaces, not by commas as in BASIC.

Unfortunately, the kernal of the Commodore 64 contains a small error. If an input takes place on the same line as additional input or some output such that the input exceeds the end of the line, the entire line is read as input. The following BASIC program illustrates this:

```
10 INPUT"Text longer than 40 characters";A$:PRINTA$
```

Not only is the data assigned into the string variable A$, but the prompt text also become part of A$ as well. Because this error is not in the BASIC interpreter but in the kernal, it also occurs in Pascal 64.

The READ statement can be used with a variable of type CHAR to read individual characters. The cursor appears on the screen and characters are

read until RETURN is pressed. To avoid this, the GET statement can be used instead:

Format:

GET variable;

This statement corresponds to the BASIC GET command. It fetches a character from the keyboard buffer and returns it in the given variable. If the keyboard buffer is empty, a CHR(0) is returned.

3.10 Formatted numeric output

In Pascal it is possible to output numbers in a uniform format. This is permitted on the screen or printer (chapter 7) using the WRITE or WRITELN statements.

Format:

WRITELN (expression:value1:value2);

The two values may be numbers or variables. The first value gives the total length of the output. If the output is shorter than this length, the output is right-justified within a field of value1 spaces. The second value gives the number of places after the decimal. Numbers with an exponential portion (numbers that must be represented in scientific notation) cannot be formatted in this manner. To line up decimal points on a column of numbers, chose the same values for all of the output. The decimal point is placed at value1−value2. For this reason, the first value must always be greater than the second.

With the commands we have discussed so far, it is possible to write a small Pascal program:

```
 5 PROGRAM FIRST;
10 VAR INPUT,RESULT:REAL;
20 (* NO SUBROUTINES NECESSARY *)
30 BEGIN (* START OF THE MAIN PROGRAM *)
40 WRITELN ("I AM WAITING FOR A NUMBER.");
50 READLN (INPUT);
60 RESULT:= SIN(INPUT);
70 WRITELN ("THE SINE OF THE INPUT IS:",RESULT:10:4);
80 END. (* END OF THE MAIN PROGRAM *)
```

This program can be compiled and executed as described in Chapter 1.

CHAPTER 4

4.1 Control structures

Loops and conditional branches are used to control the order in which program statements are executed. You are already familar with one type of loop from BASIC: the FOR ... NEXT loop. You are also acquainted with IF ... THEN statement for conditional branching. This branch is limited to a single line in BASIC. In Pascal, the end of the line is treated as a space, making it possible for the THEN portion of an IF ... THEN statement to span multiple lines. The BEGIN and END; markers are used to set off the block which are executed if the condition is true (the THEN portion). In Pascal, a block demarcated by BEGIN and END; may appear anywhere a single statement is allowed.

BASIC also has the capability to force and unconditional branch with the GOTO command. Because of Pascal's efficient structured commands, this unstructured statement is unnecessary and contradicts the concept of Pascal. For the sake of completeness, it is included anyway, although it should be used only very rarely.

4.2 Loops

4.2.1 The REPEAT statement

Format:

REPEAT block UNTIL condition;

REPEAT and UNTIL replace the block markers BEGIN and END.

Operation:

The statements of "block" are executed, the condition (comparison or any other expression which returns a boolean value) is checked, and the statements of "block" are re-executed provided that the condition has not been satisfied (is FALSE). The block is executed until the condition is satisfied (TRUE) at which time execution resumes with the statements following the UNTIL.

Example:

```
10 VAR COUNTER:INTEGER;
20 BEGIN COUNTER:= 1;
30 REPEAT
40 WRITELN (COUNTER);
50 COUNTER:= COUNTER+1;
60 UNTIL COUNTER>10;
70 END.
```

This program prints the numbers 1 through 10. The number 11 is not printed since the condition is checked first and found to be satisfied. Note that the variable COUNTER is first defined before it is used.

4.2.2 The WHILE statement

Format:

WHILE condition DO block;

The condition is checked and the block is executed only if the condition is TRUE. At the end of the block, the condition is again checked and the block re-executed depending on its truth value. If the condition is false, execution continues with the statement following the block. In contrast to the REPEAT/ UNTIL structure, the block is not executed at all if the condition is false to begin with, whereas the block is always executed at least once with REPEAT/ UNTIL.

Example:

```
10 VAR COUNTER:INTEGER;
20 BEGIN COUNTER:= 1;
30 WHILE COUNTER<11 DO BEGIN
40 WRITELN (COUNTER);
50 COUNTER:= COUNTER+1;
60 END; (* END OF THE LOOP *)
70 END.
```

This program also prints the number 1 through 10. If the counter equals 11, the condition is no longer satisfied and the block is skipped.

4.2.3 The FOR statement

Format:

FOR variable:= start value TO end value DO block;

The variable is initially assigned the start value, the block is executed, and the variable is incremented by one. The block is again executed and the variable incremented until the variable is greater than the end value immediately following its incrementation. If the start value is larger than the end value at the start of the loop, the block is not executed. Program execution continues once the end value is exceeded. The start and end values may be expressions and the variable must be of type INTEGER. The end value is recalculated each time through the loop, allowing it to be changed within the loop.

The loop can also be made to count down:

FOR variable:= start value DOWNTO end value DO block;

The variable is decremented and the block is skipped when its value is smaller than the end value.

Example:

```
10 VAR COUNTER:INTEGER; BEGIN
20 FOR COUNTER:= 1 TO 10 DO
30 BEGIN WRITELN (COUNTER); END;
40 END.
```

Again we have a program which prints the number 1 through 10. By using a FOR loop, the initial assignment of the counter, the incrementation of the variable, and the checking of the condition are all done automatically. The WHILE command must be used to increment the variable by a number other than +1 or −1. If the block consists of only one statement, the BEGIN and END may be omitted.

As we mentioned before, structures may be nested within a block. The nesting depth is arbitrary, but may need to be increased at compile time (see Chapter 1).

As you can see, Pascal offers several different ways of formulating a program. The best method depends on the problem at hand. You should become familiar with all of the types of loops. REPEAT and WHILE can be used for more than just counting loops. The following program waits for a keypress:

```
10 BEGIN
20 REPEAT
30 UNTIL PEEK(203)<>64;
40 END.
```

Memory location 203 contains the code for the currently pressed key. If no key is being pressed, the location contains 64. The memory addresses 0 through 1023 are the same in Pascal as they are in BASIC. Other system addresses are also retained (such as the sprite registers or the address of the screen memory).

4.3 Conditional statements

4.3.1 The IF statement

Format:

IF condition THEN block1; ELSE block2;

or

IF condition THEN block1;

If the condition is fulfilled (TRUE), block1 will be executed, otherwise block2 will be executed, if it exists.

Example:

```
10 VAR INPUT:REAL;
20 BEGIN
30 READLN (INPUT);
40 IF INPUT<5 THEN
   BEGIN WRITELN ("THE INPUT IS SMALLER THAN 5"); END;
50 ELSE BEGIN
   WRITELN ("THE INPUT IS GREATER THAN OR EQUAL TO 5"); END;
60 END.
```

Since the blocks in this example are only one statement long, the BEGIN and END may be omitted.

This program makes only a rough statement about the variable INPUT. The following instruction allows a more exact statement about a variable or expression:

4.3.2 The CASE statement

Format:

CASE expression OF
number,number,number,...,number: block1;
number,...,number: block2;
number,...,number: blockN;
END;

The numbers must be of type INTEGER, although characters of type CHAR may also be used instead if the expression yields such a result. Floating-point numbers are not allowed.

Operation:

The expression is evaluated and the block associated with the number or character equal to the result of the expression is executed. The END; denotes the end of the CASE list and belongs to the CASE statement, not to a block. If no match is found for the result of the expression, execution continues with the statement immediately following the END; of the CASE statement.

Example:

```
10 VAR INPUT:INTEGER;
20 BEGIN
30 READLN (INPUT);
40 CASE INPUT OF
50 ; 1,2,3: BEGIN WRITELN ("1 TO 3"); END;
60 ; 4,5,6: BEGIN WRITELN ("4 TO 6"); END;
70 ; 7,8,9,10,11: BEGIN WRITELN ("7 TO 11"); END;
80 END; (* OF CASE *)
90 END. (* OF PROGRAM *)
```

The semicolon at the start of each line serves only to separate the line numbers from the numbers in the CASE statement.

This program gives a more exact description of the contents of the variable INPUT than the previous program.

4.4 The GOTO statement

Format:

```
GOTO label;
 .
 .
 .
label: statement;
```

As soon as the program encounters a GOTO statement, the execution of the program is interrupted and resumes at the label. The jump need not be forward. The label can be a number or a name. It is recommended that the label contain information on the line numbers between which the jump occurs so that the program flow may followed easily. Each label may be jumped to from only one GOTO.

Example:

```
10 VAR COUNTER:INTEGER;
20 BEGIN COUNTER:= 1;
30 FROM60TO30:
40 WRITELN (COUNTER);
50 COUNTER:= COUNTER+1;
60 IF COUNTER<11 THEN GOTO FROM60TO30;
70 END.
```

This program also outputs the numbers from 1 to 10, but this cannot be seen directly from the program text, only by following the jump. You should use the GOTO only in special cases since it can be avoided in virtually all

instances, although this may not be obvious to a BASIC programmer. The elimination of GOTO will make your programs easier to understand and modify.

In Pascal, the destination of a jump is limited. A jump into a block is not allowed. A jump within the same block is permitted, as is a jump to the outside of a block. JUmping out of a subroutine or CASE structure is stricly forbidden. To exit a subroutine, it suffices to jump to the end of the routine. There are similar restrictions in BASIC. Exiting a subroutine with GOTO can result in a stack overflow and a jump into a loop results in a NEXT WITHOUT FOR.

4.5 Using control structures

In order to illustrate a practical use of control structures, we present the following program which prints the prime numbers under 20000. The program uses the principle in which all of the multiples of prime numbers are removed from a table because they are no longer prime numbers. The first number will take a fairly long time to calculate. The important part is to note the nesting of the blocks.

