TURBO-PASC'99

L. L. CONNER ENTERPRISE

Computer & Electronics

Turbo-Pasc'99 Reference Manual

THIS PAGE IS BLANK

TABLE OF CONTENTS

1. INTRODUCTION

1.1. Overview

Turbo-Pasc'99 is a compiler package which sets itself apart from conventional systems. Until now, generating high-powered programs on a standard TI-99/4a Home Computer, with a 32K memory expansion and disk drive, was a slow tedious process. And, unless a program was written using the Editor/Assembler, a program's exection speed was generally slow.

Tubro-Pasc'99 is an integrated system which combines an editor and a compiler in one program. When programming, syntax errors are located by cursor position (after a simulated compile is executed) in the editor so that correction and new compilation are possible.

The Turbo-pasc'99 system disk also contains a linker. Programs can be stored in separately compiled modules which can be linked together before running. Using this method, libraries of various routines can be established and easily linked together to form powerful applications. The linker can also generate a memory image file which can then be started independently from option 5 (Run Program File) of the Editor/Assembler.

1.2. Using This Manual

This manual provides a basic introduction to Turbo-Pasc'99 and its software environment. This manual assumes that you are already familiar with the operating conventions of the TI-99/4a home computer and that you have previous programming experience with conventional PASCAL. It is recommended that you work through the entire manual systematically from front to back to become familiar with the Turbo-Pasc'99 language.

1.3. Optional Related Materials

Other optional related materials are available which will enable you to use and exploit the power of Turbo-Pasc'99 beyond its original specifications. Additional documentation, including highly technical programming information, is also available for the novice and advanced user. The following items are (or will be) available:

- **Turbo-Pasc'99 Training Guide - this comprehensive manual helps introduce, teach, and relay a better understanding of the Turbo-Pasc'99 programming language. Although the Training Guide was written with the novice user in mind, advanced users will find its extensive explanations and examples to be extremely helpful.**
- **Turbo-Pasc'99 Guide to Assembler Interfacing - this manual provides the advanced assembler programmer with the fundamentals and specifics of interfacing assembly programs with Turbo-Pasc'99 programs. Included are the Turbo-Pasc'99 CPU and VDP RAM storage allocations, parameter passing, and programming examples.**
- **Windows'99 - a supplementary library disk of object code routines which allows windows to be used with Turbo-Pasc'99. Programs using Windows'99**

can be made more user friendly, easily managed, and given that "professional look". Up to 20 windows can be displayed concurrently and up to two full screens can be simultaneously processed. Included is a detailed manual of the routines, their functions, and how to use them from within your programs.

- Graphics and Sound Toolbox - a supplementary library disk of object code routines which allows graphic, sprite, sound, and speech commands to be used with Turbo-Pasc'99. Included is a detailed manual of the routines, their functions, and how to use them from within your programs.

Correction for Page 3, 2.1
Affix this label to page 2

TP99 Filename has been changed to: UTIL1, UTIL2, UTIL3, UTIL4. This makes the Compiler boot from Myarc Disk Controller. You may load from option 5 Editor/Assembler.

Also, the Compiler may now be run from ramdisk.

Page 3, 2.2
The Two part assembler files are to be copied form your Editor/Assembler diskette Part A.

P52 Filename is the source code for program Counter on page 52.

2. LET'S GET STARTED

2.1. Files on the System Disk

The following files are at your disposal:

- TP99 - the Turbo-Pasc'99 compiler system; it is started with the Run Program File option of the Editor/Assembler. `UTIL1, UTIL2, UTIL3, UTIL4.`
- LK99 - the linker; it is likewise started with the Run Program File option of the Editor/Assembler.
- RUNLIB - the run time system; contains all standard routines and the necessary initialization code for a compiled program.
- RUNLIBEQ - run time system equates; necessary for assembling a compiled program.
- ASSEM1 - part one of the assembler; used by the Editor/Assembler for assembling a compiled pascal program.
- ASSEM2 - part two of the assembler.

) Copy from your Editor/Assembler Diskette A

- SIEVE1 - Erathosthenes' Sieve; this is the widely known and used benchmark test for computation of prime numbers. This program has already been transformed into the executable image form by the linker and can be started with the Run Program File of the Editor/Assembler.
- WURM* - a simple game for two people; this program has not been linked and will not start as is. It serves as an example to help explain the linker and will be linked as a practice exercise in Section 6.
- VDPLIB* - a small library with the monitor routines necessary for the program WURM* to operate.

2.2. Preparing for Operation With One Disk Drive

If you have only one disk drive, it will be necessary to prepare a fresh working disk. Format a new disk and copy the following files from the Turbo-Pasc'99 system disk to it using any disk manager:

- ASSEM1
- ASSEM2
- LK99
- RUNLIBEQ
- RUNLIB

2.3. Preparing for Operation With Two or More Drives

If your have at least two disk drives, it will be necessary to prepare a fresh working disk. Format a new disk and copy the following files from the Turbo-Pasc'99 system disk to it using any disk manager:

- RUNLIBEQ
- RUNLIB

2.4. Starting the Compiler System

Once you have prepared your disks as described in Section 2.2 or 2.3, you are ready to put Turbo-Pasc'99 to work. Make sure you have your Editor/Assembler cartridge inserted into your console before attempting to use the Turbo-Pasc'99 system.

Place the Turbo-Pasc'99 system disk in drive one. From the Editor/Assembler menu, select option 5 for Run Program File and enter the filename DSK1.TP99. The editor/compiler system will then automatically load and run itself.

3. THE COMMAND INTERPRETER

The compiler system is command oriented. All functions are selected by command abbreviations which, depending on the command, may require up to one parameter.

The cursor should be located at the lower left corner of the screen after loading the file TP99. If this is not the case, your television or monitor cannot show the extreme left column. Press the space bar a couple times and the cursor will appear.

3.1. Overview Overview of Commands

The following functions, listed alphabetically, can be selected with the command interpreter:

CO : compile - activates the compiler without code generation. The source program currently in memory is checked for syntax.

CO <filename> : compile to a file - activates the compiler with code generation. Syntax check is performed on the program currently in memory and the program is translated into machine language. The result is written onto the file <filename>.

DI <filename> : delete file - the Display Variable 80 file named in <filename> is erased from the diskette.

ED : edit - activates the full-screen editor.

FI /<string> : find string - text search function for the editor which makes it possible to search for character strings in the text currently in the editor.

GO <line number> : goto - activates the editor and sets the cursor at the given line number. This allows you to easily edit the line in which a run time error occurred.

LO <filename> : load file - loads a Display Variable 80 file from disk. The format is compatible with the Editor/Assembler, so that files created with Editor/Assembler Editor can be read.

PU <filename> : purge work file - deletes the current work file from memory and makes approximately 14 Kbytes available.

QI : quit - leave the compiler system.

SA <filename> : save source file to disk - saves the file currently in memory to disk in Display Variable 80 format. This file is normally a Turbo-Pasc'99 source program. Both TI-Writer and Editor/Assembler files can be read, altered, and saved to disk using the Turbo-Pasc'99 editor.

SI : size - displays available programming space left. If there is no program in memory, the display should read 13492 bytes available.

3.2. Command Syntax

Each command consists of two letters and, depending on the command, possibly one parameter. A command may be preceded by any number of blanks, but must be followed by at least one blank. The parameter (if present) can be followed only by blanks; no other character is permitted.

A violation of these syntax rules causes the command to be ignored and the typed command line to be erased.

3.3. Command Line Editing Functions

The following editor functions keystrokes are available while entering data on the command line:

<ENTER> : causes the command to be analyzed and, if correct, executed.

<FCTN> S : (cursor left) deletes character left of the cursor.

<FCTN> 3 : (erase) deletes the command line.

<FCTN> 8 : (redo command) brings back the last command entered to the command line.

4. The Editor

Turbo-Pasc'99 provides, in addition to a powerful compiler, an editor with considerable capabilities. The most important functions of the well-known Editor/Assembler editor have been implemented. Other interesting and comfortable features have been added to make programming with Turbo-Pasc'99 easier.

In the edit mode, the screen is 80 columns wide with three overlapping 40 column windows available for displaying text. You start with the left window displaying columns 1 through 40; the center window displays columns 21 through 60, and the right 41 through 80.

4.1. Overview of Functions

The set of functions available in the editor can be divided into four distinct groups. They include the following:

- Cursor functions: function keys that reposition the cursor.
- Line editing functions: function keys that permit line-by-line editing.
- Text editing functions: function keys that permit editing of blocks containing multiple lines.
- Miscellaneous functions.

4.1.1 Cursor Functions

Cursor functions serve to reposition the cursor in the text without altering the text in any way. They include:

<ENTER> : moves the cursor to the beginning of the next line.

<FCTN> S (cursor left): moves the cursor one position to the left in the same line. The cursor cannot be moved past the left margin.

<FCTN> D (cursor right): moves the cursor one position to the right in the same line. The cursor cannot be moved past the right margin.

<FCTN> E (cursor up): moves the cursor one line up in the same column. The cursor cannot be moved past the first line of text.

<FCTN> X (cursor down): moves the cursor one line down in the same column.

<FCTN> 4 (roll up): moves the cursor one screen down; the page of text is rolled up.

<FCTN> 5 (next window): moves the cursor one window to the right; the text is moved to the left. The current window is displayed at the lower right of the screen (W1 = window 1).

<FCTN> 6 (roll down): moves the cursor one screen up; the page of text is rolled down.

<CTRL> H (home): moves the cursor to the beginning of the current line.

<CTRL> T (top): moves the cursor to the top upper of the first screen (column 1, line 1) of the text.

<CTRL> B (bottom): moves the cursor to the first column of the last line of the text.

4.1.2 Line Editing Functions

Line editing functions serve to make changes in a line of text. They include:

<FCTN> 1 (delete character): the character under the cursor is deleted and all remaining characters on the line are shifted one space left.

<FCTN> 2 (insert character): after this key has been pressed, characters are inserted and following characters are shifted to the right. Insert mode remains in effect until another editor function is activated.

<FCTN> 3 (delete line): the line in which the cursor is currently located is deleted from memory, and following lines are shifted up one line.

<FCTN> 8 (insert line): a blank line is inserted at the current cursor position and all following lines (including the one in which the cursor was located) are shifted down one line.

4.1.3 Text Editing Functions

Text editing functions serve to make changes in whole blocks of text consisting of multiple lines. These include:

<CTRL> 1 (set first marker): marks the beginning of a block. "M1" (marker 1) is displayed at the lower left of the screen.

<CTRL> 2 (set second marker): marks the end of a block. "M2" (marker 2) is displayed at the lower left of the screen. The second marker cannot appear before the first.

After one or both of the markers have been set, functions which alter the text cannot be executed. Only cursor functions can be executed. Text cannot be edited. If text is altered, the markers are deleted and need to be reset.

