TURBO-PASC'99

(c) 1986 by WiPoSoFt

Translation by: wolhess (atariage),
Additional Editing by: daneicher99@hotmail.com

Table of Content

Content

Additional Pascal Fun Facts
First Look TURBO-PASC'99 by Stephen Shaw

Section 1

Preparations

*** Introduction**
*** We Begin**
*** The Command Interpreter**

1. Introduction

1.1. Overview on WiPoSoFt TURBO-PASC'99

WiPoSoFt TURBO-PASC'99 is a compiler package that is fundamentally different from "conventional" systems.

The path from the source code to the building of a program was until now usually long and often led to suboptimal outcome, namely the code execution speed left much to be desired. TURBO-PASC'99 has features usually found only on expensive personal computers:

* TURBO-PASC'99 is an integrated system that has both an Editor, as well as a compiler in a unified program.

* The edited source program is also resident in Memory, thereby eliminating the slow disk operations.

* Syntax error in the editor are pointed to by the cursor, thus immediate correction without any waiting time is possible.

* A Linker package is also included. Programs can be compiled as separate modules before running. In this way major libraries can be created. The linker generates from the program modules on request a Memory Image File which can then be executed using the RUN PROGRAM FILE option from the Editor / Assembler.

* The generated code is very fast, as numerous optimizations are implemented during compilation.

1.2. Use of this manual

That handbook is a gradual introduction to TURBO PASC '99 and its software environment so it should be worked through from front to back. Some chapters have practice examples, which explain the software components. Especially important is Chapter 5.3 (modularization of programs). Only when the Module concept is fully understood are the powerful capabilities of TURBO-PASC '99 fully exploited.

2. We Begin

2.1. Files on the disk
The following files are available:

* TP99 - The compiler system TURBO-PASC'99. It is loaded with the RUN PROCRAM FILE option of the editor / assembler.

* LK99 - The linker. LK99 is also launched with the RUN PROGRAM FILE option.

* RUNLIB - The Runtime system. Contains all the standard routines and the necessary initialization code for a compiled program.

* RUNLIBEQ - Runtime System Equates. Needed for assembly and compilation.

* SIEVE1 - Sieve of Eratosthenes. This is a well-known routine used as a Benchmark Test to calculate Primes. This Program was converted by the linker into an executable-Image format and can therefore be run with the RUN PROGRAM FILE option.

* WORM - A simple game for two players. This program has not yet been processed by the Linker and therefore cannot be launched as is. It aims to explain the Linker function and is found in

* VDPLIB - A small library with display routines. It is required by the program WORM.

2.2. Preparations for the operation with a single drive

If you only have a floppy drive, you will need two blank, formatted floppy disks (for more detailed information on formatting and copying, please read the DiskManager manual):

1) Copy the Assembler files (ASSM1, ASSM2) to the first empty floppy.
2) Then copy from the TURBO-PASC'99 disk the following file:
   - RUNLIBHQ

This disk you need for your programming.
Please label it as ***PROGRAM DEVELOPMENT***.

3) Copy from the TURBO-PASC'99 disk to the second blank floppy disk the following files:
   - LK99
   - RUNLIB

This disk is required to link and start your Object module.
Please label it as ***PROGRAM EXECUTION***.

2.3. Preparations for the operation with two or more drives

You need at least a blank disk, which must be prepared as follows (for more detailed information on formatting and copying, please refer to your manual for DiskManager):

1) Format the blank floppy disk.

2) Copy now from the TURBO-PASC'99 disc the following files to:
   -RUNLIBEQ
   -LK99
   -RUNLIB

With this disk, you can create and launch programs.
Please label it as ***PROGRAM DEVELOPMENT / PROGRAM EXECUTION***.

2.4. Starting the compiler system

If you now have the disk (s) prepared as described above, you can run TURBO PASC'99 . Make sure that the editor / assembler module is inserted.