All END statements are followed by a comment which indicates to which structure they belong. All BEGIN and END statements which are optional are shown as comments.

```
10 CONST N=10000;
20 VAR Z:PACKED ARRAY[0..10000] OF BOOLEAN;
30 K,I:INTEGER;
40 BEGIN
50 FOR I:= 1 TO N DO
60 ; (* BEGIN *)
70 ; Z[I]:= TRUE;
80 ; (* END OF FOR *)
90 WRITELN (2);
100 FOR I:= 1 TO N DO
110 ; (* BEGIN *)
120 ; IF Z[I] THEN
130 ; BEGIN
140 ; WRITELN (2*I+1)
150 ; K:= 1;
160 ; WHILE I+K*(2*I+1)<=N DO
170 ;    BEGIN
180 ;    Z[I+K*(2*I+1)]:= FALSE;
190 ;    K:= K+1;
200 ;    END; (* OF WHILE *)
210 ; END; (* OF THEN *)
220 ; (* END OF FOR *)
230 END. (* OF THE PROGRAM *)
```

Type this program in and compile it as described in Chapter 1. It runs significantly faster than a corresponding BASIC program even though it executes a complex index calculation in order to save space. Enter the following line:

```
45 INTEGER;
```

and recompile the program. The program should run even faster than before. We will say more about this command in chapter 10 so do not use in other programs until then.

CHAPTER 5

5.1 Procedures and functions

The procedure definitions in Pascal are placed before the main program and after the variable declarations. They consist of a parameter portion, the procedure variable declarations, nested procedure declarations, and the procedure block. This construction is similar to that of the main program.

5.1.1 Procedure declaration

Format:

```
PROCEDURE name (parameter list);
variable declarations
nested procedure declarations
block;
```

More procedure declarations may follow the END; statement of the block. The parameter list, the variable declarations and any nested procedure declarations are all optional. If a procedure has no parameters, the semicolon is placed immediately following the name.

Variable declarations were previously explained in Chapter 2. The declaration of a procedure nested within a procedure follows the same pattern as the declaration of a procedure of the main program. An outer procedure's variables belong to both the outer procedure and the nested procedure. Variables of the main program can be used by all procedures as well as the main program. For this reason they are called global variables. Each program block should use only its own variables in order to avoid confusion. Passing data between procedures and the main program should be accomplished via parameters, variables which the main program or calling routine defines are declared in the parameter list.

There are two types of parameters:

Value parameters:
: These parameters are declared as variables, but without the word VAR. When the procedure is called, the value parameters are given a value by the calling program.

Variable parameters:
: The word VAR in front of a declaration is used to distiguish variable parameters from value parameters. A variable parameter is passed by the PROCEDURE call, and can be manipulated under the name of the variable parameter. It is thereby possible for the procedure to use variables of the calling program without the need for these variables to

have a specific name. Each call of the procedure can pass a different variable. If the value of a variable parameter is changed by a procedure, the value of the variable in the calling program is also changed. Variable parameters offer the capability of passing results to the calling routine.

The parameters of a procedure can also be ARRAYs or RECORDs. These data types must be declared as variable parameters. More about arrays and data records in found in the next chapter.

Example:

PROCEDURE TEST (VAR A,B:INTEGER; C:REAL; VAR X,Y:CHAR);

The variables A, B, X, and Y are variable parameters while the variable C is a value parameter. All parameters can be used by the procedure as variables. The difference between parameters and variables declared within the procedure is that parameters have a value defined by the calling routine, whereas the procedure variable are initially undefined.

5.1.2 Calling procedures

Format:

name (parameter1,parameter2,...,parameterN);

The values of the parameters are passed to the named procedure and this is then executed. Variable parameters may only be variables, not expressions, due to their nature.

Example:

TEST (I,K,COS(3*5+Q),S,N);

A procedure may also call other procedures. If the called procedure does not belong to the calling procedure, it must be previously declared. A procedure name may not have the same name as a Pascal reserved word.

Here is an example of a procedure named INPUT which has a parameter as a real variable:

```
10 VAR X:REAL;
20 PROCEDURE INPUT (VAR NUMBER:REAL);
30 (* NO PROCEDURE VARIABLES NEEDED *)
35 (* NO NESTED PROCEDURES NEEDED *)
40 BEGIN
50 WRITELN ("PLEASE ENTER A NUMBER");
60 READLN (NUMBER);
70 END; (* PROCEDURE *)
80 BEGIN (* OF THE PROGRAM *)
90 INPUT (X);
100 WRITELN ("YOU ENTERED ",X,".");
```

The procedure works like a new statement. It would be possible, for instance, to expand the graphics statements of Pascal 64 using procedures (see Chapter 12).

In many cases, it also desirable to be able to define a new function. Pascal offers us this capability.

5.1.3 Function declaration

Format:

FUNCTION name (parameter list):function type;
variable declarations
subroutine declarations
block;

Everything that was said about procedures also applies to functions, except that the parameter list must contain at least one element. In addition, a funtion variable is defined having the name of the function and the type given as function-type. The type may be any of the four basic data types or a one-dimensional array. The ability to use arrays is important for writing string functions, for instance.

The function variable must be assigned a value, which represents the result of the function, within the function block. Otherwise, the function variable can be treated as a normal variable. For arrays, the individual fields must be assigned.

Calling a function is done the same way pre-defined functions are called. The parameter passing works in the same as for procedures.

Here is an example of a function named FACTORIAL. It computes the factorial of a value-parameter that is passed to it (N) and returns this as a real number. The factorial of a positive integer n is defined as $n(n-1)(n-2)(n-3) \ldots (3)(2)(1)$.

```
10 VAR INPUT:INTEGER;
20 FUNCTION FACTORIAL (N:INTEGER):REAL;
30 VAR COUNTER:INTEGER;
40 BEGIN
50 FACTORIAL:= 1;
60 FOR COUNTER:= 1 TO N DO
70 FACTORIAL:= FACTORIAL*COUNTER;
80 END; (* OF THE FUNCTION *)
90 BEGIN
100 READLN (INPUT);
110 WRITELN (FACTORIAL(INPUT));
120 END.
```

5.2 Recursion

In Pascal, it is possible for a procedure or function to call itself. This concept is called recursion. A recursive routine must of course have some means of determining when to stop calling itself based on when a certain condition is met. In contrast to BASIC, the variables initialized in or passed to one procedure call are not destroyed by successive calls. After returning from a call, the variables are returned to their original condition. Each new call generates new local variables which disappear again after each return. The possibilities which local/global definitions permit are shown clearly in recursion. Many programs which would normally require a stack to manage data which can be written using recursion instead. One example of recursion is the number of moves ahead which a chess program calculates. The new generation of variables applies to all variables in the subroutine, even for arrays and complex data structures.

This is an example of a recursive function to calculate factorial. As you recall, the factorial of n is defines as n(n−1)(n−2)...(3)(2)(1). Here we calculate n factorial by first calling FACTORIAL with the parameter n, which then calls itself with a new parameter, (n−1), until the parameter passed to the function is equal to 1.

```
10 VAR INPUT:INTEGER;
20 FUNCTION FACTORIAL (N:INTEGER):REAL;
30 BEGIN
40 IF N>1 THEN FACTORIAL:= FACTORIAL (N-1)*N
50 ELSE FACTORIAL:= 1;
60 END;
70 BEGIN
80 READLN (INPUT);
90 WRITELN (FACTORIAL(INPUT));
100 END.
```

Since the parameter n is decremented with each new call, it will eventually reach the limiting or ending value (1). The recursion ends and all of the n's from the various levels will be multiplied together.

Recursion can also be set up by having a subroutine call a higher-ordered subroutine and having this subroutine call the lower-level routine. Recursion can also become even more complex, involving multiple subroutines.

5.3 Exiting a procedure

Under certain conditions, it may happen that the rest of a procedure need not be executed, such as when an output subroutine determines that an output device is not connected. Naturally, this problem could be solved with an IF-THEN-ELSE condition. It is simpler however to use the command EXIT.

Format:

```
EXIT;
```

This statement causes the current procedure to stop execution and forces a return to the calling routine. When used in the main program, EXIT causes the program to end. Since EXIT corresponds to a GOTO to the end of the procedure, it should be thought in much the same manner as the GOTO, although it is somewhat clearer.

CHAPTER 6

In the previous chapter we presented all of the basic elements of Pascal so that you are now ready to develop your own Pascal programs. Chapter 13 will assist you with this. Now, however, we turn to the more advanced features of Pascal.

6.1 Data structures

In Pascal we have the ability to structure not only programs but data as well. A data structure consists of several elements. The data type of these elements may be the same throughout the structure (ARRAY) or may be different from element to element. It is important to note that the elements of a structure do not have to be one of the four basic types, but can themselves be structures. Because a data structure consists of several elements, exactly as a program block consists of several statements, it seems reasonable to refer to data structures as data blocks. Pascal offers many possibilities for working with data blocks, so in the following chapters, we will refer to an arbitrary data structure as a data block. For example, the statements for manipulating strings can also be used for other structures despite the fact that they were not specifically written for these structures.

6.2 ARRAYs

An array can be composed of elements of any desired type of data. You are already familiar with arrays from BASIC (DIM command, etc). Each array must be declared before it can be used as a variable:

```
VAR name:ARRAY[number1..number2] OF data type;
```

You have already seen this declaration in Chapter 3. The two numbers in the declaration give the minimum and maximum index of the array. In contrast to BASIC, the minimum index need not be zero. In fact, both number may be negative. Number1 must of course be smaller than number2. Theoretically, the numbers may be any desired integers, but the practical limit is the amount of memory available for variable storage. Special attention must be paid to this for arrays of type REAL since real variables require 5 bytes each. The data type can be any desired data type, even another array:

```
VAR ARR3D:ARRAY[1..10] OF ARRAY[2..5] OF
                ARRAY[-3..4] OF REAL;
```

To save space, this can also be written:

```
VAR ARR3D:ARRAY[1..10,2..5,-3..4] OF REAL;
```

An array can have up to four dimensions. An array element is accessed by placing the index in square brackets behind the array name. The index

may be an expression. The array element selected in the manner may be used as normal variable:

ARR3D[X,Y,Z]

This is how an element of the array is accesed. The individual element is of type REAL can be manipulated as any REAL variable. A three-dimensional array is a two-dimensional array of one-dimensional arrays. One can access an element of this two-dimensional array and receive as a result a single-dimensional array:

ARR3D[X,Y]

This gives us a data block with contains all of the array elements having the first dimension X and second dimension Y. The data block is a one-dimensional array. The following expression yields a two-dimensional array:

ARR3D[X]

and the entire three-dimensional array is accessed with:

ARR3D

All operators which apply to data blocks may be used on such arrays. One such operator is the assignment operator:

```
TEMPARRAY:= ARR3D;

TEMPARRAY[1,2]:= ARR3D[4,5];
```

Naturally, only data blocks having the same size and index range (that is, the same type) may be assigned to each other. Another possibiltiy is that data blocks may be transmitted to subroutines as parameters. Data blocks can also be written to and read from files (Chapter 7).