After the block has been marked, the following functions can be executed in addition to cursor movements:

<CTRL> B (delete block): deletes all text between markers.

<CTRL> M (move block): moves the block to the position in which the cursor is located when the function is called. That is, after the markers have been set, the cursor needs to be moved to the position where the block is to be moved before the function can be activated. Cursor position within the block itself is not permitted.

<CTRL> C (copy block): the marked block is copied onto the position of the cursor. That is, after the block has been marked, the cursor needs to be moved to the desired position before the function can be activated. Cursor position within the block itself is not permitted.

4.1.4 Other Functions

Two other functions are at your disposal:

<FCTN> 9 (back): exit the editor and return to the command interpreter.

<FCTN) = (QUIT): Keyboard input goes through a keyboard buffer which is processed by the editor. With <FCTN> =, the keyboard buffer can be cleared and further processing can be terminated. This can, for example, intercept the loss of lines of text if <FCTN> 3 (delete line) has been pressed too often.

4.1.5 Special Features

In order to increase the comfort of structured programming, two additional features were implemented:

Auto Indent: automatic cursor indentation. After <ENTER> has been pressed, the cursor is moved to a new line and positioned directly under the beginning of the text on the preceding line. If there is already text in the new line, the cursor is moved to the beginning of the text.

Autokey: automatic recognition of reserved words of the Turbo-Pasc'99 programming language. The reserved words are transformed into all capitals so that the program is easily readable and the programmer immediately recognizes if he has used a reserved word as a variable name. To make use of this function, you need to keep the <ALPHA-LOCK> key released and use only small letters. The command interpreter (Section 3.4) also requires the <ALPHA-LOCK> to be released.

4.2. Saving Source Code to a File

The command SA <filename> allows you to save the text currently in memory to a peripheral device other than cassette. Such devices include a disk and a printer. This can be done from the command line by entering a command such as this one:

```
SA DSK1.MYPROGRAM
```

The entire work space is saved in Display Variable 80 format on disk. The format is compatible with that of the Editor/Assembler, so that programs can be exchanged between the two packages.

The following commands would print the text in memory to a printer:

```
SA PIO
```

```
SA RS232.BA=1200.DA=8.PA=N
```

4.3. Loading a File

The command LO <filename> allows you to load a file from a peripheral device other than cassette (normally disk) into memory. This can be done from the command line by entering a command such as this one:

```
LO DSK1.MYPROGRAM
```

If you currently have text in memory which has not been saved, you will receive a warning message. If you do not wish to save the text, ignore the warning and type Y; otherwise type N or any other key.

Before loading, the screen will be cleared. If loading was successful, the first page of the new text will appear on the screen. In an error occurs, an error message will appear on the screen and the contents of memory will remain unchanged; the text which was in memory is still there.

In addition to file or device errors, the following errors could occur:

- The file to be loaded is too large for the available memory (memory full).
- There are control characters in the text to be loaded; all such characters are removed during loading.

There are times when the loading process is not interrupted and only part of the file will be loaded. The part that was loaded can be processed normally.

4.4. Deleting a File

The command DI <filename> allows Display Variable 80 files to be deleted from your data disks. This can be done from the command line by entering a command such as this one:

```
DI DSK1.MYPROGRAM
```

Be aware that a deleted file is permanently lost (unless you expend the work and patience to try your luck with a disk editor). The exclamation mark which is part of this command is intended to make it difficult to accidentally delete a file.

4.5. The Search Function

In order to search the text for a certain character string, leave the editor with <FCTN> 9 and enter a command such as:

```
FI /PROCEDURE Output
```

The forward search begins at the position where the cursor was before leaving the editor. If the character string is found, the system automatically returns to the editor and positions the cursor at the first character of the string. In order to continue searching or start a new search, you must leave the editor again. There are methods of continuing with the same search:

- Press <FCTN> 8 - the command appears again in the command line. <ENTER> activates the next search.
- Enter FI alone (without a parameter); the search continues for the string last entered.

4.6. Display Available Memory

In order to determine how much memory is available for editing, you need to leave the editor with <FCTN> 9 and enter the following command:

```
SI
```

The available memory in bytes will be displayed on the screen. Pressing any key will return you to the editor.

Note, however, that the size of available RAM is not the same as the number of characters which you can still type into the editor. Turbo-Pasc'99 reserved words, regardless of their length, require only one byte; a string of repeated characters requires only two bytes. Blanks following a text are ignored; however, each line of text requires two bytes of information, so even blank lines would reduce available memory by two bytes.

4.7. Clearing Editor Workspace

In order to purge memory and restore the full work space, leave the editor with <FCTN> 9 and enter the following command:

```
PU
```

If the memory contains text which has not been saved, a warning message will appear. If you do not wish to save the text, ignore the message by typing in Y; otherwise type N or any other key.

After purging, 13492 bytes RAM will be available.

5. The Compiler

This complete section is the most comprehensive of the entire reference manual. It is important that you take your time to understand and digest the information presented hereafter.

In order to present something as complex as the so-called grammar of a programming language to you, we cannot get around certain formal definitions. We have tried, however, to present these formal definitions in as readable and simple a form as possible without sacrifiicing the necessary explicitness.

Here is an example to help you understand the formal definitions:

Section 5.1.1 (Representation of Integer Constants) explains how we have to represent whole numbers in our program so that the compiler recognizes them as such. The definition looks like this:

< Integer constant >

: <digit> ...

: <sign> <digit> ...

<digit>

: 0 : 1 : 2 : 3 : 4 : 5 : 6 : 7 : 8 : 9

<sign>

: + : -

These few lines explicitly define the construction of an integer constant. The symbols in the definition are to be interpreted as follows:

< > The word within these brackets is to be replaced by the alternatives which follow it before it can be put into the text of the program. The word between "<" and ">" is called a nonterminal symbol because it is not a word or character which can be accepted by the program, but a new definition which must be further refined.

: The text between this and the next ":" represents an alternative. When multiple colons are at hand, any of the alternatives represented may be chosen. Such an alternative should be resolved from left to right. If it contains nonterminal symbols, it must be replaced by its alternatives before we can continue with the original alternatives. In this way, we will come to the characters or words which must be used in the programm text. These are called terminal symbols and are underlined in the definition.

... The symbol to the left of "...", whether terminal or nonterminal, can (but does not have to) be repeated.

Taking the concrete example of integer constants, the interpretation would look like this:

<integer constant> has two alternatives:

one or more <digit>s
one <sign> and one or more <digit>s

In order to resolve the first alternative, <digit>, a nonterminal symbol, has to be refined.

<digit> has ten alternatives:

Since all ten alternatives are terminal symbols, a character can be chosen and we can continue with the first alternative for <integer constant>.

The first alternative is complete. Legal integer constants according to the first alternative would be, for example: 1, 9548, 7872490370170242.

In order to resolve the second alternative, we must first resolve <sign>, then <digit>. Legal integer constants according to the second alternative would be, for example: -0, +8375, -6890106.

Legal integer constants considering both alternatives would be, for example: 0, +98765445987, -0, -8765786447657.

Beyond formal definition, conditions which cannot be represented syntactically sometimes have to be set; these are called semantic conditions.

In the example of integer constants, a limitation is introduced stating whole numbers may not exceed a certain range. According to syntax, a 20- digit integer constant would be correct; however semantically legal are only those between -32768 and +32767.

5.1. Lexical structure of the Turbo-Pasc'99 Language

This Section explains the lexicographic representation of integer, real and string constants as well as of identifiers (names). In addition, special symbols and comparison operators are briefly treated. You will also find out how to make comments (explanatory documentation) in a Turbo-Pasc'99 program.

Blanks, comments, and end of line (<RETURN>) serve to separate the above objects.

5.1.1. Representation of Integer Constants

Integers (whole numbers) are processed directly by the CPU (central processing unit) and must therefore match the word length of the processor. Since the TMS9900 (the CPU of the TI-99/4a) is a 16-bit processor, only numbers between -32768 and +32767 can be processed. The lexical structure of integer constants is defined as follows:

<integer constant>

: <digit> ...

: <sign> <digit> ...

<digit>

:0:1:2:3:4:5:6:7:8:9

<sign>

:+:-

In addition, the number must be in the range of -32768 to +32767.

5.1.2. Representation of Real Constants

The most important distinction of real constants as opposed to integers is that they cover a much larger range, and so they can no longer be processed directly by the CPU. This results in much longer computation time (even in the case of simple floating point operations such as addition).

The legal range of real constants is as follows:

-9.9999999999999E127 .. -1E-128
0
1E128 .. 9.9999999999999e127

Real numbers must have either a decimal point, an "E", or both so that the compiler can distinquish them from integers. Depending on the number of significant digits, a real constant of either 4, 6 or 8 bytes' length results (see Section 5.2.7 - Variable declaration).

Real numbers can be represented in two ways:

Decimal representation - no exponent is used, making it easier to recognize the actual size of the number. The lexical structure of decimal representation is defined as follows:

<decimal real constant>

: <sign> <digit> ... .

: <sign> <digit> ... . <digit> ...

: <digit> ... .

: <digit> ... . <digit> ...

Exponential representation - (scientific notation) the size of the number is determined by the exponent. This method may be less clear to the novice, but it has the advantage of being able to represent very large (1E127) or very small (1E-128) numbers.

<exponential real constant>

: <decimal real constant> E <sign> <digit> ...

: <decimal real constant> E <digit> ...

5.1.3. Representation of String Constants

String constants must appear between quotation marks (") and can contain any characters. If a quotation mark is desired in the string itself, then the quotation mark must appear double.

A string constant may not go beyond the end of a line. In order to represent a longer string, the concatenation (joining two strings) operator "&" is used. (See Section 5.2.9.10.2 - String expressions.) An empty string ("") is also possible/valid.

5.1.4. Identifiers

In contrast to reserved words (see Appendix A), identifiers are names which are chosen by the user to identify his objects (e.g., constants, variables). See Section 5.2.4 - Declarations.

Identifiers are formed as follows:

<Identifier>

: <letter>

: <letter> <character> ...

<character>

: <letter>

: <number>

: _ (underline character)

Although identifiers can extend up to 80 characters (the maximum length of a line), it is recommended that they be limited to not more than 10 characters since all characters in the identifiers are significant and available memory of the TI-99/4a is limited. Capital and small letters are not differentiated.

5.1.5. Special Symbols and Comparison Operators

In order that the compiler can recognize them, comparison operators which are made up of two special symbols (e.g., "<=") may not be separated (e.g., "< =" would be wrong). Appendix A gives a complete list of special symbols and comparison operators.

5.1.6. Comments

A comment is any text contained in the special brackets "{ }". Comments can be nested to a maximum nesting level of 65535, for example:

{This is a {nested} comment}

Compiler options (see Appendix C) are also placed in comments (e.g., {$I+}).