1) Insert the TURBO-PASC'99 disk into drive 1

2) Now select in the Editor / Assembler menu "RUN PROGRAM FILE"

3) Start the program "TP99"

After a short delay the TURBO-PASC '99-logo appears on the screen, and you are in the command interpreter. Remove the TURBO-PASC'99 disc from the drive and insert the prepared disk PROGRAM DEVELOPMENT in Drive 1.

** Running TURBO-PASC ’99 with Classic 99 Emulator **

Download Classic99 Emulator from:
http://www.harmlesslion.com/software/Classic99/

Using the classic99 version 399.0X.

DSK1. is mapped to the disk image TP99V2.dsk
DSK2. is mapped to the FIAD folder DSK. in the classic99 directory
DSK3. is mapped to the disk image TPDEMO1.dsk

The disk image is called: TP99V2.dsk; from this I need the program LK99 to start before I can run the demo programs from the disk image: TPDEMO1.dsk.

The programs are the files: COUNT1, FAK1, LISTER1, MARGIN1 and SIEVE1. You can run the programs as described after running LK99 (you can press F9 after loading to get back to the EA) and a warm reset.

To run the programs after a cold reset, classic99 shows only a green screen.
On the real TI I see the green screen too, but also the drive DSK1. works (to access the RUNLIB file, I think) and then the programs are running.

***** TP99V3 *****

For a step by step tutorial of editing, compiling and running a program, with TP99V3 – please see TurboPasc99_first steps.docx

3. The Command Interpreter

The compiler is a command-oriented system. All functions are supported by a command shortcut to which - depending on the command - up to a parameter may be added.

On the left bottom of the screen you should see the cursor. If this is not the case, then your TV or monitor left screen column is not visible. Press the spacebar a couple of times and the cursor should appear.

3.1. Command Overview

The following functions can be executed with the command interpreter (in alphabetical order):

- CO: Compile: launches the compiler without any code generation. The source program is syntactically checked.

- CO <filename>: Compile to file - launches the compiler with code generation. The source program is syntactically checked, then converted to machine language converted and saved to file <filename>.

- D!: <filename>: Delete File (DV80) - The <filename> File is deleted from the disk.

- ED: Enable the full-screen editor.

- FI / <text string>: Find String - text search for the editor. Allows you to search for strings.

-GO Line Number Invokes editor and takes you to a specific line.

- LO <filename>: Load file - load a file (DV80) from disk. That format is compatible with the Editor / Assembler, so the E/A-Editor created files can be read.

- PU: Purge Workfile - Erase the memory. The working memory is released, with approximately 14 KB made available.

- Q!: Quit - Exit the system compiler.

- SA <filename>: Save to file - saves the working memory to a file. The main memory (usually

a turbo-PASC'99 source program in DV80 format) is stored to disk. These files can be used with the E/A-Editor, since they are compatible.

- SI: Size - Returns the size of free memory. This function displays the free memory up to 13492 Bytes.

Section 2

Program Development

*** The Editor**
*** The Compiler**

4. The Editor

TURBO-PASC '99 also includes a powerful editor with many functions. The main functions of the known E/A-Editor have been implemented in addition to interesting and useful extended properties which make programming in TURBO PASC'99 especially easy.

4.1. Features

The provided Editor functions can be divided into 4 groups:

- Cursor functions: Function keys that move the cursor
- Line editing functions: Allow single line editing
- Text editing functions: Allow block-wise text editing
- Other features

4.1.1 Cursor Functions

Cursor functions are used to reposition the cursor in the text, without changing it in any way. These include:

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

<FCTN> S (cursor left): Moves the cursor on the same line one position to the left.
Cursor cannot be moved beyond the left text margin.

<FCTN> D (cursor right): Moves the cursor in the same line one position to the right.
Cursor cannot be moved beyond the right margin.

<FCTN> E (cursor up): Moves the cursor in the same column and line up.
Cursor cannot move beyond the first text line.

<FCTN> X (cursor down): Moves the cursor in the same column one row down.

<FCTN> 4 (roll up): Moves the cursor one screen down, i.e. the text scrolls up.