6.3 RECORDs

Pascal's abilities to work with data blocks do not end with arrays. For example, an address list contains not only characters but also numbers. This combination of several different data types is difficult to represent in a single array. Pascal offers the ability to combine several data types into one cohesive unit (RECORD). The individual variables are set off by RECORD and END; and represent a new data type.

```
VAR name,name,...,name:RECORD
                           name1:data type;
                           name2:data type;
```

```
        .
        .
        .
        nameX:data type;
      END;
```

The data types of the variables may be any of the basic four, they may be arrays, and even other records. The following declaration is possible:

```
VAR ADDRESS:RECORD
        NAME:RECORD
            FIRSTNAME:ARRAY[1..10] OF CHAR;
            LASTNAME:ARRAY[1..10] OF CHAR;
        STREET:ARRAY[1..15] OF CHAR;
        CITY:ARRAY[1..15] OF CHAR;
        STATE:ARRAY[1..2] OF CHAR;
        ZIPCODE:INTEGER;
        END;
```

An individual record element is accessed by placing a period followed by the name of the element behind the record name. The following types of references can be made:

ADDRESS.ZIPCODE	is of type INTEGER
ADDRESS.CITY	is an array of type CHAR
ADDRESS.NAME.FIRSTNAME	is an array of type character
ADDRESS.NAME.LASTNAME[1]	is the first character of the last name
ADDRESS.NAME	is a two-element RECORD
ADDRESS	contains all of the address data

These parts of the record can be manipulated as desired since they either refer to other records or are actually variables. Data blocks can again be assigned to each other:

```
ADDRESS.NAME.FIRSTNAME:= ADDRESS.NAME.LASTNAME;
```

Records may also serve as subroutine parameters. All operations possible with data blocks of type ARRAY are also possible with records.

6.4 Combining RECORDs and ARRAYs

Since we can make an array of any data type, we can also make arrays of records. Several records of the same type can thereby be brought together in an array:

```
VAR NAMELIST:ARRAY[1..100] OF
        RECORD
        FIRSTNAME:ARRAY[1..10] OF CHAR;
        LASTNAME:ARRAY[1..10] OF CHAR;
        END;
```

Records and arrays may be nested inside each other as desired. The nesting of records is abitrary, the nesting of arrays is limited to four dimensions. If the last data type of a record is not a basic data type but always another record, only three dimensions may be used. It should be noted that an individual element of such a structure is accessed first by the record element and then by array element:

NAMELIST.FIRSTNAME[10]	the last name of the tenth person
NAMELIST[11]	both names of the 11th person
NAMELIST	the entire file
NAMELIST.LASTNAME[20,2]	the second letter of the last name of the 20th person

A possible use of this file might be for sorting names alphabetically because the following assignment is possible:

```
NAMELIST[X]:= NAMELIST[Y];
```

Records can be used for more than just addresses. They may be used for any group of data which have something in common or just belong together. A typical use might be the three coordinates which denote a point in a three-dimensional Cartesian coordinate system. Another mathematical use is for storing complex numbers.

6.5 WITH statement

In many cases, several successive references have to made to the same record. Each time the record is accessed, the record name must be given. This is made easier if we use the WITH statement.

Format:

```
WITH record name DO BEGIN statements; END;
```

Example:

```
READLN (NAMELIST.FIRSTNAME[1],NAMELIST.LASTNAME[1]);
NAMELIST.FIRSTNAME[2]:= NAMELIST.FIRSTNAME[1];
```

This can be shortened to:

```
WITH NAMELIST DO BEGIN
READLN (FIRSTNAME[1],LASTNAME[2]);
FIRSTNAME[2]:= FIRSTNAME[1];
END;
```

The compiler places the record name in front of all variables, providing doing so does not cause an error. If several WITH commands are nested, the

compiler places both record names in front of the variable names. A nested WITH construction must correspond to a nested RECORD structure.

Example program:

```
10 VAR DFILE:ARRAY[1..20] OF RECORD
20 FIRSTNAME:ARRAY[1..10] OF CHAR;
30 LASTNAME:ARRAY[1..10] OF CHAR;
40 END;
50 I:INTEGER; CHOICE:CHAR;
60 BEGIN
70 WITH DFILE DO BEGIN REPEAT
80 WRITE ("O=OUTPUT, I=INPUT"); REALN (CHOICE);
85 WRITE ("RECORD?"); READLN (I);
90 CASE CHOICE OF
100 "O": BEGIN WRITELN (FIRSTNAME[I],LASTNAME[I]); END;
110 "I": BEGIN READLN (FIRSTNAME[I],LASTNAME[I]); END;
120 END;
130 UNTIL CHOICE="*"; END; END;
```

6.6 Working with strings

6.6.1 Subranges

In standard Pascal, a string is represented as an array of characters. Since such an array represents a data block, we already have operators for strings. In many cases, however, we want to work with just part of the string, not the whole thing. For this reason, Pascal 64 gives you the ability to form subranges. To specify a subrange, you use the starting and ending values, separated by two periods, instead of the array indices:

array[start value..end value]

The starting and ending values may be expressions. Since the subrange defined in this manner is itself a data block, all of the data block operations may be performed on it. A section can be formed from any array in any dimension. The individual elements need not be one of the four basic data types, although they are most often of type CHAR. In all cases, the section must be the last dimension given. No additional dimension indices may follow, even if the array has additional dimensions. Array subranges may also be formed from ARRAY/RECORD combinations. The following examples demonstrate the possibilities which are possible for string manipulation:

The following arrays are defined:

VAR STRING1,STRING2:ARRAY[1..1000] OF CHAR;

To insert a character into a string, one uses the following statements:

STRING1[55..1000]:= STRING1[54..999];
STRING1[54]:= "A";

The string structure from 54 on is moved up by one memory location and the character "A" is placed in the vacated location at 54. An entire string array may be inserted in the same manner.

To delete the character again, one would write:

```
STRING1[54..999]:= STRING1[55..1000];
```

The order in which the arrays subrange are specified in the assignment determines in which direction the shift will take place. In the first example, the upper subrange was specified first, meaning that the shift direction is up. In the second example, the shift direction is down since the lower subrange is specified first. Many versions of Pascal have similar expansions of standard Pascal for string management, but the shift direction must usually be specified explicitly (MOVELEFT, MOVERIGHT). In Pascal 64, the compiler determines this automatically.

Data block assignments offer even more capabilities:

```
STRING2[1..50]:= STRING1[50..99];
```

The first 50 elements of the array STRING2 are assigned the given subrange of STRING1. The ending value, and with it the length of the subrange, is given twice in the assignment. This is not absolutely necessary. The compiler is only interested in the length of the subrange to be manipulated:

```
STRING2[1..0]:= STRING1[50..99];
```

Since the array STRING2 starts with the index 1, we can shorten the assignment still more:

```
STRING2:= STRING1[50..99];
```

With the statements and techniques we have learned so far, it would not be difficult to write a simple word processor. We have already shown examples of the two basic functions of a word processor, insertion and deletion. Although moving entire array sections is done faster than moving characters element by element with a loop, the speed of the 64 is not limitless. With very large strings (20000 bytes), inserting a character may take several seconds. It is a good idea not to insert characters one at time for this reason.

Subranges of arrays are permitted wherever any other type of data block is allowed. One exception is the variable parameter of a subroutine, since only entire data structures may be passed to subroutines.

6.6.2 The length of strings

Since not all of the characters of a string may be used immediately, it is a good idea to have a marker for the end of the string. Pascal 64 uses CHR(0)

and fills all unused elements of a string with this character when inputting from the keyboard. When printing to the screen or printer, only the characters preceding the CHR(0) are displayed and anything after the CHR(0) is ignored. The function LENGTH can be used to determine the length of a string:

Format:

LENGTH(data block)

This function returns the number of elements in the array preceding a CHR(0). Although it may be used with other data blocks, it has meaning only for strings.

Example:

STRING[1..LENGTH(STRING)]

returns all of the elements of the array up to the first CHR(0).

STRING[1..LENGTH(STRING)+1]

returns all of the array elements up to and including the first CHR(0) encountered. In this case, the CHR(0) MUST be present.

Because each newly assigned string has an end and does not necessarily occupy the entire array, programs must always check for and include a CHR(0) at the end of a string. Except for input, a CHR(0) is not automatically placed at the end of a string.

6.6.3 Additonal string operations

The following statement is used to fill an array with a character:

FILLCHAR (data block,character);

If the character chosen is CHR(0), all elements of the block are cleared and considered to be defined.

The STR statement serves to convert a number into a string:

STR (expression,data block);

The result of the expression is converted into a character string. The data block must be large enough to contain all of the digits.

The inverse of STR is a function called VAL:

VAL (data block)

The data block (string) is converted to a number. This corresponds to the BASIC command VAL.

There is a simple method for forming an array of type CHAR:

"Text for the array"

The text enclosed in quotation marks is a data block because it is longer than a single array element. Note that standard Pascal requires that strings be enclosed in single quotes. Pascal 64 uses quotation marks (also called double quotes) so that cursor and color control characters may be inserted into the string when using the BASIC editor. The data block is automatically terminated with a CHR(0). You can initialize an array as follows:

STRING:= "SOME TEXT FOR THE ARRAY";

The text may contain control characters.

The comparison operators <, >, =, <>, <=, >= are defined for strings. The result of a comparison is determined based on alphabetic ordering, as it is in BASIC. A CHR(0) again acts as an end-of-string marker.

The expression STRING1<STRING2 is TRUE if the contents of STRING1 come alphabetically before the contents of STRING2;
otherwise it is FALSE.