5.2. Language Elements of TURBO-PASC'99

5.2.1. Program Structure

A program is defined as:

<program> : <program head> <block> .

5.2.2. Program Heading

The program heading is declared as:

<program heading> PROGRAM program name ;

The program name is an identifier and has no special meaning for the syntax analysis of the rest of the program, but does have documentary value.

5.2.3. Block

The block is defined as:

<block> : <declaration part> <statement part>

5.2.4. Declaration Part

The declaration part of a block consists of the following objects:

<declaration part> :

<label declarations>

<constant definitions>

<variable declarations>

<procedure or function declarations> ...

It is to be noted that the object must be declared in this order. An identifier can be used only after it has been declared in the program.

The exceptions to the preceding statement are the predefined standard names as listed in Appendix A. Although these standard names can be redefined, we recommend, for the sake of program clarity, that such redefinition be avoided by the programmer.

There are also reserved words (key words) which cannot be used as identifiers. A list of all these reserved words also appears in Appendix A.

5.2.5. Label Declarations

Label declarations can be omitted or defined as follows:

<label declarations>

: LABEL name ;

: LABEL <names> ... name ;

<names>

: name ,

The "name" is an identifier.

It should be noted that the declared label must be set in the statement part of the program.

5.2.6. Constant Definitions

These can be omitted or defined as follows:

<constant definitions>

: CONST <constant assignment> ...

<constant assignment>

: constant name = constant value ;

"Constant name" is an identifier.

Constant names can be used in the program instead of constants; by changing the constant assignment at the beginning of a block, the value changes throughout the block. Examples of constants: 1.3434 (real), "hello" (string), etc.

Standard constants (PI, E, etc.) are likewise listed in Appendix A. These can also be redefined, but, for the sake of program clarity, its is recommended that you avoid such redefinitions.

5.2.7. Variable declarations

These can be omitted or defined as follows:

<variable declarations>

: VAR <variable declaration> ...

<variable declaration>

: variable name : <type definition> ;

: <variable names> ... variable name
: <type definition> ;

<variable names>

: variable name ,

"Variable name" is an identifier.

<type definition>

: <base type>

: ARRAY [<index spec>] OF <base type>

: STREAM [integer constant]

: BLOCK [integer constant]

: RELATIVE [integer constant]

<index spec>

: integer constant

: <integer constants> ... integer constant

<integer constants>

: integer constant ,

<base type> :INTEGER

: REAL [integer constant]

: BOOLEAN

: STRING [integer constant]

All integer constants must be positive.

In the ARRAY declaration, the integer constants determine the dimensions of the array. Dimensioning of the array always begins at 0.

In the FILE declarations STREAM, BLOCK and RELATIVE, the integer constant gives the logical record length of the file. This length is limited to 255. The meaning of this declaration is explained in Section 5.5.3 - File I/O. A file variable require s 2 bytes in RAM.

The INTEGER declaration requires exactly 2 bytes in RAM. This data type can be processed directly by the CPU.

The REAL declaration permits only the integer constants 4, 6 or 8. This determines the size in bytes of memory allocation for the real variable. The smaller this value, the faster the programs runs; however, the precision in significant digits is d iminished.

The BOOLEAN declaration requires 1 byte of memory allocation and can be processed directly by the CPU.

The STRING declaration limits the integer constant, and thereby the length of the string and memory allocation, to 255. One additional byte of memory is allocated to mark the length of the string.

5.2.8. Procedure and Function Declarations

This part can either be omitted or is defined as follows:

<procedure or function declaration>

: PROCEDURE procedure name <formal parameters> ; <body>

: FUNCTION function name <formal parameters> ; : <base type> ;
<body> ;

Procedure and function names are identifiers.

The formal parameter list can either be omitted or is defined as follows:

<formal parameters>

: (<formal>)

: (<formals> ... <formal>)

<formals>

: <formal> ;

<formal>

: VAR <formal parameter spec>

: <formal parameter spec>

<formal parameter spec>

: parameter name : <type definition>

: <parameter list>...parameter name : <type definition>

<parameter list>

: parameter name ,

The parameter name is an identifier. Parameters can be used like variables.

The reserved word VAR in the formal parameter list means that the parameters which follow are reference parameters, i.e., a pointer is passed for the respective parameter,

not a value. As a result, every change in the formal parameter in the procedu re or function (e.g., by an assignment statement) also changes the value of the variable in the invoking program. This type of parameter is used in variable parameters.

If the reserved word VAR is not used, then the actual value of the parameter is passed. When the formal parameter is changed, the value of the variable in the invoking program is not changed. The procedure or function body can contain the following alternatives:

<body>

: <block>

: EXTERNAL

: FORWARD

The keyword EXTERNAL tells the compiler that the procedure or function is to be found in a MODULE. A module is a separate program file in which procedures or functions are stored, comparable to a library. In-depth explanation can be found in Section 5.3 - Modularizing programs.

The keyword FORWARD informs the compiler that the procedure or function is defined in program text following in the same block. As a result, the formal parameter list cannot be given again. The FORWARD declaration must be used in the case of indirect recursion (two procedures which invoke each other).

5.2.9. The Statement Part

The statement part is defined as follows:

<statement part>

: BEGIN <instructions> ... END

<instructions>

: <statements> ;

<statements>

: label : <statement>

: <statement>

The label must be declared in the same block in which it is set (see Section 5.2.5 - Label declaration). The statement can either be omitted or is defined as follows:

<statement>

: <assignment statement>

: <procedure invocation>

: <goto statement>

: <compound statement>

: <if statement>

: <repeat statement>

: <while statement>

: <for statement>

: <case statement>

5.2.9.1. The Statement

<assignment statement>

: <variable> := <expression>

: function name := <expression>

<variable>

: variable name

: array variable name [<index expression>]

<index expression>

: <expression>

: <expressions> ... <expression>

<expressions>

: <expression> ,

Index expressions must be of the type integer. (More on expressions can be found in 5.2.9.10.) The following assignment statements are legal:

integer to integer

boolean to boolean

real[n] to real[m] n,m IN [4, 6, 8]

real[n] to integer n IN [4, 6, 8]

integer to real[n] nIN [4, 6, 8]

string[n] to string[m] 0 < = n,m < = 255

Assignment statements can be made only with base types. The compiler fits the length of string and real types to their assignment, i.e., if a longer type is assigned to a shorter one, the longer one is truncated.

In assignments of real to integer types, run time errors can occur (see Appendix B) if the real value is too large to be converted to an integer value.

Assignment to a function name can only occur within the function block. Thereby the return value of the function is determined.

5.2.9.2. Procedure Invocation

<procedure invocation>

: procedure name (<actual parameter list>)

The actual parameter list (in the invoking program) must contain the same number of parameters as the formal parameter list (in the declaration of the invoked procedure).

<actual parameter list>

: <actual parameter>

: <actual parameters> ... <actual parameter>

<actual parameters>

: <actual parameter> ,

<actual parameter>

: <expression>

: <variable>

If a reference parameter (key word VAR) is declared in the formal parameter list, then the corresponding actual parameter must be a variable and cannot be an arithmetic expression.

In general, the types in actual and formal parameters have to match. As in assignment statements, length adjustment is possible with real and string base types, but only with parameters passed by value. Reference parameters and arrays require an exact match, including length.

In passing an array, only the name of the array (without index values) is given as actual parameter. We recommend passing arrays as reference parameters in order to save memory and run time.

File variables are likewise passed by simply giving the file name.

5.2.9.3. The Goto Statement

<goto statement>

: GOTO label

The label must be in the same or in an hierarchically higher block. Since the language Turbo-Pasc'99 offers elements of structured programming (e.g., WHILE, CASE, REPEAT), so that the goto statement becomes superfluous in most cases since it reduces the clarity and readability of a program.

5.2.9.4. The Compound Statement

<compound statement>

: BEGIN <statements> ... END

With the help of this statement, multiple statements can be combined into one.

5.2.9.5. The IF Statement

<if statement>

: IF <expression> THEN <statement>

: IF <expression> THEN <statement> ELSE <statement>

The expression must be boolean. No semicolon may appear before the ELSE. The result would be that the IF statement would be concluded after the THEN branch and the ELSE could be interpreted as a new statement, which would lead to a syntax error.

5.2.9.6. The REPEAT Statement

<repeat statement>

: REPEAT <statements> ... UNTIL <expression>

The expression must be boolean. The repeat loop is repeated until the boolean expression is TRUE. The repeat loop is executed at least one time because the condition is tested at the end of the loop.

5.2.9.7. The WHILE Statement

<while statement>

: WHILE <expression> DO <statement>

The expression must be boolean. The while loop is executed until the value of the expression is FALSE. If the expression is FALSE at the beginning, the while loop is not executed at all (in contrast to the repeat loop).

5.2.9.8. The FOR statement

<for statement>

: FOR variable := <expression> TO <expression>
DO <statement>

: FOR variable := <expression> DOWNTO <expression>
DO <statement>

The control variable and the expressions must be integer types. The key word TO means that the loop counts upwards; DOWNTO means that the loop counts downwards. Both are in steps of 1.

In order to avoid logical program errors, it is urgently advised that control variables be declared locally (see Section 5.4 - Scope of Identifiers) as simple integer variables.

5.2.9.9. The CASE Statement

<case statement>

: CASE <expression> OF <case body> END

The expression must be of the integer type.

<case body>

: <case alternative>

: <case alternative> ...

<case alternative>

: <case label list> : <statement> :

<case label list>

: <case label>

: <case labels> ... <case label>

<case labels>

: <case label> ,

<case label>

: integer constant

: <interval>

<interval> : integer constant .. integer constant

Case alternative are processed sequentially. An alternative is fulfilled as soon as the value of the expression matches one of the constants or lies within an interval. As

soon as this is the case with one alternative, the following statement is ex ecuted and the case statement is concluded.

If none of the case alternatives is fulfilled, a run time error is produced. The following trick might serve you as an error trap to catch all other alternatives. Let your last case alternative be:

minint .. maxint : <statement>

These are standard constants (see Appendix A).

5.2.9.10. Expressions

Expressions consist of operands (constants, variables, etc.), operators (+,-,*,/,etc.) and parentheses. A type is always connected with an expression. Operators are divided into the hierarchy levels 0, 1, 2, and 3.

Evaluation of expressions:

- Operators of a lower hierarchy level are evaluated before those of higher levels.
- Operators are evaluated from left to right.
- Parentheses override priority rules and are evaluated first.

Level 3:

<expression>

: <simple expression> <comparison operator> <simple expression>

<comparison operator>

: < : > : >= : <= : <> : =

Level 2:

<simple expression>

: <sign> <term>

: <sign> <term> <additon> ...

: <term>

: <term> <additon> ...

<additon>

: <addition operator> <term>

<sign>

: + : -

Level 1:

<term>

: <factor>

: <factor> <multiplication> ...