<FCTN> 5 (next window): Moves the cursor one screen to the right, i.e. the text scrolls to the left. A page of text is divided into three overlapping text sections (Windows), with the currently active window being indicated on the bottom right of the screen (W1 = Window 1, etc.).

<FCTN> 6 (roll Down): Moves the cursor one screen up, i.e. the page of text is scrolled down.

<CTRL> R (home): Moves the cursor to the first column of the line in which it is located.

<CTRL> T (top): Moves the cursor to the first column of the first line in the text.

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

4.1.2. Line Editing Functions

Line editing functions are used to change a line of text. These include:

<FCTN> 1 (delete character): The character under the cursor is deleted, and all subsequent characters are shifted one to the left.

<FCTN> 2 (insert character): When this function is activated, characters are inserted at the location of the cursor and remains active until another editor function is selected.

<FCTN> 3 (delete Line): The line where the cursor is located is deleted from memory, and all subsequent lines are scrolled one line up.

<FCTN> 8 (insert Line): A blank line is inserted at the current cursor location. All following lines (including where the one where the cursor was located before the insertion) are scrolled down.

4.1.3. Text Editing Functions

Text Editing functions are used on text blocks, with a block of text consisting of several lines. These include:

<CTRL> 1 (set first of marker): This key will mark the beginning a block of text to be highlighted. The symbol "M1" (Marker 1) will be displayed on the bottom left of the screen.

<CTRL> 2 (set second marker): This will mark the end of a text block. "M2" (Marker 2) will be displayed on the bottom of the screen. After one or both markers are set, no text-editing functions are allowed, only cursor functions. If text editing is attempted, then the markers will be deleted and will need to be reset. Please note that you cannot set Marker 2 before Marker 1.

After the block markers are set, in addition to the aforementioned cursor movements, the following functions can be used:

<CTRL> D (block Delete): Deletes the selected lines.

<CTRL> M (move Block): Moves the selected block to the current location of the cursor which must be located outside of the marked area.

<CTRL> C (copy block): Copies the selected text to the current cursor location.
Copying within a block is not allowed.

4.1.4. Other Features

Two additional features are available:

<FCTN> 9 (back): Exit the editor and return to the Command Interpreter.

<FCTN> = (Quit): Keyboard input is stored in a keyboard buffer, which is subsequently processed by the editor. <FCTN> = will clear the keyboard buffer and prevent further processing, thereby avoiding any changes made. This is useful for example if <FCTN 3> (delete line) is accidentally pressed and you want to restore the deleted line.

4.1.5. Extras

To increase the ease of structured programming, two interesting extensions have been implemented:

AutoIndent: Automatic cursor for faster more efficient programming. After pressing <ENTER>, the cursor is located on a new line at the same column location as the previous one. If, however that line is not empty, then the cursor is located at the beginning of the text on the line.

AutoKey: Automatic keyword recognition for TURBO-PASC'99. Keywords are automatically recognized and converted to upper case so as to improve the readability of the program. It is therefore recommended to release the <ALPHA-LOCK> key when editing a program in order to take full advantage of that feature.

4.2. Saving The Memory To A File

By means of the command SA <filename>, the contents of memory are stored on any peripheral device except cassette (disk, printer, etc). All you have to do is leave the editor with <FCTN> 9 (back) and enter the following command:

```
SA DSK1.MYPROGRAM
```

All memory contents are stored in DV80 format on floppy disk. The format is compatible with that of the Editor / Assembler i.e. files can be exchanged between the two program packages.

The following commands will print the contents of memory:

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

4.3. Loading A File

A file can be loaded into memory via the command LO <filename> from any peripheral device (except cassette), typically from disk. You must exit the Editor with <FCTN> 9 then enter for example the following command:

```
LO DSK1.MYPROGRAM
```

A warning message will come up if you have unsaved text in memory and you will have the option of cancelling the loading operation. Before loading the screen goes blank, and if loading was successful, then the first page of text will appear on screen. In case of an error, a message will appear on screen and memory contents will be retained. In addition to these File / Device Errors, the following errors can also occur:

- The file to load does not fit into memory (Memory full).
- Non-ASCII characters removed.