The following program sorts strings:

```
10 PROGRAM SORT;
20 VAR ARR:ARRAY[1..20] OF ARRAY[1..10] OF CHAR;
30 VAR STRING:ARRAY[1..10] OF CHAR; I,J:INTEGER;
40 BEGIN
50 FOR I:= 1 TO 20 DO READLN (ARR[I]);
60 FOR J:= 1 TO 19 DO
70 FOR I:= 1 TO 19 DO
80 IF ARR[I]>ARR[I+1] THEN
90 BEGIN STRING:= ARR[I];
100 ; ARR[I]:= ARR[I+1];
110 ; ARR[I+1]:= STRING;
120 END;
130 FOR I:= 1 TO 20 DO WRITELN (ARR[I]);
140 END.
```

6.7 String functions

You may find that some string functions found in BASIC are missing from Pascal 64. But since the result of a Pascal function may be an array, it is easy to write your own string functions:

```
10 VAR A,A1:ARRAY[1..100] OF CHAR;
20 FUNCTION MID (A:ARRAY[1..100] OF CHAR;
```

```
    I,J:INTEGER):ARRAY[1..100] OF CHAR;
30 BEGIN
40 MID:= A[I..I+J-1];
50 MID[J+1]:= CHR(0);
60 END;
70 FUNCTION CONCAT (A,B:ARRAY[1..100] OF CHAR):
    ARRAY[1..100] OF CHAR;
80 BEGIN
90 CONCAT:= A;
    CONCAT[LENGTH(A)+1..0]:= B[1..LENGTH(B)+1];
100 END;
110 BEGIN
120 READLN (A,A1);
130 WRITELN (CONCAT(A,A1));
140 WRITELN (MID(A,10,5));
150 END.
```

Note that the variable A is declared more than once and each time represents a different variable with a different storage place, corresponding to the Pascal local/global definitions. This means that in the main program you do not have to worry about which variables the subroutines use. The function MID requires the same parameters as the corresponding BASIC function. The function CONCAT accepts two strings as input, places one after the other, and returns this single string. Both functions expect strings ending with CHR(0) and return such an array. Additional functions can easily be added in the same manner.

6.8 Packed arrays

Larger computers contain memory with more bits per word than does the Commodore 64. It would be wasteful of memory to assign an entire memory location to a variable which might require just one bit (boolean). For this reason, Pascal has the ability to pack multiple array elements into a single memory location (PACKED ARRAY). On a computer with an 8-bit word length, a packed array is usually not necessary. In addition, accessing a packed array takes longer because of the additional calculations necessary to cross-reference indices and parts of memory locations. A packed array must first be UNPACKed before it can be accessed; the computer does this automatically.

For microcomputers, this procedure has little importance. For that reason, only packed arrays of type BOOLEAN are allowed in Pascal 64. A packed array may be accessed directly.

Format:

VAR name:PACKED ARRAY[0..end value] OF BOOLEAN;

The packed array must have a starting value of 0 and must have an ending value which is 1 less than a multiple of 8. If this is not the case, the

compiler will automatically round up the ending value. A packed array may be an element of another array or a record, but only the last dimension may be packed. A packed array may be accessed normally, each element containing a value of TRUE or FALSE. No subranges may be formed from packed fields. Each element requires only one bit, meaning a field of 10000 elements requires only 1250 bytes. A packed array is ideal for very large arrays which have only two possible values. An example can be found in Chapter 4.

CHAPTER 7

7.1 Files

In order to work with files, Pascal allows the commands WRITE, WRITELN, READ, and READLN to be used with devices other than the screen and keyboard. This is done simiar to using file in BASIC.

Before a file can be opened, it must first be declared as a variable.

VAR name,device address,secondary address:FILE;

The device address and secondary address are the same parameters that are used in the BASIC OPEN command. The OPEN command logical file number is replaced by the **name**. The VAR declaration reserves the **name** of the data structure FILE which is used when opening and closing the file as below.

The file is actually opened by the RESET statement:

RESET (name,"filename,text,etc.");

The RESET statement can also be replaced with REWRITE. It is not necesssary to use RESET and REWRITE to distinguish between a read and write command because this is done through the secondary address (datasette) or through the text appended to the filename (disk). The RESET statement corresponds to the OPEN command in BASIC. The filename is unnecessary for many devices (printer) and may be omitted in these cases. The filename may also be replaced with a data block (string). The other parameters for RESET are already given in the file declaration.

A file is closed with the CLOSE statement:

CLOSE (name);

A file should always be closed before the end of a program so that no data is lost.

A file is accesed through the existing input/output commands (WRITE, WRITELN, READ, READLN). The filename is placed before the parameter list for any of the statements, so the computer knows which file is being accessed. The data is output exactly as it would be on the screen and can be read in again in a corresponding manner. To read data with READLN, the data must have been written in the corresponding format, i.e. it must have been written with WRITELN.

Example:

```
10 VAR CHARACTER:CHAR; DISK,8,8:FILE;
20 PRINTER,4,0:FILE; STATUS:INTEGER;
30 BEGIN
40 RESET (DISK,"TESTFILE,S,R"); RESET(PRINTER);
50 REPEAT
60 READ (DISK,CHARACTER); STATUS:= PEEK(144);
70 WRITE (PRINTER,CHARACTER); WRITE (CHARACTER);
80 UNTIL STATUS=64; (* END OF LOOP *)
90 WRITELN; WRITELN ("DONE."); CLOSE (DISK);
100 CLOSE (PRINTER); END.
```

The program reads a file saved on the disk under the name TESTFILE and then prints this on the screen and printer. Since we used a variable of type CHAR, this is done in single characters at a time.

7.2 Error channel and file status

The function PEEK(144) returns the file status of the last input or output function and corresponds to the BASIC variable ST. If the memory location 144 contains a 64, the end of the file was reached.

A file is automatically predeclared by Pascal 64 for the disk error channel:

VAR ERROR,8,15:FILE;

This declaration need not be written in a program, it is automatically included by the compiler. The error channel is opened with:

RESET (ERROR);

Now file messages can be received or commands sent over this file. The error channel is closed with:

CLOSE (ERROR);

Remember that closing the error channel also closes all other files on the disk. The error channel should only be closed at the end of the program.

7.3 Input/output of data blocks

As indicated in Chapter 6, it is possible to input and output data blocks. Several cases must be distinguished:

- The only data blocks which it makes sense to output to the screen or printer are those of type CHAR. The data block will then be printed as a string. A CHR(0) causes the output to cease.

- It is also possible to input only data blocks of type CHAR from the keyboard. Pressing the RETURN key indicates that the input is done and the remaining elements are filled with CHR(0).

- It is possible to input or output data blocks using the disk and all other devices with device numbers greater than 7 (such as a second disk). The data block is thereby written or read byte by byte. This also applies to data blocks of type CHAR – a CHR(0) is treated as any other character.

Particularly interesting is the input and output of data by records or even by entire files with a single statement. Sequential files are not well suited for this sort of access since all of the preceding records must be read before a given record can be read in. Sequential files do not permit direct access to their elements.

7.4 Relative files

Relative files represent an efficient method of data access. Because the disk drive user's manual does not explain the use of realtive files in detail, we will begin with a discussion of the principles of relative files. Pascal allows relative files to be used and worked with greater ease than BASIC. In addition, the Pascal data type RECORD may be used with relative files. (The RECORD# statement used for relative files in BASIC 4.0 should not be confused with the RECORD data type in PASCAL. Note also the temporary distinction between a relative record and a Pascal RECORD)

The principle of relative files is a sequence of several data records. All records have the same length so that only the number of the record need be used to allow the disk to find it in the shortest possible time. The entire file need not be read in as is the case with sequential files. The individual data record may be written or read in a manner similar to sequential files. Writing and reading may even be done simultaneously – it is not necessary to designate a relative file as read or write. The length of the record may not be exceeded under any circumstances, however. If a specific portion of a data record is to be accessed, the data pointer may be positioned over individual bytes.

The following procedures are used for managing relative files:

Creating a relative file:

- The file is opened and the length of the data record is given (max. 254 bytes).
- The number of the last data record of the file is denoted.

Writing a record:

- The file is opened if it is not already opened.
- The file pointer is positioned to the data record to be written. If this record

has never been used, the disk will respond with a RECORD NOT PRESENT error. This message can be ignored in this case.
- The data record is written.

Reading a record:

- The file is opened if required.
- The file pointer is positioned to the record to be read. If this record has never before been written, the error message RECORD NOT PRESENT will be sent over the error channel, and this may of course NOT be ignored.

After all of the changes or additions have been made to a relative file, it must be closed in order to prevent data from being lost.

The advantages of relative files are clear:

- Direct and therefore faster access of individual data records and even individual bytes.
- Preservation of the main memory as direct access memory.
- The ability to read and write in the same file.

Additional information and uses of relative files and other types of files can be found in the book *The Anatomy of 1541 Disk Drive* by First Publishing Ltd.

7.5 Using relative files

A relative file must be declared as any other data structure as a variable:

```
VAR name,device address,secondary address:FILE;
```

This corresponds to the declaration of sequential files.

To open a relative, a variable of type INTEGER (or CHAR) must be present which contains the length of the data record. In addition, the disk error channel must be open.

```
RESET (name,"filename,L,",variable);
```

The record length is replaced with a dash "–" and then by the contents of the variable. A number may be used directly, instead of the variable. The filename together with the file type "L" and the dash may be combined into an array of type CHAR. In a certain sense, a relative file in Pascal is really a FILE OF RECORD. Because the organization of relative files is handled by the disk drive and not by the computer, the data type FILE in Pascal 64 requires no further information.

A relative file is closed as usual with:

CLOSE (filename);

The positioning to a record is done with the statement SEEK:

SEEK (name,record number,byte number);

If the byte number is omitted, the first byte will be accessed. The record number and byte number may also be expressions. The error channel must be opened before this statement can be used.

The desired data record can now be read or written at the given byte position. To determine the length of a record, the lengths of the individual basic data types must be summed. When using WRITELN instead of WRITE, an extra byte must be added for the trailing RETURN.