<multiplication>

: <multiplication operator> <term>

<multiplication operator>

: * : / : DIV : MOD : AND

Level 0:

<factor>

: <variable>

: <constant>

: (<expression>)

: NOT <factor>

: function name (<actual parameter list>)

The same conventions count for invocation of a function as for a procedure (see Section 5.2.9.2 - Procedure Invocation).

5.2.9.10.1 Arithmetic Expressions

Arithmetic expressions have a value of the integer or real type. The operators need to be type compatible with the operands. The following rules of combination are in force:

Level 2:

integer +,- integer --> integer

real[n] +,- integer --> real[n]

real[n] +,- real[m] --> real[max(n,m)]

Level 1:

integer *,DIV,MOD integer --> integer

integer / integer --> real[4]

real[n] *,/ integer --> real[n]

real[n] *,/ real[m] --> real[max(n,m)]

5.2.9.10.2 String Expressions

String expressions have a value of the type string. The following rules of combination are in force:

Level 2:

string[n] & string[m] --> string[min(n+m,255)]

The operator "&" is the concatenation operator for strings.

5.2.9.10.3 Logical Expressions

Logical expressions have values of the base type boolean, i.e., either true or false. The following rules of combination are in force:

Level 3:

integer =,<,<=,>=,<>,> integer --> boolean

real[n] =,<,<=,>=,<>,> integer -->boolean

real[n] =,<,<=,>=,<>,> real[m] --> boolean

string[n] =,<,<=,>=,<>,> string[n] --> boolean

Level 2:

boolean OR boolean --> boolean

Level 1:

boolean AND boolean --> boolean

Level 0:

--- NOT boolean --> boolean

5.3. Modularizing Programs

This section will explain how to reduce programs into smaller modules which can be saved in separate files and independently compiled. In addition, the possible communication mechanisms between modules will be discussed.

5.3.1. Main Modules

How to construct a main module is described in Section 5.2 - Language Elements of Turbo-Pasc'99. Main modules must alway begin with the key word PROGRAM. Furthermore, every program must contain such a module since the execution of any Turbo-Pasc'99 program begins and ends within the main module.

5.3.2. Library Modules

Library modules are comprised of a collection of procedures and functions which share a common characteristic (graphic utilities, file management utilities, etc.). Though similar in construction to the main module, library modules contain no statement part.

Construction syntax:

<module>

: <module head> <module declaration part>

<module head> : MODULE module name ;

The module name is an identifier whose sole purpose is documentation.

<module declaration part> :

<constant definitions>

<variable declarations>

<procedure and function declarations> ...

These objects are declared exactly as explained in Section 5.2 - Language Elements of Turbo-Pasc'99. You should note, however, that variable declaration has a different meaning here. It serves the purpose of communication with other modules in the sense of COMMON ranges.

5.3.2.1. Exporting Procedures and Functions

The procedures and functions in the module declarative part are exported either to other library modules or to the main module (i.e., made accessible to other modules). This correspondence is given by the key word EXTERNAL.

The following precautions should be noted:

- In exporting/importing of identifiers, only the first six characters (excluding the character "_") are considered.
- Identifiers for procedures and functions must be unambiguous.
- If an identifier is imported via EXTERNAL by a main module or a library module, it must also be exported by a library module.

- If a procedure or function in a declarative part of a module is declared EXTERNAL, it can import but not export.
- If these precautions are not heeded, errors may arise in the linking of the modules (see Section 6 - The linker).

5.3.3. Communication by Parameter Passing

Parameter passing occurs as described in Section 5.2.9.2 - Parameter Passing. It should be noted that assignment compatibility is not tested, leaving the programmer to assure that his list of formal parameters for the EXTERNAL definition corresponds one to one with the formal parameter list of the module.

5.3.4. Communication through COMMON Ranges

Here, a variable declaration in the declarative part of the module resides in the same central storage address as a variable declaration of the main program in the main module. For example, if both modules declare an integer variable, this variable could be assigned a value in the main program and the value could be changed in the library module.

This type of communication is fast because, instead of parameters being passed, reference is made directly to the storage cells. However, the COMMON approach must be used with caution. In no case can more variables in the main module be overlapped than exist there. The program would subsequently crash.

The most infallible method is to employ the same variable declarations as in the highest declaration level in the main module. Other names can naturally be used. The important thing is that the type declaration and the order of declaration is the same.

The following examples are intended to make the module concept clearer:

Communication by parameter passing:

```
PROGRAM Main; {main module}
  VAR result: INTEGER;
  { Procedures / functions imported from a library module. }
  PROCEDURE  add (a,b:INTEGER; VAR sum:INTEGER); EXTERNAL;
  PROCEDURE  sub (a,b:INTEGER; VAR diff:INTEGER); EXTERNAL;
  FUNCTION   mul (a,b:INTEGER) : INTEGER; EXTERNAL;

  BEGIN
    add (5,7,result);
    sub (3,4,result);
    Writeln(mul (3,4));
  END.

MODULE maths; {library module}
  { Procedures / Functions exported to other modules }
  PROCEDURE add (x,y:integer; VAR z:integer);
  BEGIN
    z:= x+y;
  END;
  PROCEDURE sub (x,y:integer; VAR z:integer);
  BEGIN
    z:= x-y;
  END;
```

```
FUNCTION mul (x,y: INTEGER): INTEGER;
BEGIN
   mul := x*y;
END;
```

Communication through COMMON ranges:

```
PROGRAM main; {main module}
   VAR n:integer;
      arra: ARRAY [100] OF STRING[10];
      i : INTEGER;
   PROCEDURE outcom; EXTERNAL;

   BEGIN
      readln(n);
      for i: = 0 to n do readln(arra[i%];
      outcom;
   END.

MODULE out; {library module}
   { The COMMON Variables must be declared in the same sequence as in the main
   module. The names need not necessarily be the same.}

   VAR  k:INTEGER; { k is superimposed on the variable n in the main module. }
   f:ARRAY[100] OF STRING[10];  { f is superimposed on the variable arra in the main
   module.}

   PROCEDURE outcom;
   VAR i:integer;
   BEGIN
      FOR i: = 0 to k do writeln(f[i]);
   END; .
```

5.4. Scope of Identifiers

The scope of an identifier can be local or global. The following scope rules make the use of identifiers clear:

1) An identifier may be used only in a procedure (function) in which it has been declared (it is then local to that procedure) and in all procedures and functions declared within that procedure (it is global to these procedures). The identifier is not known to procedures outside the nesting level of the declaration.

Identifiers declared in the main program are known to all procedures and functions.

Standard identifiers (see Appendix A) can be thought of as declared in an invisible block enclosing every program and therefore known to all parts of the program.

2) When an identifier is declared anew in a subordinate procedure or function, it is a distinct entity which only coincidentally has the same name. The scope of the originally declared identifier does not extend into this block. Both rules 1 and 2 apply for the new identifier.

Example:

```
PROCEDURE Level1;
   VAR i,j,k : INTEGER;

PROCEDURE Level2;
   VAR i,j,l : BOOLEAN;

   BEGIN {Level 2}
      { valid: i,j,l : BOOLEAN k : INTEGER }
   END;

   BEGIN {Level1}
   { valid: i,j,k : INTEGER }
   END;
```

Turbo-Pasc'99 contains a so-called dynamic storage model. By clever arrangement of a program in procedures and functions, considerable storage can be saved.

Since the scope of variables declared in a block does not extend outside that block, the storage occupied by that variable is also freed (made available for other purposes) upon leaving the procedure/function. Furthermore, this means that the variab le values are undefined with each new invocation since another procedure might have used the same storeage location and altered the values of the variables.

If a procedure or function requires a variable whose value is to be retained through several invocations of this procedure/function, then the variable needs to be declared in the outer block containing this procedure/function or in the main program itself as a global variable.

5.5. Standard Procedures and Functions

This section explains all procedures and functions which are at your disposal in the run time system (under RUNLIB on the Turbo-Pasc'99 disk). This run time system is practically a just like a library (see Section 5.3- Modularizing Programs), yet it does not have to be linked as module to your program. Since every program that you write will use the run time system, Turbo-Pasc'99 saves you the trouble of linking it yourself.

As you may have realized, RUNLIB is a memory image file, which sets it apart from normal modules, which are Display Fixed 80 files created by the assembler. Memory image files load faster than Display Fixed 80 files.

The run time system contains the following function groups:

- SCREEN - special routines for monitor control.
- CONSOLE I/O - procedures for keyboard input and monitor output.
- FILE I/O - procedures and functions for file management.
- MATH - mathematic functions.
- STRINGS - functions for manipulation of strings.
- CONVERSION - functions for comfortable conversion of data types.
- MISCELLANEOUS - other procedures and functions.

The following pages explain the declaration of standard procedures and functions and provide short examples for each.

Certain standard procedures create a problem in declaration because, as so- called generic procedures, they must accommodate any number of parameters of any type.

Example: Write(3.14); is correct - but so is
Write("Hello", TRUE, 12, 3.14);

The parameter list for generic procedures is given as "GENERIC".

5.5.1. Screen

SCREEN contains procedures for organizing the screen. The screen in BASIC contains 768 characters (24 x 32) and in the Editor/Assembler Editor, 960 (24 x 40). The 32 character display can assign one of 16 colors to character groups up to 8 bytes, but less information can be displayed than on the other display. The 40 character display offers only two colors to choose from, but can display text more clearly.

The following procedures were implemented in order to allow you to choose either a 32 or 40 character display.

5.5.1.1. Graphics

Declaration: PROCEDURE Graphics;

Explanation: Changes to 32 character screen; if the graphics screen is already activated, the invocation is ignored, otherwise the screen is cleared.

Example: Graphics;

5.5.1.2. Text

Declaration: PROCEDURE Text;

Explanation: Changes to 40 character screen; if the text screen is already activated, the invocation is ignored, otherwise the screen is cleared.

Example: Text;

5.5.1.3. Cls

Declaration: PROCEDURE Cls;

Explanation: The screen is cleared, automatically matched to the activated 32 or 40 character screen.

Example: Cls;

5.5.1.4. Screen

Declaration: PROCEDURE Screen (foreground color, background color : INTEGER);

Explanation: Sets the general foreground and background colors. On the 32 character screen, all character groups and the frame are set to this color combination; the 40 character screen receives the color combination as foreground/background. Both parameters must be in range of 1 through 16, and adhere to the BASIC color.

Example: Screen (16,5);

5.5.2. Console I/O (Input and Output)

Console I/O allows you to communicate with your program. The currently active screen is matched automatically.

5.5.2.1. Cursor

Declaration: PROCEDURE Cursor (row, column: INTEGER);

Explanation: Positons the cursor for input/output operations; row must be in the range 1 through 24; column mst be in the range 1 through 32 for the 32 character screen or 1 through 40 for the 40 character screen.