In these cases, the loading process is not interrupted, and the file can still be edited.

4.4. Deleting A File

To delete a file from a peripheral (generally a disk drive), exit the Editor with <FCTN> 9 then enter the following command for example:

```
D! DSK1.MYPROGRAM
```

Please note that files are deleted permanently although it may be possible to recover them with a sector editor or other file recovery utility. Only DV80 files can be deleted in order to prevent the accidental deletion of system files.

4.5. Text Search function

To search the memory for a particular string, you must exit the editor with <FCTN> 9 then enter the following command for example:

```
FI / PROCEDURE Output
```

The search starts from the point at which the cursor has been before leaving the editor. If the string is found, the system will return to the editor and position the cursor on the first letter of the string. If a further search is required, the editor must be exited again. Now there are two possibilities:

- Press <FCTN> 8 then <ENTER> - you will return to the command line and activate a renewed search.
- Press FI <ENTER> - the last specified string is searched.

4.6. Display Of Free Memory

To identify how much memory is still available for editing, you have to leave the editor and enter the following command:

```
SI
```

The amount of free RAM in bytes will be displayed. Pressing any key will enter the Editor. The amount of free memory is not synonymous with the number of characters that you can type. TURBO-PASC'99 keywords need only one byte of RAM regardless of their actual length. Text trailing spaces are completely ignored, however blank lines require two bytes of RAM.

4.7. Clearing The Memory

To clear the entire memory, you must leave the editor and enter the following command:

```
PU
```

If there is unsaved text in memory, you will get a warning message. You may type Y (for IGNORE), otherwise type N or any other key to cancel the command. After the memory is cleared, you will have 13492 bytes available.

5. The Compiler

This chapter is the most complex. It is therefore important that you work through this section very carefully. In order to present something as complex as the so-called "grammar" of a programming language, we unfortunately cannot avoid formal definitions, which we have tried to present in a readable form without giving up clarity. In order to understand the importance of formal definitions, an example is given:

Chapter 5.1.1 - Use Of Integers

We will explain how to represent integers in your program text so that the compiler recognizes them as such. The definition is as follows:

<integer constant>

: <digit> ...

: <sign> <digit> ...

<digit>

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

<sign>
:+: -

The above exemplifies the formal definition of integer constants. Below is the description of the definition icons used:

< > - replace the description in between with the appropriate text

: - The text to the right of this symbol until the next ':' or end of the text is an alternative option. When multiple ':' are present, you can choose one of the alternatives shown.

... - The parameter to the left of '...' can be repeated multiple times.

Specifically, in the example of the integer constant, the interpretation would be:

<integer constant> has two alternatives:

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

Valid integer constants according to the first alternative would therefore for example be:
1, 9548, 7, 872, 490, 370, 170, 242.

To resolve the second option, you must first define <sign> then <digit>.
Valid integer constants accordingly would therefore for example be:
-0, +8375, -689010641.

In addition to the formal definitions, there are specified conditions that cannot be represented syntactically, so-called semantic conditions. In the example of the integer constant, that would be the restriction that integers not exceed a certain size. While it would be semantically correct for example to enter a 20-digit number, however only numbers between -32768 and 32767 are allowed.

5.1.2. Construct of Real Constants

The fundamental difference between integer and real constants is in the much wider range of values in the latter which cannot be processed directly by the CPU. The permitted value range is:

-9.99999999999998127..-1E-128
0
lE-128..9.9999999999999E127

Real numbers must either contain a decimal point, contain an 'E', or both, so that the compiler can distinguish between integer and real constants. Depending on the number of significant digits given, a Real type of length 4, 6 or 8 is allocated (see also Section 5.2.7. - Variables agreement).