The following program demonstrates the use of relative files:

```
10 VAR DFILE,8,2:FILE;
20 DATA:RECORD CITY:ARRAY[1..12] OF CHAR; ZIP:INTEGER; END;
30 POS,CHOICE,LEN:INTEGER;
40 BEGIN
50 LEN:= 15; RESET (DFILE,"FILE.REL,L,-",LEN);
60 RESET (ERROR); REPEAT
70 WRITE ("2=READ, 1=WRITE, 0=END? "); READLN (CHOICE);
75 WRITELN ("POSITION? "); READLN (POS); SEEK (DFILE,POS);
80 CASE CHOICE OF
90 ; 1: BEGIN READLN (DATA.CITY,DATA.ZIP);
100 WRITELN (DFILE,DATA); END;
110 ; 2: BEGIN READLN (DFILE,DATA);
    WRITELN (DATA.STAT,DATA.ZIP); END;
120 END; UNTIL CHOICE=0;
130 CLOSE (DFILE); END.
```

CHAPTER 8

8.1 SETs

The set in Pascal is a collection of several elements of the same basic type. A set is therefore a data structure like arrays and records. Each element of the basic type is either present within a set or absent from it. There is no direct access to the individual elements of the set, as is possible using the index of an array. It is only possible to determine if an element is present in a set or not. If an element is assigned to a set twice, the contents of the set will not change. Several sets may be combined using the mathematical operators. The data type of a set may be either INTEGER, CHAR, or, since elements of a user-defined TYPE are internally represented as integers, of a user-defined TYPE. Since the actual type is only important for output, and sets cannot be output, it is not necessary to specify the type of set in Pascal 64. A set may therefore contain any desired elements of type INTEGER or type CHAR. REAL values will automatically be converted to integers. A set should only be used for one data type so you may wish to give the data type of set as a comment behind it to remind you. The set declaration is the same as for a basic data type, although the set itself does not represent a basic data type:

```
VAR name:SET;
```

A set may be placed in a declaration wherever a basic data type may be placed. Sets may be used as elements of arrays and records. In standard Pascal, the range of values which may be assigned to a set are limited by most compilers and must be given in the declaration. The size of the range varies from compiler to compiler but is usually between 64 and 256. Sets in Pascal 64 can contain any of the elements of type INTEGER or CHAR. A set could contain both the values −20000 and 20000, for example. In spite of this, the maximum number of elements in a set is limited, although not by their value range. If a set is assigned more than 63 elements, an OVERFLOW message is given. To determine if an element of a desired data type is in a set, you need only determine if the array index or pointer address is contained in it. Using this trick, it is possible to make sets of data types other than INTEGER or CHAR. It must be noted that a set does not represent a data block. In spite of this, sets can be assigned to each other or passed to subroutines. If sets are to be saved to disk, they must be saved as part of a record so that the the record can be output.

8.2 Representing sets

A number can be given as a variable or as a value directly in a program. The same applies to sets. A set is represented directly by listing the set elements and placing these in square brackets:

```
[5,4,3,-1,X,Q+1,5 DIV 4,7,8,9,10]
```

or with a set of type CHAR:

```
["A","B","C","R",CHR(X),CHR(X+1),CHR(X+2)]
```

The elements of a set may be the results of expressions. A set without elements is represented as an empty set:

```
[]
```

It is obvious that set elements will frequently have numerically successive values. A range of numbers or characters may be specified by the usual double period. The order of elements in a set is entirely irrelevant. The above examples can also be written:

```
[-1,3..5,7..10,X,Q+1,5 DIV 4]
["A".."C","R",CHR(X)..CHR(X+2)]
```

A set specified in this manner may be manipulated directly or assigned to a set variable. The following assignments are possible:

```
SET1:= [1,5,7,9..20]
SET2:= []
PUNCTUATION:= [",","?",".","!",";",":"]
```

8.3 Set operations

The most important set operator is the operator IN. It determines whether or not an element is present within a set.

Format:

```
expression IN set
```

This operator returns a result of type BOOLEAN (true or false) and assumes a place in the algebraic hierarchy of operators. It is placed at the same level as the comparison operators. All other set operators have the same hierarchy as their REAL-operator counterparts. With the help of the operator IN, we have a simple way of replacing a CASE statement:

```
IF A IN ["0".."9"] THEN WRITELN ("DIGIT");
IF A IN PUNCTUATION THEN WRITELN ("PUNCTUATION");
```

The union of two sets contains all of the elements of both and is itself of type set:

[1,3]+[2,4,5] is identical to [1.5]

In order to add an element to a set, it must first be turned into one itself:

SET1:= SET1+[NEWNUMBER];

The difference of two sets is a set containing all of the elements of the first set which do not appear in the second:

[1,2,5,6]–[4,5,8] yields [1,2,6]

The intersection of two sets consists of all elements common to both:

[1,2,5,6]*[4,5,8] yields [5]

Set comparisons:

– equality of sets

SET1=SET2

is TRUE if both sets contain the same elements.

– inequality of sets

SET1<>SET2

is TRUE if at least one element of one set is not contained in the other.

– inclusion (is contained in)

SET1<=SET2

is TRUE if every element of SET1 is also in SET2 (SET1 is a subset of SET2).

– inclusion (contains)

SET1>=SET2

is TRUE if each element of SET2 is also contained in SET1 (SET1 is a superset of SET2).

8.4 Using sets

Sets are useful for avoiding complicated comparisons usings IF statements. Sets can also be used in their mathematical sense, of course. Through the ability to store array indices or pointers, sets can be used to identify elements of more complex data structures.

The following program searches entered text for specific groups (sets) of characters. On of these groups is the special characters which can be defined prior to entering the text. A RETURN ends the special character definition and the text input.

```
20 VAR SPECIAL:SET;
30 SPECIALC,PUNCTC,LETTERC,DIGITC:INTEGER;
40 CHARACTER:CHAR;
50 BEGIN
60 WRITELN ("PLEASE ENTER SPECIAL CHARACTERS");
65 SPECIAL:= []; SPECIALC:= 0;
70 REPEAT
80 READ (CHARACTER);
85 IF CHARACTER<>CHR(13) THEN SPECIAL:= SPECIAL+
    [CHARACTER];
90 UNTIL CHARACTER=CHR(13); WRITELN;
100 PUNCTC:= 0; LETTERC:= 0; DIGITC:= 0;
110 REPEAT READ (CHARACTER);
120 IF CHARACTER IN SPECIAL THEN SPECIALC:= SPECIALC+1;
130 IF CHARACTER IN ["0".."9"] THEN DIGITC:= DIGITC+1;
140 IF CHARACTER IN ["A".."Z"] THEN LETTERC:= LETTERC+1;
150 IF CHARACTER IN [",",".",";",":","?","!"]
   THEN PUNCTC:= PUNCTC+1;
160 UNTIL CHARACTER=CHR(13); WRITELN;
170 WRITELN (SPECIALC," SPECIAL CHARACTERS");
180 WRITELN (LETTERC," LETTERS");
190 WRITELN (DIGITC," DIGITS");
200 WRITELN (PUNCTC," PUNCTUATION MARKS");
210 END.
```

CHAPTER 9

9.1 Dynamic data structures (linked lists)

All of the data structures we've talked about up to this point, except for files, are static, meaning that their size is determined before program execution and cannot be changed while the program in running. Frequently, however, you do not know the exact size of a data structure beforehand, or want to make the most efficient possible use of memory by using a dynamic structure, one that allocates only as much space as you need at any given moment. To this end, Pascal provides pointers which we can use to make what are called linked lists.

In a linked list, an element may be inserted or deleted without affecting the position in memory of the other elements. With an array, for instance, many of the elements must be shifted in memory in order to insert a new one. Although this is easy to program, as shown in Chapter 6, it requires considerably more time to execute than does a comparable operation performed on a linked list. Lists can represent any data structure, not just arrays where each element has just one predecessor or successor. In Pascal, it is possible to determine the memory address of variables and their relationships to each other from within the program. This is done using pointers.

The principle behind pointers is really quite simple. A variable or data structure is assigned a pointer variable by declaration. In order to be able to access the data structure or its elements, the address of the data structure is first transferred to the pointer variable and then the data structure may be indirectly accessed through the pointer variable. To be able to build a list, the data structure contains one or more element which give the address of the predecessor or successor. In order to access the successor of a data structure, the address of the successor is simply assigned to the pointer variable and the successor can then be accessed though the pointer. Because this structure also has a predecessor, it is possible to access the entire linked list using a single pointer. At least one pointer should be defined which points to the start or end of the list so that the list can always be searched from the beginning or end. Because the list elements are only tied together by pointers, their actual order in memory is unimportant. A new element can be inserted into the list by assigning a new memory location to the element and changing two pointers. List elements may also possess more than one predecessor or successor.

Declaration of dynamic data structures:

VAR list name: ↑ data structure;

The data structure should be of type RECORD so that one element of the structure can be be a pointer and point to the predecessor or successor. An

INTEGER variable may take the place of a pointer within the data structure in Pascal 64 because integer variables and pointers may be freely assigned to each other. In standard Pascal, this variable would have to be declared again as a pointer. This difference occurs only in the declaration and has no effect on the rest of the program. The data type INTEGER can also be declared as a POINTER in order to show that a variable functions as a pointer.

The following example demonstrates a pointer declaration:

```
VAR P: ↑ RECORD SUCCESSOR:POINTER; NUMBER:REAL; END;
```

It is assumed for now that the list is already constucted and the variable P contains the address of a list element. The access of a list element occurs as follows:

P	is the variable which contains the address of the list element.
P ↑	is the list element (in the case, a data block of type RECORD with length 7 bytes)
P ↑ .SUCCESSOR	contains the address of the next list element
P ↑ .NUMBER	is the actual contents of the list element and can is treated as any other REAL variable.

To access the next list element, the following assignment suffices to advance the pointer:

```
P:= P ↑ .SUCCESSOR;
```

Now all of the described accesses apply to the next list element.

Most lists have a beginning and an end; seldom are lists chained circularly. It may occur that an element has no successor or that a pointer points to a non-existing list element, if the list has no elements. A pointer which does not point to anything must be assigned the value NIL. NIL means Not In List and is a constant. A pointer which contains NIL points to an element with undefined contents. If such an element is inadvertently accessed, there will be no problem provided the program notices this in due time. Such an element may never be assigned a value in any event.

Naturally, the program is not required to manage the memory itself in order to generate list elements. This work is performed by a Pascal routine:

```
NEW (pointer name);
```

The address of a free memory area will be given to the named variable and the memory reserved for the corresponding list element. Their is no opportunity to free the reserved memory space again. Should this element be deleted because its address was not saved anywhere, the memory space is

lost. It is therefore recommended that for large lists, a second list be made of the list elements no longer required so that these may be recovered if necessary. To free the space reserved for the entire dynamic structure, one uses:

```
LISTINIT;
```

This command erases all lists and is automatically executed at the start of the program. If a list should no longer fit in memory, an OUT OF MEMORY error will be given.

9.2 Using pointers

At least two pointers are required to construct a simple list. One is needed to point to the list element currently being worked with, and one to always point to the start of the list so that the list can be searched from start to finish. A additional aid pointer will also be declared.

```
VAR AID,P,START: ↑ RECORD
                   NEXT:POINTER;
                   WORD:ARRAY[1..10] OF CHAR;
                   END;
```

Since no list elements exist at the start of the program, START may not yet point to a variable:

```
START:= NIL;
```

The second pointer is used when inserting an element:

```
NEW (P);
```

The new element must naturally contain a value:

```
READLN (P ↑ .WORD);
```

The new list element should be used as the new start so that no pointer to the end of the list is required.