Example: Cursor (12, 20);

5.5.2.2. Write

Declaration: PROCEDURE Write (GENERIC);

Explanation: Prints out desired text on screen. The output can also be formatted with the following options:

integer: field width (1 format option)
boolean: field width (1 format option)
string: field width (1 format option)
real: field width (1 format option)
real: field width/accuracy (2 format options)

Truncating (cutting off) options whose format option is too small is not possible. Instead the format specification is ignored.

Examples (without formatting):

```
Write (12);
Output: 12

Write (1.23);
Output: 1.23E+000

Write ("This is text");
Output: This is text

Write (true);
Output: TRUE
```

Examples (with formatting):

```
Write (12:4);
Output:  12

Write (-1.23:5:2);
Output: -1.23

Write ("Expansion", 15);
Output: Expansion_____
```

5.5.2.3. Writeln

Declaration: PROCEDURE Writeln (GENERIC);

Explanation: Exactly as Write but with an additional carriage return to positon cursor at the start of the next line.

Example: Writeln (1, 2.05, "three", 20);

5.5.2.4. Read

Declaration: PROCEDURE Read (VAR GENERIC);

Explanation: Assignes values to identifiers as they are read from the keyboard. The actual parameter list may contain only reference parameters.

Lexical correctness for integer and real constants is checked at input (see Section 5.1 - Lexical structure of Turbo'-Pasc'99). If an error occurs, the input value is ignored and must be entered again.

A format option can be specified at input; it defines the length of the input field.

String constants - which are input without quotation marks - have an additional condition in their format option: If no format option is given, all succeeding blanks are ignored; if a format option is given, then blanks are added to the end until the given length is achieved.

Every input must be confirmed with the <ENTER> key since this is the only legal character for separating input. Three parameters, for example, would require the <ENTER> key to be pressed three times.

Example: Read (name, address: 20, age 3);

Input: Meyer <ENTER>
123 Main St. <ENTER>
123 <ENTER>

5.5.2.5. ReadLn

Declaration: PROCEDURE ReadLn (VAR GENERIC);

Explanation: Operates exactly as Read, but with a carriage return after all parameters have been input (cursor repositioned at start of next line).

Example: ReadLn (name, address, age);

5.5.3. File I/O (Input and Output)

File I/O contains the routines that you will need for file management. Both sequential (stream, sequential) and direct address (relative, DA - Direct Address) files are supported.

Declaring file variables in Section 5.2.7 - Variable Declaration. The meaning of the declarations is as follows:

- STREAM - sequential file with variable record length. Logical record length is set in the declaration in the form of an integer constant. Only string type objects can be written on such a file. If you need to write arithmetic expressions onto a stream file, they first need to be converted into strings with the included conversion functions (see Section 5.5.6 - Conversion). Likewise, only strings can be read form a stream file.
- BLOCK - sequential file with fixed record length. Logical record length is set in the declaration in the form of an integer constant. Objects of all base types can be written onto such a file. The value of the objects is stored in its internal representation, so each object requires as much storage as given in its declaration. Since the object type is not stored in the file, it is up to the programmer to ensure the compatibility of data types. The system monitors the logical record length and recognizes if too little or too much information is read from a data entry. This would lead to a run time error.
- RELATIVE - DA file with fixed file length. The same conditions as with BLOCK apply.

5.5.3.1. Open

Declaration: PROCEDURE Open (VAR FileVar : FileType;

FileName : STRING[n];
OpenMode : INTEGER;

Explanation: Assignment of a physical file to a logical file definition of any type and opening of that file.

FileVar is the logical file name and must be of the STREAM, BLOCK or RELATIVE type (see Section 5.2.7 - Variable Declaration).

FileName is the physical file name and gives the name of the peripheral device and the file located there.

OpenMode gives the direction of data tranfer (INPUT, OUTPUT, APPEND). These names are accessible as standard constants (see Appendix A).

Example: VAR File1 : STREAM[80];
...
Open (File1, "DSK1.MYPROGRAM", INPUT);

5.5.3.2. Put

Declaration: PROCEDURE Put (VAR FileVar : FileType; GENERIC); {for STREAM}

PROCEDURE Put (VAR FileVar : FileType; VAR GENERIC); {for BLOCK, RELATIVE}

Explanation: Output of expressions onto a file. All parameters are grouped alongside one another and written as a logical record onto the file. STREAM files add an automatic line feed if the logical record extend beyond the record length; BLOCK and RELATIVE files generate a run time error.

Permissible parameter types depend on the file type:

STREAM - string only.
BLOCK - all base types.
RELATIVE - all base types.

In order to increase dependability in file communication, BLOCK and RELATIVE files require that only variables be used as parameters. For example, in calculating record length, you reserve a field of the type REAL[8]; an attempt to store the constant 3.1 in this field would lead to a run time error since the compiler assigns this constant REAL[4].

Example: Put (File1,"This is great,","isn't it?");

5.5.3.3. PutLn

Declaration: PROCEDURE PutLn (VAR FileVar : FileType; GENERIC); {for STREAM}

PROCEDURE PutLn (VAR FileVar : FileType; VAR GENERIC); {for BLOCK, RELATIVE}

Explanation: The same conditions apply as with Put. The compiler also tests whether the given parameters cummulatively make up exactly the established record length for a BLOCK or RELATIVE file; otherwise a run time error appears on the screen. Afterwards, the next record in the file is prepared for processing.

Example: PutLn (File1, "This is the end.");

5.5.3.4. Get

Declaration: PROCEDURE Get (VAR FileVar : FileType; VAR GENERIC);

Explanation: Reads desired values fron a file.

When the end of a logical record has been reached before all parameters have been assigned a value, a STREAM file will automatically continue to read the next record. BLOCK and RELATIVE files, however, will produce a run time error. A run

time error will likewise occur if an attempt is made to read beyond the end of the file. The function EOF permits checking to see if more records are available.

Permissible paramter types depend on the flie type:

STREAM - only strings.
BLOCK - all base types.
RELATIVE - all base types.

Example: Get (File1, Name, Address, Age);

5.5.3.5. GetLn

Declaration: PROCEDURE GetIn (VAR FileVar: FileType; VAR GENERIC);

Explanation: The same conditions apply as with Get. The compiler also tests whether the given parameters cummulatively make up exactly the established record length for a BLOCK or RELATIVE file; otherwise a run time error appears on the screen. Afterwards, the next record in the file is prepared for processing.

Example: GetLn (File1, NearEndd, Endd);

5.5.3.6. Seek

Declaration: PROCEDURE Seek (VAR FileVar : FileType; RecNum : INTEGER);

Explanation: Pprepares the record with the given record for processing. Seek can only be used with RELATIVE files.

Example: Seek (DA_File1, 12);

5.5.3.7. EOF (End of File)

Declaration: FUNCTION EOF (VAR FileVar : FileType) : BOOLEAN;

Explanation: Determines whether the end of file of a sequential file (STREAM or BLOCK) has been reached. The function is meaningful only for INPUT files since, if EOF returns TRUE, the next attempted read would produce a run time error.

Example: IF EOF (File1) THEN

WriteLn ("All records have been read");

5.5.3.8. EOLN (End of Line)

Declaration: FUNCTION EOLN (VAR FileVar : FileType) : BOOLEAN;

Explanation: Determines whether the end of a logical record in a STREAM file has been reached. This is particularly useful since STREAM files have a variable record length. The function is meaningful for INPUT files only.

5.5.3.9. Close

Declaration: PROCEDURE Close (FileVar : FileType);

Explanation: Closes a file. If this procedure is not used, all files are automatically closed at the end of program execution.

Example: Close (File1);

Close (DA_File1);

5.5.4. Math

MATH contains the usual floating decimal functions as known from BASIC. The following table shows which arguments are legal and what type of value they return. The actual meaning of these functions can be found in most any BASIC handbook.

Function	Meaning	Argument	Returns
ABS	Absolute Value	INTEGER	INTEGER
ABS	Absolute Value	REAL[n]	REAL[n]
ARCTAN	Arc Tangent	INTEGER	REAL[4]
ARCTAN	Arc Tangent	REAL[n]	REAL[n]
COS	Cosine	INTEGER	REAL[4]
COS	Cosine	REAL[n]	REAL[n]
EXP	Exponent	INTEGER	REAL[4]
EXP	Exponent	REAL[n]	REAL[n]
INT	Integer	INTEGER	REAL[4]
INT	Integer	REAL[n]	REAL[n]
LN	Natural Log	INTEGER	REAL[4]
LN	Natural Log	REAL[n]	REAL[n]
SIN	Sine	INTEGER	REAL[4]
SIN	Sine	REAL[n]	REAL[n]
SQRT	Square Root	INTEGER	REAL[4]
SQRT	Square Root	REAL[n]	REAL[n]
TAN	Tangent	INTEGER	REAL[4]
TAN	Tangent	REAL[n]	REAL[n]

5.5.5. Strings

String functions provide diverse routines for handling character strings. Most of the string functions listed operate just like their BASIC equivalents.

5.5.5.1. ASC

Declaration: FUNCTION ASC (CHARACT : STRING[1]) : INTEGER

Explanation: Returns ASCII value of the character.

Example: WriteLn (ASC ("A")); {returns 65}

5.5.5.2. CHR

Declaration: FUNCTION CHR (CharCode : INTEGER) : STRING[1];

Explanation: Returns character whose ASCII value is given; inverse of ASC.

Example: WriteLn (CHR (65)); {returns A}

5.5.5.3. LEN

Declaration: FUNCTION LEN (StringA : STRING[n]) : INTEGER;

Explanation: Returns the length of the string.

Example: WriteLn (LEN ("Hello")); {returns 5}

5.5.5.4. SEG

Declaration: FUNCTION SEG (StringA : STRING[n]; StartPos : INTEGER; Span : INTEGER) : STRING[n];

Explanation: Returns a segment (substring) which begins at the StartPos of the given StringA and extends a Span characters.

Example: WriteLn (SEG ("Hello", 2, 3); {returns "all".}

5.5.6. Conversion

Conversion provides powerful routines for converting objects of various types. The conversion of real or integer types to strings can be handled as described under formatting in Section 5.5.2.2 - Console I/O.

5.5.6.1. CIR

Declaration: FUNCTION CIR (WholeNum : INTEGER) : REAL[4];

Explanation: Converts an Integer expression to a Real expression with a length of 4.

Example: WriteLn (CIR (12));

5.5.6.2. CIS

Declaration: FUNCTION CIS (WholeNum : INTEGER) : STRING[n];

Explanation: Converts an Integer expression to a String with a length of n. A format can be specified (see Section 5.5.2.2 - Write).

Example: Writeln (CIS (12:4));

5.5.6.3. CRI

Declaration: FUNCTION CRI (RealNum : REAL[n]) : INTEGER;

Explanation: Converts a Real expression to an Integer with rounding if necessary. A real number which is too large results in a run time error.