Real numbers can be represented in two ways:

Decimal representation - the exponent is omitted. This representation is suitable for smaller

numbers in terms of value. Example: 600, 3001

Exponent representation - The size of the number is determined by the value of the exponent. This representation can be confusing but is necessary to express the value of large numbers. Example: 6.02E23

5.1.2.1. Decimal Representation

The lexical structure of the decimal expansion is defined as follows:

<real constant decimal>

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

5.1.2.2. Exponent Representation

<real constant exponent>

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

5.1.3 Making string constants

String constants must be enclosed in quotes (“) and can contain several characters. If a quote is to be inside the string constant, it must be represented with a double quote. A string constant must not cross the line end. To make longer string constants, the concatenation operator & can be used (See chapter 5.2.9.10.2 – String expressions) Empty stings can of course also be specified as (“”).

5.1.4 Names

Names are contrary to keywords (see appendix A) names which the user specifies and uses to name his objects.
Names are built as follows:

<Name>
: <Letter>
: <Letter> <All sorts of characters>
<All sorts of characters>
: <Letter>
:<Digit>
: _ (underscore character)

No difference is made for upper or lower case letters. Even if names can be up to 80 characters long (max line length), it’s wise to limit oneself to about 10 characters, since all characters are significant and the memory of the TI-99/4A is limited.

5.1.5. Special and compound characters

With compound characters we understand a chain of two special characters, e.g. '<='. For the compiler to recognize these, they must have no space in between (e.g. '< =' is not allowed). In appendix A all special and compounded characters are listed.

5.1.6. Comments

Comments consist of any text, that enclosed in braces. Comments may be nested, up to a nesting depth of 65535 levels.

Example:
{ This is a {nested} comment}

Inside comments compiler options can be specified (e.g. {I+}). The available compiler options are explained in appendix C.

5.2. Language elements of the TURBO-PASC'99

5.2.1. Program structure

A program is defined as:

<Program> : <Program header> <Block>.

5.2.2. The program header

The program header is declared as:

<Program head> :PROGRAM program_name;

The program name is a name which represents the program, but has no further use, except for documentation purpose.

5.2.3. The block

The block is defined as:

<Block> : <Declaration part> <Statement part>

5.2.4. The declaration part

The declaration part consists of the following objects:

<Declaration part> :
<Label declaration>
<Constant declaration>
<Variable declaration>
<Procedure and function declarations>

It must be noticed, that the objects must be declared in this order. A name can't be used until it has been defined in the source code text.

An exception from this rule are those names, that has a predefined meaning; these are declared as standard names and are listed in appendix A. These names can be redefined, but the programmer should not use that ability, in order to maintain readability of the program.

There is a list of words, which must not be used as names. These words are called keywords. They are explained in detail in appendix A.

5.2.5. Label declaration

You can either omit this part or declare labels as follows:

<Label declaration>
: LABEL label;
:LABEL <labels>… label;
<Labels> : label,

The label is a name.

It should be noted that a declared label must be used in the statement part of the block.

5.2.6. Constant declaration

You may omit or declare constants as follows:

<Constant declaration>
: CONST <Constant assignent>
<Constant assignment>
: Constant name = constant value:

The constant name is a name.

Constant names can be used instead of simple constants. Simple constants are for example 1.3434, “hello” and so on.

There are a number of standard constants available, like PI, E etc. They are listed an appendix A. They can be redefined in the constant declaration part, but to maintain program readability, that possibility should be avoided.

5.2.7. Variable declarations

You can either omit or declare variables as follows:

<Variable declaration>
: VAR <Variable definition>
<Variable definition>
: Variable name : <Type declaration>
: <Variable names> … Variable name: <Type declaration>
<Variable names>: Variable name ,

The variable name is a name.

<Type declaration>
: <Basic type>
: ARRAY [<Index specification>] of <Basic type>
: STREAM [Integer constant]
: BLOCK [Integer constant]
: RELATIVE [Integer constant]
<Index specification>
: Integer constant
: <Integer constants> … Integer constant
<Integer constants>: Integer constant ,
<Basic type> : INTEGER
: REAL [Integer constant]
: BOOLEAN
: STRING [Integer constant]

These integer constants must be positive.