Because the new element is supposed to be placed in front of the start, it has the previous start as its successor:

```
P ↑ .NEXT:= START;
```

In addition, the new element is now the start of the list:

```
START:= P;
```

Other elements may be added in the same manner, starting with NEW.

In order to find a certain list element, the list is searched from the beginning:

P:= START;

If this list element is not the one desired, the next element is taken:

P:= P ↑ .NEXT;

This continues until either the desired list element is found, or an element with the address NIL is encountered.

To insert an element into our sample list, the pointer P must point to the new element and the pointer variable named AID to the element before the one to be inserted: The successor of the new element P ↑ is the previous successor of the element AID ↑ :

P ↑ .NEXT:= AID ↑ .NEXT;

The new successor of the element AID ↑ is P ↑ .

AID ↑ .NEXT:= P;

To remove a list element from the list, the previous successor of the element becomes the successor of it predecessor. In the following example, AID^ is the predecessor of the element to be deleted:

P:= AID ↑ .NEXT;
AID ↑ .NEXT:= P ↑ .NEXT;

This sample program illustrates inserting and searching:

```
10 PROGRAM EMPLOYEELIST;
21 VAR P,START:↑RECORD
22 ;               NEXT:POINTER;
23 ;               NAME:ARRAY[1..20] OF CHAR;
24 ;               NUMBER:INTEGER;
25 ;             END;
30 VAR CHOICE:CHAR; SEARCHNAME:ARRAY[1..20] OF CHAR;
40 BEGIN
50 START:= NIL; REPEAT
60 WRITE ("I=INSERT, S=SEARCH, X=END"); READLN (CHOICE);
70 CASE CHOICE OF
80 "I":  BEGIN
90 ;     NEW (P); WRITELN ("NAME?"); READLN (P↑.NAME);
100 ;             WRITELN ("NUMBER?"); READLN (P↑.NUMBER);
110 ;    P↑.NEXT:= START; START:= P;
120 ;    END;
130 "S":  BEGIN
140 ;     WRITELN ("NAME TO SEARCH FOR?");
          READLN (SEARCHNAME); P:= START;
150 ;     WHILE P<>NIL DO
160 ;                  BEGIN
```

```
170 ;                    IF P↑.NAME=SEARCHNAME THEN WRITELN
                            (P↑.NUMBER);
180 ;                    P:= P↑.NEXT;
190 ;                    END;
200 ;     END;
210 END;
220 UNTIL CHOICE="X";
230 END.
```

9.3 Additional list structures

With the help of pointers it is possible to generate other kinds of list structures. The best known structure is the stack which allows access only to the top value. The stack is designated as a LIFO (Last In, First Out) structure. With a FIFO (First In, First Out) structure, elements are placed in at one end of the list and taken out of the other. The FIFO structure is known as a buffer. Lists may also be doubly linked – each element contains pointers to its successor and predecessor. This has the advantage that the list can be searched in either direction. A closed list is a list in which the successor of the last element is the first element. The list has no beginning and no end. Such a list can also be double-linked, of course. More complicated structures have two or more successors for each element. Such a structure is called a tree because the structure continually divides and branches out. The ends of the branches must naturally be denoted with NIL. Starting from the root, it is quite simple to find an element if a consistent decision criterium is used when forming the list. Trees with two successors per element are called binary trees. An element of the tree might point back to the root or list elements may have several predecessors.

9.4 Combinations of data structures

By combining various data structures, it is possible to get around some of the limitations of simpler data structures. The most important combination is the array/record combination. As mentioned before, the capabilities of sets can be expanded by saving array indices or pointer addresses in sets. Even an ARRAY OF SET can be used to overcome the limit of 63 elements per set. Also very helpful is the ability to assign addresses (such as the address of the screen memory, 1024) directly to pointers, without using NEW. This allows entire arrays to be assigned to memory address, a kind of super POKE command. In the following example, certain data structures are used in an unconventional manner, but is a good proof of the versatility of data structures and the possibilities they offer:

```
10 PROGRAM SCROLL;
20 VAR SCREEN:↑ARRA[1..1000] OF CHAR;
30 I,J,X:INTEGER; C:REAL;
40 LINE:ARRAY[1..40] OF CHAR;
60 BEGIN SCREEN:= 1024;
```

```
70 WRITELN ("TEXT?"); READLN (LINE);
80 FOR I:= 1 TO 25 DO WRITELN; WRITELN (LINE);
85 FOR I:= 0 TO 999 DO POKE I+55296,1;
90 FOR I:= 1 TO 1000 DO BEGIN
100 FOR J:= 7 DOWNTO 0 DO BEGIN
110 POKE 53270,J; FOR X:= 1 TO 30 DO BEGIN END;
120 END; REPEAT UNTIL PEEK(53266)<10;
130 SCREEN↑[1..0]:= SCREEN↑[2..1000]; END; END.
```

CHAPTER 10

10.1 Graphics statements

10.1.1 Initializing the graphics screen

Switching the screen to high-resolution graphics is done with the GRAPHIC statement.

Format:

GRAPHIC mode;

If the mode is equal to 1, the graphic screen is switched on but not erased. The resolution is 320 x 200 points in two colors. The background color remains the same while the point color is the same as the cursor color. Multi-color graphics are discussed in Chapter 12.

If the mode is equal to 0, the normal screen is switched back on and cleared.

If the GRAPHIC statement is used anywhere in a Pascal program, the memory from 8192 to 16384 is reserved for graphics, and the area from 5130 to 8192 is free for machine language programs or other purposes. These two areas can also be used for other purposes. One possible use is for storing multiple screen images which can be exchanged quickly with the help of pointers.

Sprites cannot be placed in the range 4096 to 8191; the video controller accesses the character generator in this area although the main processor has RAM there. More about the inner workings of the video controller chip can be found in our book *The Anatomy of the Commodore 64* or your Commodore User's Guide.

10.1.2 Clearing the graphics screen

Format:

SCREENCLEAR;

This statement erases the graphics screen. It should always be used when first switching over to the graphics screen.

10.1.3 Setting a point

Format:

PLOT expression1,expression2;

PLOT sets a point having the same color as the characters on the normal screen. Expression1 gives the X-coordinate (0–319), expression2 gives the Y-coordinate (0–199).

10.1.4 Erasing a point

UNPLOT expression1,expression2

Erases a point and otherwise works the same as the PLOT statement.

Example:

```
10 CONST PI=3.14159265; STEP=0.196349541;
20 VAR X,I:INTEGER;
30 BEGIN GRAPHIC 1; SCREENCLEAR;
40 I:= PI*-1;
50 FOR X:= 0 TO 319 DO BEGIN
60 PLOT X,100+90*SIN(I);
70 I:= I+STEP;;
80 END;
90 GRAPHIC 0; END.
```

You can add additonal graphics statements of your own with the help of procedures. You needn't worry that user-defined functions will be slower than commands in BASIC expansions. The main reason that graphics subroutines in run so slowly is because of the complicated calculation required for the pixel addresses. This is done for you in Pascal with the PLOT statement. The advantage of user-defined functions is their adaptability to specific problems. One important graphics statement is drawing a line. For this reason, such a routine can be found in Chapter 12. This subroutine may also provide you with useful information for developing a three-dimensional DRAW statement.

10.1.5 The SPRITE statement

Format:

SPRITE number,"bit pattern";

The bit pattern is saved at the address given as the number. The number of bits must be a multiple of 8. When defining a sprite, 24 bits are specified per line; the SPRITE statement must be used 21 times; and the address must be incremented by three for each successive command. A space represents an erased bit, any other character sets a bit. The SPRITE statement be used for other purposes, such as creating a new character set.

Example:

```
SPRITE 960,"       XXXXXXXXXX       ";
SPRITE 963,"      XXXXXXXXXXXX      ";
```

defines the first two lines of sprite number 15.

Sprites 11 (address 704 = 64*11), 13 (832), 14 (896), 15 (960), 32 (2048), 33 (2112), 34 (2176), and 35 (2240) may be used. When sets are also used, only sprites 11, 32, and 33 are available.

When graphics are activated, sprites 253 (16192), 254 (16256), and 255 (16320) are also available. If graphics are turned on but then turned off again, all of the sprites which can be stored in the graphics RAM may also be used. If necessary, you can try using sprites which are stored in memory which is not being used at the moment. Sprite number 8 is only disturbed by the READ statement, for instance.

10.2 The POKE statement

Format:

```
POKE expression1,expression2;
```

This command writes the result of expression2 into the memory location specfified by the result of expression1. This corresponds to the BASIC command of the same name. The POKE statement can be replaced by a pointer, which opens up even greater capabilities than changing single memory locations. This was demonstrated in Chapter 9.

10.3 Executing machine language programs

Format:

```
SYS expression;
```

A machine language program starting at expression is executed. This command is also identical to the BASIC command. The memory area from $C000 to $CFFF is available for machina language programs, provided no recursive calls are used, or these do not require much variable storage (recursive variables are stored starting at $A000).

Example:

```
SYS 64738;
```

resets the computer.

10.4 The high-speed mode

Pascal 64 gives you the choice of executing arithmetic operations in either integer or floating-point format. So long as it does not interfere with the

result, the compiler chooses the integer mode. With the operators *, /, +, and −, this cannot always be determined and the compiler cannot decide. For example, of one multiplies two integers, the result may be too large for an integer to contain. The INTEGER; statement in Pascal 64 is used for this reason.

Format:

INTEGER;

This statement has the effect that the operations *, −, and + are executed in integer format if and only if both operand are integers. An operand counts as type INTEGER if it is an integer variable, if the result of a arithemtic operation is an integer, or if a number or constant has no decimal point and lies in the integer range (−32768 to 32767). The result of an integer operation is itself of type INTEGER and is automatically converted if a REAL result is required. If an INTEGER result is required, this has the advantage that no conversion from REAL to INTEGER need be performed and time is saved. This is the case for POKE or array indices, for example.

The operators DIV and MOD are used for integer division. The / operation is always executed as a REAL operation. This is generally necessary for a division.