Example: WriteLn (CRI (1.2));

5.5.6.4. CRS

Declaration: FUNCTION CRS (RealNum : REAL[n]) : STRING[m];

Explanation: Converts a Real expression to a String n long. Format can be specified as in 5.2.2.2 - Write.

Example: WriteLn (CRS (1.2:10:1));

5.5.6.5. CSI

Declaration: FUNCTION CSI (StringA : STRING[n]) : INTEGER;

Explanation: Converts a String expression n long to an Integer. The integer string must be lexically correct (see Section 5.1.1).

Example: WriteLn (CSI ("12"));

5.5.6.6. CSR

Declaration: FUNCTION CSR (StringA : STRING[n]) : REAL[m];

Explanation: Converts a String expression n long to a Real expression. The real string must be lexically correct (see Section 5.1.2).

Example: WriteLn (CSR ("-3.14"));

5.5.7. Miscellaneous

MISCELLANEOUS contains routines to directly read the keyboard and generate random numbers.

5.5.7.1. Key

Declaration: PROCEDURE Key (KeyNum, VAR KeyCode, VAR Status : INTEGER);

Explanation: Reads one key from the keyboard.

Example: REPEAT Key (O, T, S) UNTIL S>0;

5.5.7.2. Randomize

Declaration: PROCEDURE Randomize;

Explanation: Seeding of the random number generator. If random numbers are to be used, this routine should be called at the beginning of the program to change the sequence of random numbers with each time the program is started.

Example: Randomize;

5.5.7.3. RND

Declaration: FUNCTION RND (Peak : INTEGER) : INTEGER;

Explanation: Generates a random number between 0 and the upper limit Peak value.

Example: WriteLn (RND (32767));

6. The Linker

Turbo-Pasc'99 package includes powerful program which is used as a loader for compiled modules and to generate program files.

The first component, a loader for tagged object code, can be compared to the LOAD AND RUN option of the Editor/Assembler. And the second, the program file generator, works similarly to the save utility on the Editor/Assembler disk.

6.1. Loading the Linker

Place the diskette Turbo-Pasc'99 system disk in drive 1. Select the option RUN PROGRAM FILE from the Editor/Assembler menu and enter DSK1.LK99. The linker will proceed to load and after a short period of time its title screen will appear.

6.2. Loader for Tagged Object Code

The blinking cursor should be located on the left edge of the screen. A complete line editor with which you can enter your module names is at your disposal. The following editing functions are available:

<ENTER>: The module name is selected and the respective module is loaded from diskette.

<FCTN> S (cursor left): moves cursor one position to the left.

<FCTN> D (cursor right): moves cursor one space to the right.

<FCTN> 1 (delete character): deletes character under the cursor.

<FCTN> 2 (insert character): switches to insert mode; typed characters will be inserted until the pressing of another function key overrides insert mode.

<FCTN> 6 (begin): positions cursor at start of the current line.

<FCTN> 8 (redo): all selected modules are ignored and the cursor is positioned at the beginning of the first line.

<FCTN> 9 (back): exits the linker.

Two types of errors can occur during the loading process:

- Errors which require the last name to be re-entered (see Section 24.12.1 - Input/Output Error Codes of the Editor/Assembler manual).
- Errors which require all modules to be re-entered. A Duplicate Symbol message indicates a module was loaded more than once or modules have identical external definitions. A Memory Full message is displayed if too many modules were loaded and the available memory is exhausted.

After the desired modules have been properly loaded, the end of the link process is indicated by pressing <ENTER> alone. This can lead to the following errors:

- At Least One Module Must Be Loaded: the null entry was made before a valid module name was entered.

- Main Module Not Loaded: only library modules were loaded; a main module is missing. The cursor is positioned at the beginning of the first line to permit input of the missing main module.
- Unresolved Reference: one or more library modules were not loaded. The cursor is positioned at the beginning of the first line to permit input of the missing library module.

When module input has been properly completed, the second component of the linker is activated.

6.3. Program File Generator

You will be asked whether you want to generate a program file. If you enter Y, the line editor described above is again at your disposal so you may enter the name of your program file. After the name has been input, memory image files are generated from the original relocatable modules. These can then be loaded from the Run Program File option (#6) of the Editor/Assembler.

The generated program file is split into 8k files and the last letter of the file name is independently incremented (see the Save Utility in Editor/Assembler Handbook). It is therefore recommmended that the last character of your file name be a number (e.g., DSK1.MYPROGRAM1).

A program file is only generated if no run time errors occur (see Appendix B). If an error does occur, the program must be corrected, recompiled, and relinked until it is error-free.

6.4. Your First Linker Exercise

The Turbo-Pasc'99 diskette contains the two files WURM*, the main module of a gamr program, and VDPMOD*, a screen utility library module. In order to generate a ready-to-run program from these modules, follow these steps:

1. Load LK99 as described in Section 6.1.

2. Enter one of the two module names along using the proper drive designation. (DSK1.WURM* if the Turbo-Pasc'99 disk is in drive 1). The drive should be activated and if loading has been successful the cursor will move to the next line.

3. Enter the name of the second module ~~(DSK1.VDPMOD*)~~: (DSK1.VDPLIB*)

4. Since all necessary modules have been loaded, the null entry can now terminate this phase. With the cursor on the third input line, press <ENTER>.

6. You will now be asked whether a memory image file is to be generated. Type Y.

6. Enter the program name (DSK2.WURM1). If you are using a single disk system you will have to replace the Turbo-Pasc'99 system disk with your working disk and use DSK1.WURM1 as your filename. After this has been completed, the program file generator will generate a memory image file.

7. When asked if you want to run the program, answer N. The cursor will be set at the first input line ready to link more modules. Since we are done, press <FCTN> 9.

8. Select the Run Program File option from the Editor/Assembler menu and enter the name of your memory image file (DSK1.WURM1, or DSK2.WURM1, depending on the drive it was saved on). The program should run and be now be at your disposal.

Correction for Page 42
Affix this label

6.4. Your first Linker Exercise

3. (DSK1.VDPLIB*) not as shown

7. Starting User Programs

Programs can be started using the linker or using the Run Program File (option #5) from the Editor/Assembler menu.

7.1. Starting from the Linker

In order to use this method, you must first load the linker (see Section 6.1) and then your compiled modules (see Section 6.2). You will have the option of generating a program file (see Section 6.3). After these input prompts have been completed the screen color will change to the color red. You will then be asked whether you want to run your program. If you type in Y, the program is executed.

Be sure that a disk (like the Turbo-Pasc'99 system disk) is in drive 1 since it contains the file RUNLIB. This is a memory image file which contains the entire run time system of Turbo-Pasc'99 (all standard procedures, functions, initializing routines). If the file RUNLIB is not in drive 1, the error message "Insert RUNLIB in Drive 1" will appear. Insert the proper disk and press any key (other then <FCTN> 9 (back)) which will return you to the TI- 99/4a power-up screen) to continue.

7.2. Starting from the Editor/Assembler Menu

This method is without a doubt the easiest of the two. The prerequisite is that you generated a program file (see Section 6.3) after having loaded the modules (see Section 6.2).

Select the Editor/Assembler menu option (#5) Run Program File. Before selecting the program name, make sure that the proper disk with your program is in drive 1. After the program name has been entered, the program will start by itself.

A. Appendix A - Reserved Words

A.1. Turbo-Pasc'99 Key Words

AND	ARRAY	BEGIN	BLOCK	BOOLEAN
CASE	CONST	DIV	DO	DOWNTO
ELSE	END	EXTERNAL	FOR	FORWARD
FUNCTION	GOTO	IF	INTEGER	LABEL
MOD	MODULE	NOT	OF	OR
PROCEDURE	PROGRAM	REAL	RELATIVE	REPEAT
STREAM	STRING	THEN	TO	UNTIL
VAR	WHILE			

A.2. Standard Names

abs	append	asc	atn	chr
cir	cls	close	cls	cos
cri	crs	csi	csr	cursor
e	eof e	oln	exp	false
get	getln	graphics	input	int
key	len	ln	maxint	minint
open	output	pi	put	putln
randomize	read	readln	rnd	screen
seek	seg	sin	sqrt	tan
text	true	write	writeln	

A.3. Special Symbols and Operators

&	..	%	.	(
)	+	-	*	/
,	:	;	<	=
>	< =	< >	> =	: =

B. Appendix B - Error Messages

B.1. Lexical Errors

1 Invalid numeric constant
2 String extends beyond end of line
3 Invalid character

B.2. Syntax Errors

11 'PROGRAM' or 'MODULE' expected
12 identifier expected
13 '.' expected
14 invalid character after end of program
15 ' = ' expected
16 constant expected
17 ':' expected
18 ' ' expected
19 integer constant expected
20 '%' expected
21 'OF' expected
22 base type expected
23 ')' expected
24 '(' expected
25 ',' expected
26 operand expected
27 ';' expected
28 'BEGIN' expected
29 'END' expected
30 ': = ' expected
31 'THEN' expected
32 'DO' expected
33 'UNTIL' expected
34 'TO' or 'DOWNTO' expected

B.3. Semantic Errors

51 more than 9 procedures/funtions nested
52 identifier declared twice
53 FORWARD-reference not resolved
54 insufficient memory for program
55 declared label not set
56 incorrect array dimensions
57 invalid real type
58 invalid String type
59 identifier nott declared
60 constant not in valid range
61 conflict in types
62 identifier used incorrectly
63 integer expression expected
64 boolean expression expected
65 string expression expected

66 integer oder real expression expected
67 simple expression expected
68 working range of compiler insufficient
69 label set more than once
70 assignment to function identifier not within function
71 loop variable not declared locally
72 invalid file type

B.4. Run time errors

0-7 standard file errors (see E/A Manual Section 24.12.1)
8 file not open
9 multiple assignment of a physical file logical file name
10 more than 10 open files
11 logical record length violated
12 division by 0
13 over/underflow in arithmetic calculation
14 error in mathematic function
16 incorrect Parameter in standard procedure
17 invalid format
18 array index outside limits
19 invalid CASE Alternative
20 stack overflow

C. Appendix C - Compiler Options

With the help of compiler options, you can greatly influence the type of code generated by the compiler. Employment of such options can be compared to setting switches. In the text of a program, these switches can be turned on and off at will.

As explained in Chapter 5.1.6 - Comments - these options are located in the comments. In the case of nested comments, only those compiler options residing in the first nesting level can be recognized.

Rules for the construction of a compiler option:

<option> : $ <letter> <sign>

<letter> : B : E : A : S : I

<sign> : + : -

If these rules are violated, the compiler option will not be recognized as such and is treated as a normal comment. The letter indicates the particular compiler option. The '+' activates the option; the '-' deactivates it.

C.1. Option B: Boolean Expression Evaluation

Option B: (default $B-)

$B+ causes the evaluation of boolean expressions to terminate as soon as the value is established.