In certain circumstances an error may result from range overflow that would not occur when using floating-point arithemetic. The INTEGER; statement must therefore be used with care, particularly with multiplication which can quickly exceed the legal range from −32768 to 32767. Because the INTEGER statement only has effect when both operands are of type INTEGER, it may often be used without error.

Turning off the optimization mode:

Format:

REAL;

This command switches back to the standard mode. This is important so that calculations can be made quickly inside of a loop and large numbers may be manipulated without danger outside of it, for example. In order to make optimum use of the INTEGER mode, it should be turned off only when necessary.

Both of these commands work only on the program parts following them (not on subroutines) and are independent of the path of execution.

Example:

```
10 BEGIN
20 IF 1=0 THEN INTEGER;
30 WRITELN (2*3);
40 END.
```

The multiplication is executed in integer format although the program portion containing INTEGER is never executed. (INTEGER and REAL are compiler directives, affecting the code generated, not the way this code is executed.)

To execute the entire program in integer format, place the INTEGER; statement after the BEGIN of the main program.

10.5 Simultaneous execution of two programs

The Commodore 64 possesses many capatilities for interrupt control. These are provided by the video processor and the I/O kernal routines. With Pascal 64, it is possible for the first time to develop interrupt routines in a high-level language. Until now, this was only possible in machine language. When the computer receives an interrupt, a special program, is called which performs such operations as reading the keyboard and flashing the cursor. This normally occurs sixty times per second, independent of normal BASIC. At this place we introduce the INTERRUPT statement:

INTERRUPT procedure name;

This statement directs the interrupt to a Pascal subroutine. This subroutine may use only its own variables and may not expect any parameters. This subroutine is called before before the normal interrupt routine, normally sixty times per second. The normal Pascal program is undisturbed. Naturally, these programs are actually executed consecutively, but because they alternate 60 times per second, only the execution will seem somewhat slower.

A procedure intended to handle an interrupt must satisfy certain conditions:

- The procedure must execute fairly quickly, within 1/60th of a second so that execution can return to the main program before the next interrupt occurs. If necessary, it is possible to write a routine which can determine that its work is not done and continue with this. The EXIT; command can be used in these cases to leave the subroutine prematurely. The INTEGER statement can also be used to alleviate this problem.

- The procedure may not execute any floating-point or other time-consuming operations such as READLN. Data types such as CHAR, BOOLEAN, or INTEGER do not affect the main program, nor do calculations made using these types. One need only ensure that integer calculations always return an integer. Use of the INTEGER statement is therefore necessary. Division must always be executed with DIV.

- No restrictions apply to the normal Pascal program running along with the interrupt routine.

These conditions appear somewhat strict, but quite a lot can be accomplished with integer arithmetic. Array indices are of type INTEGER for

example, and it is possible to use arrays and data blocks in an interrupt routine. One limitation applies to the POKE statement. Addresses over 32767 are not allowed since these cannot be represented by an integer and are therefore not allowed:

POKE 50000,1; is not allowed in an interrupt routine.

This is not as bad as it might seem since every integer consists of 16 bits, as does a memory address. One need only use the corresponding negative (two's complement) value of number greater than 32767. To determine the address to use in the POKE statement from the actual memory address, simply subtract 65536 or 2 ↑ 16.

POKE −15536,1; has the desired effect.

If this seems too complicated, another method may be used which involves overflow in the INTEGER mode:

POKE 20000+30000,1; also leads to the desired result because the addition is of type INTEGER and also yields an integer value, though only when using the INTEGER; statement. This little trick works only in the high-speed mode and should normally not be used.

The interrupt procedure is turned off with:

INTERRUPT OFF;

If this statement is not used, the interrupt procedure will continue to execute even after the main program is done.

The interrupt command makes full use of the capabilities of your Commodore 64 and can be used for interrupts other than the system interrupt. More about the interrupt registers of the I/O kernal can be found in the book The Anatomy of the Commodore 64. As an example of the use of the interrupt command we present a sprite control routine which is independent of the main program. For demonstration purposes, the interrupt is never turned off in this program. The effects of this program can be removed only be turning the computer off.

```
10 VAR Y,I:INTEGER;
11 PROCEDURE MOVE;
12 BEGIN INTEGER;
13 POKE -12288,Y; POKE -12287,Y; Y:= Y+1; REAL; END;
20 BEGIN
30 SPRITE 832,"          XXXXX          ";
31 SPRITE 835,"        XXXXXXXXX        ";
32 SPRITE 838,"       XXXXXXXXXXX       ";
33 SPRITE 841,"      XXXXXXXXXXXXX      ";
34 SPRITE 844,"      XXXXXXXXXXXXX      ";
35 SPRITE 847,"      XXXXXXXXXXXXX      ";
36 SPRITE 850,"      XXXXXXXXXXXXX      ";
37 SPRITE 853,"       XXXXXXXXXXX       ";
```

```
38 SPRITE 856,"      XXXXXXXXXXX       ";
39 SPRITE 859,"      X XXXXXXX X       ";
40 SPRITE 862,"       X XXXXX X        ";
41 SPRITE 865,"       X  XXX  X        ";
42 SPRITE 868,"       X   X   X        ";
43 SPRITE 871,"        X  X  X         ";
44 SPRITE 874,"        X  X  X         ";
45 SPRITE 877,"         XXXXX          ";
46 SPRITE 880,"         XXXXX          ";
47 SPRITE 883,"         XXXXX          ";
48 SPRITE 886,"          XXX           ";
49 SPRITE 889," ";
50 SPRITE 892," ";
60 POKE 2040,13;
70 POKE 53269,1;
80 Y:= 0; INTERRUPT MOVE;
90 FOR I:= 1 TO 1000 DO WRITE (I);
120 END.
```

10.6 Run-time error messages

BAD SUBSCRIPT
An attempt was made to access an array variable with an index outside the declared range. The size of this block is not checked when assigning data blocks because this often need not be known (array sections).

DIVISION BY ZERO
A floating-point variable was divided by zero.

ILLEGAL QUANTITY
A value lies outside of the range allowed for the command, such as an unpermitted coordinate for PLOT or UNPLOT.

OUT OF MEMORY
The space allocates for the machine stack or recursion stack was exceeded. This may happen as a result of deep nesting of user-defined functions or through a very deep recursive nesting. The message will also result if the command NEW cannot find enough space for a list element.

OVERFLOW
A floating point calculation exceeded the legal range (1.70141183E+38) or a set contains more than 63 elements.

Other error messages correspond to those of the BASIC interpreter. Syntax and other such errors are given by the compiler.

CHAPTER 11

11.1 Symbol files for ASSEMBLER/MONITOR-64

Pascal 64 variables, particularly those of type INTEGER or CHAR, may be used by a machine language program. In order that the ASSEMBLER/ MONITOR 64 available from First Publishing Ltd can recognize these variables, it must have access to a symbol table containing the addresses and names of the variables. The following statement is used to create a disk file of the addresses of the Pascal variables:

SST "name,S,W";

Pascal 64 writes all variables known at the time the statement is executed to the variable file. Although the statement is executed by the compiler, local variables are NOT saved if the command is executed in the main program. It should be used immediately after the SYS statement so that all of the important variables are accounted for.

The symbol file produced by the compiler is not yet readable by the assembler because the variables are still in the form in which Pascal 64 saved them. The program SYMBOL is used to convert the Pascal 64 variables into a symbol file that is readable by ASSEMBLER/MONITOR 64. After starting the program SYMBOL, insert the diskette containing the variable file produced by Pascal 64 and enter the name of the file. Answer the question about printer output with Y or N and enter the name of the file to be created. The program displays the names of the variables and their addresses. All names will be given without special characters ("." or "^") and only the starting address of an array are given. If you want to place a name in the new file, simply press RETURN. Pressing "A" will transfer all variables to the new file and pressing "S" will cause the program to skip all following variables. Any other keys will cause the program to move on to the next name.

A symbol file created in this manner may be read by the ASSEMBLER/ MONITOR 64 and the variables are treated as normal symbols. More information can be found in the ASSEMBLER/MONITOR 64 manual.

11.2 Jump to monitor

Format:

STOP;

This program will cause the machine language command BRK to be inserted into the Pascal program. If a monitor is present and active when this command is executed, control passes to it. Memory addresses can be examined and the execution of the Pascal program can be supervised from

within the monitor. If the interruption is no longer needed, it can be changed to a NOP ($EA). The Pascal program can be continued by entering the monitor command ".G".

CHAPTER 12

To help you create your own subroutine library, we have included the following graphics routines:

12.1 Drawing a line

The procedure DRAW must be given four parameters. The first two form the coordinate of the starting point of the line and last two give the end point of the line.

```
10 PROCEDURE DRAW (X1,Y1:REAL; X2,Y2:INTEGER);
20 VAR COUNTER,DX,DY:INTEGER; STEP:REAL;
30 BEGIN X1:= TRUNC(X1); Y1:= TRUNC(Y1);
35 DX:= X2-X1; DY:= Y2-Y1;
40 IF (X1=X2) AND (Y1=Y2) THEN PLOT X1,Y1; ELSE BEGIN
50 IF ABS(DX)>ABS(DY) THEN BEGIN
60 STEP:= DY/ABS(DX);
70 COUNTER:= DX/ABS(DX); WHILE X1<>X2 DO BEGIN
80 PLOT X1,Y1; Y1:= Y1+STEP; X1:= X1+COUNTER; END; END;
90 ELSE BEGIN
100 STEP:= DX/ABS(DY);
110 COUNTER:= DY/ABS(DY); WHILE Y1<>Y2 DO BEGIN
120 PLOT X1,Y1; X1:= X1+STEP; Y1:= Y1+COUNTER; END; END;
130 END; END;
```

When inserting this subroutine into your own programs, the line numbers must, naturally, be changed. It is better to save the routine separately and insert it via EXTERNAL and the linker.