Example:

```
{ $B+ }
IF (i>10) OR (f(i)<>0) THEN;
```

If (i > 10) is satisfied, then the second expression, (f(i) < > 0), is not evaluated since the value of the IF expression is already established as true.

Activating this option can cause longer code generation in the case of complex expressions. However, run time is shortened since the entire expression does not always need to be evaluated.

C.2. Option E: Monitor Overflow/Underflow of Integers

Option E: (default $E-)

$E+ causes overflow and underflow testing in integer addition and subtraction. Activating this option causes longer code generation and should only be used if a strict test of adherence to the valid range is desired.

C.3. Option A: Monitor Array Index Overflow/Underflow

Option A: (default $A-)

$A+ monitors violation of indices with arrays. Although activation causes long code generation and a slower run time, it leads to greater security. Deactivation of this option can, in the worst case, lead to an unrecoverable program crash.

C.4. Option S: Assembler Source Code Commenting

Option S: (default $S-)

$S+ causes the Turbo-Pasc'99 source code line to be carried over as comment in the assembler source code. This option has no influence on code length or on run time performance. It does, however, cause a larger assembler source file and therefore a longer assembling time.

C.5. Option I: Variable Initialization

Option I: (default $I-)

$I+ causes all variables to be initialized, depending on type, with 0, 0.0, false, or "". This has no influence on the code length. In the case of large amounts of variables, especially large arrays, to be initialized, the run time can be slower since a procedure, for example, requires the initialization with each invocation (based on the dynamic storage model, Section 5.4 - Scope of Identifiers).

D. Appendix D - Practice Session

This section runs you through a complete practice session to help give you a feel of the Turbo-Pasc'99 system in action. Although the following practice sessions will give you a better understanding of how to manipulate your system and working diskettes, you are free to find more effective methods to suit your needs. Sections 2.2 or 2.3, depending on your disk drive configuration, should be read before going through the practice sessions.

Be sure to have your Editor/Assembler cartridge inserted into your console prior to using the Turbo-Pasc'99 system.

D.1. Single Drive Practice Session

1. Place the TP99 system disk into drive one.

2. From the Editor/Assembler menu, select option 5 for Run Program File and enter the filename DSK1.TP99. The editor/compiler system will then automatically load and run itself.

3. Replace the TP99 system disk with your working disk (that is used to store your program files).

4. Enter the editor by entering ED on the command line.

5. Enter the program COUNTER listed at the end of this section. When you are finished entering the program, depress the FCTN and 9 key at the same time to return you to the command line.

6. Activate the compiler by entering CO on the command line. Your program will be checked for proper syntax, and if it was entered correctly no errors will have been detected.

7. Enter SA DSK1.COUNTER/S to save your source code to your working disk.

8. Enter CO DSK1.COUNTER/O to activate the compiler and generate an assembly language source code file that can now be compiled using the Editor/Assembler.

9. Quit the editor/compiler system by entering Q! on the command line.

10. From the Editor/Assembler menu, select option 2 to load the assembler.

11. Enter DSK1.COUNTER/O when prompted for a source file name, and DSK1.COUNTER/L when prompted for an object file name. Press enter for the next two prompts until the 'assembler executing' message appears.

12. From the Editor/Assembler menu, select option 5 for Run Program File and enter the filename DSK1.LK99. The linker will then automatically load and run itself.

13. Enter DSK1.COUNTER/L when prompted for a module name, and press enter when the cursor appears to the right of your module name.

14. Enter yes (Y) when asked if you would like to generate a program file, and DSK1.COUNTER when asked for a program name.

15. You may then run your program, or exit from the linker. The COUNTER program that was generated by the linker may be run at anytime using option 5 (run program file) of the Editor/Assembler.

D.2. Dual Drive Practice Session

1. Place the TP99 system disk into drive one and your working disk into drive two. Your working disk will be used to store your program files.

2. From the Editor/Assembler menu, select option 5 for Run Program File and enter the filename DSK1.TP99. The editor/compiler system will then automatically load and run itself.

3. Enter the editor by entering ED on the command line.

4. Enter the program COUNTER listed at the end of this section. When you are finished entering the program, depress the FCTN and 9 key at the same time to return you to the command line.

5. Activate the compiler by entering CO on the command line. Your program will be checked for proper syntax, and if it was entered correctly no errors will have been detected.

6. Enter SA DSK2.COUNTER/S to save your source code to your working disk.

7. Enter CO DSK2.COUNTER/O to activate the compiler and generate an assembly language source code file that can now be compiled using the Editor/Assembler.

8. Quit the editor/compiler system by entering QI on the command line.

9. From the Editor/Assembler menu, select option 2 to load the assembler.

10. Enter DSK2.COUNTER/O when prompted for a source file name, and DSK2.COUNTER/L when prompted for an object file name. Press enter for the next two prompts until the 'assembler executing' message appears.

11. From the Editor/Assembler menu, select option 5 for run program file and enter the filename DSK1.LK99. The linker will then automatically load and run itself.

12. Enter DSK2.COUNTER/L when prompted for a module name, and press enter when the cursor appears to the right of your module name.

13. Enter yes (Y) when asked if you would like to generate a program file, and DSK1.COUNTER when asked for a program name.

14. You may then run your program, or exit from the linker. The COUNTER program that was generated by the linker may be run at anytime using option 5 (run program file) of the Editor/Assembler.

D.3. Sample Practice Program

```
PROGRAM counter;
  {Counts integer numbers from 1 to 100 and display them on the screen}
  VAR
    i:INTEGER
  BEGIN {counter}
    cls;
    i:=0;
    WHILE i<100 DO BEGIN
      i:=i+1;
      writeln(i);
    END; {while i}
  END. {counter}
```

```
Correction for Page 52
Affix this label

PROGRAM counter;
VAR
i:INTEGER;
BEGIN
cls;
i:=0;
WHILE i<100 DO BEGIN
i:=i+1;
writeln(i);
END;
END.
```

E. Appendix E - Sample Programs

E.1. Sieve

```
PROGRAM Sieve;
{ Eratosthenes' Sieve for prime numbers }

CONST
   Size = 8190;
   Loops = 10;

VAR
   i,
   k,
   Iter,
   Count,
   Prime: INTEGER;
   Flags: ARRAY[Size] of BOOLEAN;

BEGIN {Sieve}
   Cls;
   Cursor(3,1);
   WriteLn("Eratosthenes' Sieve");
   Cursor(7,1);
   WriteLn("Array Size : ",Size);
   WriteLn("Number of Loops : ",Loops);
   Cursor(10,1);
   WriteLn("Step Size : ");
   FOR Iter := Loops DOWNTO 1 DOBEGIN
      Cursor(10,19);
      Write(Iter:3);
      Count := 0;
      FOR i := 0 TO Size DO;
         Flags[i] := TRUE;
      FORi := 0TO Size DO BEGIN
         IF Flags[i] THEN BEGIN
            Prime := i + i + 3;
            k := i + Prime;
            WHILE k <= Size DO BEGIN
               Flags[k] := false;
               k := k + Prime;
            END;{ WHILE k }
            Count := Count + 1;
         END;{ IF Flags }
      END;{ FOR i }
      Screen (2, Iter+6);
   END;{ FOR Iter }
   Cursor(22,1);
   WriteLn(Loops," times ",Count:1," prime numbers found");
END.{ Sieve }
```

E.2. Recursive Function

```
PROGRAM Recursive_Function;
{ Computation of n! (factorial) }

VAR Numb : REAL[8];

FUNCTION Fac(n : REAL[8]) : REAL[8];
   BEGIN
      IF n < = 1.0 THEN
         Fac : = 1.0
      ELSE
         Fac : = n*Fac(n-1);
   END;

BEGIN{main program}
   REPEAT
      Write("Enter a number : ");
      ReadLn(Numb);
      WriteLn(Numb:4:0,"! = ",Fac(Numb));
   UNTIL Numb < = 0.0
END.
```

E.3. File Lister

```
PROGRAM File_Lister;
{ Output of VAR80 files on printer;
PIO printer connection can be adapted}

LABEL Done;

VAR f: STREAM[80];{file to be output}
   outf : STREAM[80];{output file}
   c: STRING[1];
   fnam : STRING[15];
   lc : INTEGER;

BEGIN
   cursor(2,1);
   writeln("* FILE-LISTER *");
   open(outf, "PIO", output);{PIO can be replaced
   for other applications}
   WHILEtrueDO BEGIN
      cursor(12,3);
      write("FILE TO LIST? ");
      readln(fnam);
      IF fnam = "" THEN
         GOTO Done;
      open(f,fnam,input);
      lc : = 1;{line count begins at 1}
      WHILE NOT eof(f) DO BEGIN
         put(outf,cis(lc:4)," "); {sends line count to printer}
         WHILE NOT eoln(f) DO BEGIN{read & print text line }
```

```
            get(f,c);
            put(outf,c);
          END;
          lc := lc + 1;
          putln(outf);{printer line feed}
          getln(f); {set file pointer to next text line}
        END;
        close(f);
        putln(outf,chr(12));{printer page feed}
    END;
  Done:{label to leave loop}
END.
```

E.4. Wurm

```
PROGRAM Wurm;

CONST
    right = 3;
    left = 2;
    limit = 10;
    up = 5;
    down = 0;
    sub = 40;
    slb = 1;
    zlb = 3;
    zub = 24;


VAR
    dir,z,s : ARRAY[2] OF INTEGER;
    displa: ARRAY[2] OF STRING;
    reply : STRING[1];
    grad,point1,point2 : INTEGER;
    goof1,goof2: BOOLEAN;

PROCEDURE peekv(row,col : INTEGER; VAR byte : STRING[1]);
    EXTERNAL;{ external assembler program }


PROCEDURE Instructions;
    VAR k,s: INTEGER;
    BEGIN
cls;
    writeln("** SNAKES"); writeln; writeln;
    writeln("TWO PLAYERS ARE FIGHTING AGAINST EACH ");
    writeln("OTHER."); writeln;
    writeln(" DON'T TOUCH THE BORDER, YOURSELF, OR ");
    writeln("THE OTHER SNAKE."); writeln;
    writeln("GAME ENDS AFTER ",limit," MISTAKES.");
    writeln;
writeln("LEFT PLAYER USES 'E','S','D','X'");
    writeln("RIGHT PLAYER USES 'I','J','K','M'");
    cursor(24,1);
```

```
    write("PRESS ANY KEY . . .");
    REPEAT
      key (0,k,s)
    UNTIL S0;
END;{ Instructions }


PROCEDURE pokev(row,col : INTEGER; byte : STRING[1]);
    EXTERNAL;{ external assembler program }


PROCEDURE score;
    BEGIN
      cursor(1,1);
      write("**SCORE** A:",point1:1," B:",point2:1);
    END;


PROCEDURE wait_till_players_ready;
    VAR k,s1,s2 : INTEGER;
    BEGIN
      cursor(18,7);
      write("BOTH PLAYERS PRESS ANY KEY!");
      REPEAT
        key(1,k,s1);
        key(2,k,s2);
      UNTIL (s10) AND (s20);
      cursor(18,7);
      write(" ");
    END;

PROCEDURE init;
    VAR i : INTEGER;
    BEGIN
      cls;
      cursor(2,1);
      write("+------------------------------+");
      FOR i:=3 TO 22DO BEGIN
        cursor(i,1);write("|");
        cursor(i,40); write("|");
      END;
      cursor(23,1);
      write("+------------------------------+");
      displa[1] := "A";
      z[1] := 10;
      s[1] := 5;
      dir[1] := right;
      pokev(z[1],s[1],displa[1]);
      displa[2] := "B";
      z[2] := 10;
      s[2] := 36;
      dir[2] := left;
      pokev(z[2],s[2],displa[2]);
    END;
```

```
PROCEDURE direction(sel : INTEGER);
  VAR
    i,k,s : INTEGER;
  BEGIN
    i := 0;
    REPEAT
      i := i + 1;
      key(sel,k,s);
    UNTIL (s > 0) OR (i(grad-1)*10);
    IF s > 0 THEN
      CASE k OF
        left,right,up,down : dir[sel] := k;
        minint..maxint :
      END;
    END;


PROCEDURE player(sel : INTEGER; VAR goof : BOOLEAN);
  VAR
    byte : STRING[1];
  BEGIN
    goof := false;
    direction(sel);
    CASE dir[sel] OF
      left: s[sel] := s[sel]-1;
      right : s[sel] := s[sel] + 1;
      up: z[sel] := z[sel]-1;
      down: z[sel] := z[sel] + 1;
    END;
    IF (s[sel] = slb) AND (s[sel] > = slb AND (s[sel] < = sub) AND
        (z[sel] > = zlb) AND (z[sel] < = zub) THEN BEGIN
      peekv(z[sel],s[sel],byte);
      IF byte " " THEN
        goof := true
      ELSE
        pokev(z[sel],s[sel],displa[sel]);
      END
    ELSE goof := true;
  END;

BEGIN
  instructions;
  REPEAT
    cls;
    cursor(2,12);
    write("################");
    cursor(3,12);
    write("# #");
    cursor(4,12);
    write("#S N A K E S#");
    cursor(5,12);
    write("# #");
    cursor(6,12);
    write("################");
    cursor(10,1);
```

```
    writeln("ENTER LEVEL:");
    writeln;
    writeln("1HELL-SNAKER");
    writeln("2TURBO-SNAKER");
    writeln("3MASTER-SNAKER");
    writeln("4AMATEUR SNAKER");
    writeln("5NOVICE SNAKER");
    writeln("6GRANNY SNAKER");
    writeln; writeln;
    writeln("WHICH NUMBER (LEVEL) DO YOU CHOOSE?");
    REPEAT
       cursor(10,14);
       read(grad:1);
    UNTIL (grad > = 1) AND (grad < = 6);
    point1 : = 0;
    point2 : = 0;
    REPEAT
       init;
       score;
       wait_till_players_ready;
       REPEAT
          player(1,goof1);
          player(2,goof2);
       UNTIL goof1 OR goof2;
       IF goof1 AND NOT goof2 THEN
          point1 : = point1 + 1;
          IF goof2 AND NOT goof1 THEN
             point2 : = point2 + 1;
          score;
       UNTIL (point1 = limit) OR (point2 = limit);
       cursor(23,1);
       write("ANOTHER GAME(Y/N) ?");
       readln(reply:1);
    UNTIL (reply = "N") OR (reply = "n");
END.
```

IMPORTANT NOTICE

This documentation and the software to which it pertains to are the licensed property of L L Conner Enterprise and are to be used only in accordance with the specifications set forth herein.

L L Conner Enterprise provides this manual "as is" without any warranty of any kind, either expressed or implied, including but not limited to the implied warranties of merchantability and fitness for a particular purpose. L L Conner Enterprise reserves the right to make improvements or changes to this product at any time without notice.

While every precaution has been taken in the preparation of this documentation and software, L L Conner Enterprise assumes no responsibility for errors or omissions nor any liability for any damages resulting from the use of the information contained herein.

CUSTOMER SUPPORT

In order to take advantage of the customer support program, you must register your copy of Turbo-Pasc'99. To register, simply fill out and mail the enclosed registration form to:

L. L. CONNER ENTERPRISE
Computer & Electronics
1521 FERRY STREET • LAFAYETTE, INDIANA 47904

The registration form must be filled out completely in order for you to be registered. Once you have registered you will be eligible to receive support for your Turbo-Pasc'99 compiler system. Please be sure to include the serial number of your copy of Turbo-Pasc'99 with all correspondence.

Because your compiler diskette is copy protected, a second compiler diskette has been included in the event the first one fails to operate properly. If you try to change the contents of either diskette in any way, whether by attempted copying (which is in violation of the law and will be handled as such) or by saving other files on the disk, it is possible that you may render the disk unusable. In the event either of the two compiler diskettes are damaged you may return the damaged original diskette(s) for replacement at a nominal charge of $5.00 each.

This should of course be written as several procedures!!!

USING TURBO PASC 99 FOR FILES.

1. USING STREAM...

STREAM has the advantage that you may output literals as well as variables. Using BLOCK you MUST use variables.

STREAM has the disadvantage that data has to be read in one character at a time, using the EOLN function, and you have to use a blank PUTLN in between each line.

The following code WORKS!!! NOTE- if you try to use a variable length less than 40, the file will be opened as DV40 anyway!

I have used a generic format which means that there are unused variables in there and printed remarks you can leave out, but a very useful way of bebugging programs in this language- putting in your own form opf TRACE to see where hang ups occur!

```
PROGRAM files1;

VAR name:STREAM[40];
    in1,in2,in3,out1,out2,out3:STRING[12];
    cn:STRING[1];
    a,b:INTEGER;
BEGIN
cls;
write("   enter a string:");
readln(in1);
write("   enter an integer:");
readln(in2);
write("   enter a LARGE whole number");
readln(in3);

writeln("about to open file");
open(name,"DSK1.FILE",output);
writeln("file open");

putln(name,in1);
putln(name);
putln(name,in2);
putln(name);
putln(name,in3);
putln(name);
writeln("3 records writ. now close");
close(name);

writeln("file closed. now to reopen");
open(name,"DSK1.FILE",input);
writeln("file open.now to read");
writeln("reading string:");

 WHILE NOT eoln(name) DO BEGIN
 get(name,cn);
 write(cn);
 END;
writeln("  first file read.now 2nd");

 getln(name,cn);
```

```
  WHILE NOT eoln(name) DO BEGIN
  get(name,cn);
  write(cn);
  END;
writeln("` thats 2nd now 3rd");

  getln(name,cn);
   WHILE NOT eoln(name) DO BEGIN
   get(name,cn);
   write(cn);
   END;
writeln(" ");
writeln("THREE RECORDS WRITTEN AND READ");
close(name);
REPEAT
 key(0,a\,b);
UNTIL (b<>0);
END.
```

```
PROGRAM files2;

VAR nams:BLOCK[16];
    nami:BLOCK[2];
    namr:BLOCK[8];
    in1,out1:STRING[15];
    in2,out2:INTEGER;
    in3,out3:REAL[8];
    cn:STRING[1];
    a,b:INTEGER;
BEGIN
cls;
write("   enter a string:");
readln(in1);
a:=len(in1);
writeln(" original length ",a);
FOR b:=0 TO 15-a DO BEGIN
  in1:=in1&" ";
END;
writeln(" string length:",len(in1));
writeln("   enter an integer:");
readln(in2);
write("   enter a LARGE whole number");
readln(in3);

writeln("about to open file");
open(nams,"DSK1.FILES",output);
writeln("file open");
writeln("sending string");
putln(nams,in1);
putln(nams,in1);
close(nams);

open(nami,"DSK1.FILEI",output);
writeln("sending integer");
putln(nami,in2);
putln(nami,in2);
close(nami);
```

4

```
writeln("OPENING real file  ");
open(namr,"DSK1.FILER",output);
writeln("  sending real number");
putln(namr,in3);
putln(namr,in3);
writeln("3 records writ. now close");
close(namr);
writeln("file closed. now to reopen");
open(nams,"DSK1.FILES",input);
writeln("file open.now to read");

writeln("reading string:");
 getln(nams,out1);
 writeln(out1);
 getln(nams,out1);
 writeln(out1);
 close(nams);
writeln("  first file read.now 2nd");

 open(nami,"DSK1.FILEI",input);
 getln(nami,out2);
 writeln(out2);
 getln(nami,out2);
 writeln(out2);
 close(nami);
writeln("  thats 2nd now 3rd");

 open(namr,"DSK1.FILER",input);
 getln(namr,out3);
 writeln(out3);
 getln(namr,out3);
 writeln(out3);
writeln(" ");
 close(namr);
writeln("THREE RECORDS WRITTEN AND READ");
REPEAT
 key(0,a,b);
UNTIL (b<>0);
END.
```

Chelmsford.

3 December 1989

Dear Stephen,

Very sincere thanks for your prompt assistance, which gave me EXACTLY the clues I needed to make sense of the system. As it appears to be totally non-standard, my textbooks were not much help.

My special requirement is for a 'catalogue' file where the number of items is unknown on both record and playback. I have achieved this without dummy markers as shown below. Enter a null string to close the file and initiate playback.

All good wishes, and thanks again.

Sincerely,

John

```
PROGRAM FI;

VAR F:STREAM[80];
    TXT:STRING[31];
    CH:STRING[1];

BEGIN
  GRAPHICS;
  OPEN(F,"DSK1.PASFI",OUTPUT);
  REPEAT
    BEGIN
      READLN(TXT);
      IF TXT<>"" THEN
      PUTLN(F,TXT);
    END;
  UNTIL TXT="";
  CLOSE(F);

  OPEN(F,"DSK1.PASFI",INPUT);
  REPEAT
    BEGIN
      GET(F,CH);
      IF NOT EOF(F) THEN
      WRITE(CH);
      IF EOLN(F) THEN
      WRITELN
    END;
  UNTIL EOF(F);
  CLOSE(F)
END.
```

Notes on Turbo-Pasc'99
Reference Manual:

The manual was black plastic spine bound
Front and back were protected with a thick
highly static clear plastic sheet

Many pages were blank and NOT included
in the numbering. These blank pages
were all on the LEFT (<--) when the book
was open.

The totally blank and un-numbered pages
have not been scanned.
Looking at the numbers on the bottom of the
pages, the following pages are blank on
the back:
11, 40, 43, 44, 45, 52

Private correspondence relating to
Turbo-Pasc'99 is included as an
appendix and contains help with
using data files.