12.2 Activating multi-color graphics

The procedure MULTICOLOR requires four parameters (0–15) which determine the four colors in the multi-color mode.

```
10 PROCEDURE MULTICOLOR (C0,C1,C2,C3:INTEGER);
20 VAR COUNTER,C12:INTEGER;
30 BEGIN INTEGER;
40 POKE 53270,PEEK(53270) OR 16;
50 POKE 53281,C0;
60 C12:= C1*16+C2;
70 FOR COUNTER:= 1024 TO 2023 DO POKE COUNTER,C12;
80 FOR COUNTER:= 55296 TO 56295 DO POKE COUNTER,C3;
90 REAL; END;
```

12.3 Setting a point in multi-color mode

The multi-plot procedure requires three parameters. These are the X-coordinate (0–159), Y-coordinate (0–199), and color (0–3).

```
10 PROCEDURE MULTIPLOT (X,YM,C:INTEGER);
20 BEGIN INTEGER;
30 IF C AND 1=1 THEN PLOT XM*2+1,YM;
40 ELSE UNPLOT XM*2+1,YM;
50 IF C AND 2=2 THEN PLOT XM*2,YM;
60 ELSE UNPLOT XM*2,YM;
70 REAL; END;
```

CHAPTER 13

13.1 Introduction to structured programming

Structured programming is founded on the concept of dividing a program into parts which have only one starting point and one ending point (BEGIN...END) which are themselves composed of blocks. Blocks may only be placed after each other, next to each other (using conditional branches), or in loops. They may not overlap and no explicit jumps (GOTO) are allowed. By representing a block as a box, a so-called structogram can be created, a good graphic alternative to a flow chart. The block's structure can then be explained within the box (loops or branches) and its blocks represented inside it. The conditions of branches can be written above each block.

The problem to be solved must be broken into logical parts, and these parts broken down still farther until the individual parts can be easily coded into Pascal statements. This method of programming is called top-down programming because the solution starts at the top (the entire problem) and involves progressively breaking the problem down into simpler and simpler parts. This allows even large, complex problems to be solved.

A small practical example of this program technique might involve the following problem:

Print all of the prime numbers less than 1000.

This problem can be divided into two basic steps:

- Checking to see if a number is prime.
- The creation of numbers from 1 to 1000 and passing control to the check block.

The number 1 need not be checked. The current number will be stored in the variable N.

The rough structure of the program looks like this:

```
10 VAR N:INTEGER; BEGIN
20 FOR N:= 2 TO 1000 DO BEGIN
30 check to see if prime
60 print the number if prime
70 END; END.
```

In order to determine if a number is prime, we will divide it into its roots.

First these numbers must be generated. The variable COUNTER will be used for this purpose:

```
10 VAR N,COUNTER:INTEGER; BEGIN
20 FOR N:= 2 TO 1000 DO BEGIN COUNTER:= 1;
30 REPEAT
40 COUNTER:= COUNTER+1;
50 UNTIL COUNTER>SQRT(N) OR evenly divisible;
60 IF NOT(evenly divisible) WRITELN (N);
70 END; END;
```

All that remains is the test for divisibility:

```
10 VAR N,COUNTER:INTEGER; BEGIN
20 FOR N:= 2 TO 1000 DO BEGIN COUNTER:= 1;
30 REPEAT
40 COUNTER:= COUNTER+1;
50 UNTIL (COUNTER>SQRT(N)) OR (N MOD COUNTER=0);
60 IF NOT(N MOD COUNTER=0) WRITELN (N);
70 END; END;
```

The number is prime if all divisions have a remainder. The remainder calculation is done with MOD.

The program can be optimized in a number of ways. Even numbers need not be checked, for example. This and larger programs can be made more readable through the careful choice of variable names and liberal use of descriptive comments.

A refined program for calculating prime numbers can be found in Chapter 5. It uses a different method and requires considerably more memory.

CHAPTER 14

14.1 An overview of Pascal 64

14.1.1 Data types

REAL	– Range +/–1.70141183E+38, 9 place accuracy
INTEGER	– Range +/–32767, no decimal places
CHAR	– All representable characters, length 1 character
BOOLEAN	– TRUE or FALSE
SET	– 63 INTEGER or CHAR elements per set
RECORD	– Combination of desired data types
ARRAY	– Array of desired data types
PACKED ARRAY [0..end] OF BOOLEAN	
	– packed array of boolean values
↑ data type	– indirect access to a data structure for construction of lists
FILE	– sequential or relative file for storage of data

Data types may be nested as desired.

14.1.2 Declarations

Declared names may be any length and may consist of letters and digits. The first character must be a letter.

PROGRAM name;
– program header

VAR variable list:type;
– variable declaration with desired type

CONST name=value;
– constant declaration

TYPE name=(name list);
– type declaration

14.1.3 Procedure and Functions

PROCEDURE name (parameter list); variable declaration subroutine declarations;
Definition of a procedure which is called like a command and has its own local variables and subroutines.

FUNCTION name (parameter list):type; variable declaration subroutine declarations;
Defines a function which is called like a built-in function. The function type may also be an array or a string.

14.1.4 Structures

REPEAT block; UNTIL condition;
Repeats a block until the condition is fulfilled.

WHILE condition DO block;
Repeats a block as long as the condition is fulfilled.

FOR variable:= start value TO end value DO block;
Increments the variable from start value to end value and executes the block each time.

FOR variable:= start value DOWNTO end value DO block;
Decrements the variable from start value to end value and executes the block each time.

IF condition THEN block;
If the condition is fulfilled, the block is executed.

IF condition THEN block1; ELSE block2;
If the condition is fulfilled, block1 is executed, otherwise block2 is executed.

```
CASE expression OF
value list: block;
value list: block;
    ....       ...
END;
```

If a value is identical to the result of the expression, the block following it will be executed.

EXIT;
Exits a procedure.

```
GOTO label;
    ....
label:  ...
```

Unconditional jump.

WITH record name DO block;
The record name is prefixed to all variables for which this makes sense.

14.1.5 Input/output statements

READ (variable list);
The variables are read in from the keyboard.

READLN (variable list);
The variables are read from the keyboard and a RETURN is expected.

WRITE (expression,"text",data block,string,...);
The results of the expressions, the data blocks, and the texts are displayed on the screen.

WRITELN (expression,"text",...);
As WRITE, plus a RETURN is added to the end of the output.

GET variable;
Gets a characters from the keyboard and places it in the variable.

VAR name,device,secondary address,text:FILE;
A file is defined.

RESET (name,"filename"); or REWRITE (name,"filename");
A file is opened.

RESET (name,"filename,L,−",variable);
A relative file is opened.

CLOSE (name);
Closes a file.

WRITE (name,list); or WRITELN (name,list);
Outputs to a file.

READ (name,list); or READLN (name,list);
Inputs from file.

SEEK (name,record number,byte);or SEEK (name,record number);
Selects a record in a relative file.

14.1.6 Other statements

GRAPHIC 1;
Turns on high-resolution graphics.

GRAPHIC 0;
Switched to normal screen.

SCREENCLEAR;
Clears the graphics screen.

PLOT expression1,expression2;
Sets a graphics point.

UNPLOT expression1,expression2;
Erases a graphics point.

SPRITE address,"bit pattern";
Writes a bit pattern in the memory area.

POKE expression1,expression2;
Expression2 is writtento the memory location specified by expression1.

SYS expression;
Executes a machine language program.

INTEGER;
Turns on the high speed mode.

REAL;
Turns off the high speed mode.

INTERRUPT procedure name;
Interrupt-controlled simultaneous execution of two programs.

INTERRUPT OFF;
Turns the the expanded interrupt off.

FILLCHAR (string,character);
Fills a string with a character.

STR (number,string);
Converts a number into a string.

NEW (name);
Generates a new list element.

LISTINIT;
Erases all lists.

SST "name,S,W";
Creates a symbol table for ASSEMBLER/MONITOR-64 package with the help of the program SYMBOL.

14.1.7 Assignments

variable:= expression;
The result of the expression is assigned to the variable.

data block:= data block;
A data block is assigned to another.

14.1.8 Arithmetic operations

+	addition (applies also to sets)
–	subtraction (applies also to sets)
*	multiplication (applies also to sets)
/	division
**	exponentiation
DIV	integer division
MOD	calculates remainder
AND	logical and binary AND

OR	logical and binary OR
=	equal to (also applies to sets and arrays)
<>	not equal to (also applies to sets and arrays)
>	greater than (also applies to arrays)
<	less than (also applies to arrays)
<=	less than or equal to (applies to sets and arrays)
>=	greater than or equal to (applies to sets and arrays)
IN	determines if number in set

All operators consisting of letters must be enclosed in spaces.

14.1.9 Standard Functions

SIN	sine, angle in radians
COS	cosine
TAN	tangent
ARCTAN	arctangent
EXP	exponentiation of e
LN	natural logarithm
SQR	square
SQRT	square root
ABS	absolute value
TRUNC	integer portion
NOT	logical and binary NOT
PEEK	contents of memory location
ORD	number code of character
CHR	character for code
RND	random number
SUCC	successor
PRED	predecessor
LENGTH	length of a string
VAL	value of a digit string

14.1.10 Access to data structures

array name[index list]
Returns the basic element or data block of the one or more dimensional array. The indices may be expressions.

array name[start value..end value]
Returns a section of a field. The section may be defined by expressions.

record name.element name
Returns an element of a record.

pointer name ↑
Returns the element of a list to which the pointer points.

The access methods may be combined arbitrarily.

14.1.11 Predeclared names

TRUE	true, equal to −1 if converted to integer
FALSE	false, equal to 0 if converted to integer
NIL	end of list, equal to 0 if converted to integer
ERROR	error channel for disk drive with address 8, channel address 15

14.1.12 Special characters for I/O

CHR(0) End of a string
CHR(13) End of a line and number separator
CHR(32) Space and number separator

BIBLIOGRAPHY

The following books may be helpful to you in learning more about the Pascal language and programming using the Pascal 64 Package.

Bowles, Kenneth L., MICROCOMPUTER PROBLEM SOLVING USING PASCAL, Springer-Verlag, New York, 1977.

Conway, R., Gries, D., and Zimmerman, E., A PRIMER ON PASCAL, Winthrop Publishers, Cambridge, MA, 1976.

Grogono, Peter, PROGRAMMING IN PASCAL, Addison-Wesley, 1978.

Kieburtz, Richard, STRUCTURED PROGRMMING AND PROBLEM SOLVING WITH PASCAL, Prentice-Hall, Englewood Cliffs, NJ, 1978.

Jensen, Kathleen and Wirth, Niklaus, PASCAL USER MANUAL AND REPORT, Springer-Verlag, New York, 1974.

Wirth, Niklaus, SYSTEMATIC PROGRAMMING: AN INTRODUCTION, Prentice-Hall, Englewood Cliffs, NJ, 1973.

Wirth, Niklaus, ALGORITHMS + DATA STRUCTURES = PROGRAMS, Prentice-Hall, Englewood Cliffs, NJ, 1976.

Zaks, Rodney, INTRODUCTION TO PASCAL, Sybex, Berkley, CA, 1981.