For the ARRAY declaration the integer constants determines the upper index. It should be noted that the lower index is always 0.

For the FILE declarations STREAM, BLOCK and RELATIVE the integer constant represents the length for each data record. It must be no higher than 255. The meaning of these declarations is explained in chapter 5.5.3. – File I/O. A file variable requires 2 bytes of main memory.

The INTEGER declaration uses 2 bytes of main memory. This data type can be operated upon directly by the CPU.

For the REAL data type, the constant must have one of the values 4, 6 or 8. This determines the size of the REAL variable. Each size requires either 4, 6 or 8 bytes in main memory, respectively. The smaller the value the faster the software execution will be, but since there are less significant digits the accuracy will also be lower.

For the BOOLEAN declaration one byte of main memory will be reserved. Just like with integers, the CPU can use this data type directly.

For the STRING declaration the integer constant may be up to 255. This constant determines the string length and thus also the main memory demand. In addition one length byte will be allocated, to represent

the current length of the string.

5.2.8. Procedure and function declarations

You can either omit this section or declare them as follows:
<Procedure- or function declaration>
: PROCEDURE Procedure name <Formal parameter> ; <Body>
: FUNCTION Function name <Formal parameter> : <Basic type> ; <Body>;
The procedure and function names are names.

The formal parameter list can be omitted or declared as follows:
<Formal parameter>
: (<Formal>)
: (<Formals> … <Formal>)
<Formals>
: <Formal> ;
<Formal>
: VAR <Formal Parameter Spec>
: <Formal Parameter Spec>
<FormalParameterSpec>
: Parametername : <Type definition>
: <Parameter list> … <Parameter name> : <Type definition>
<Parameter list>
: Parameter name ,

The parameter name is a name. Parameters can be used like variables.

The keyword VAR in the formal parameter list indicates that the following parameters are used as reference parameters, i.e. a pointer to the real variable is transferred, not the value of the parameter. This implies that any change in the formal parameter in the procedure or function (e.g. with an assignment) will also change the value of the corresponding variable in the calling program. This kind of parameter is used to return results.

When the keyword VAR is omitted, the value of the current parameter is transferred. If the formal variable is changed, this will be local to the called procedure. The variable in the calling program will remain the same.

The procedure or function body can have the following alternatives:
<Body>
: <Block>
: EXTERNAL
: FORWARD

The keyword EXTERNAL tells the compiler that the corresponding procedure or function is located in a MODULE. That's an independent program file, where procedures and functions are stored. It's similar to a library. This is explained in detail in chapter 5.3 – Modularizing of programs.

When the keyword FORWARD is given, the compiler is told that the function or procedure is declared in the same logical block, but further down the source code text. In the next declaration, the list of formal parameters must not be repeated. The FORWARD declaration is necessary to use to facilitate indirect recursion (two procedures both call each other).

5.2.9. The statement part

The statement part is defined as follows:

<Statement part>
: BEGIN <Statements> … END
<Statements>
: <Statement>;
<Statement>
: Label : <statement>
: <statement>

A label must be declared in the same block as it's used (see chapter 5.2.5 – Label declarations)
The statement can be omitted or defined as follows:

<statement>
: <Assignment>
: <Procedure call>
: <goto command>
: <statement block>
: <if statement>
: <repeat statement>
: <while statement>
: <for statement>
: <case statement>

5.2.9.1. The assignment

<Assignment>
: <Variable> := <expression>
: Function name := <expression>
<Variable>
: Variable name
: Arrayvariable name [<Index expression>]
<Index expression>
: <Expression>
: <Expressions> … <Expression>
<Expressions>
: <Expression> ,

The index expression must be of type integer.

The following assignments are allowed:

- integer from integer
- boolean from boolean
- real[n] from real[m] n,m in [4, 6, 8]
- real[n] from integer n in [4, 6, 8]
- integer from real[n] n in [4, 6, 8]
- string[n] from string[m] 0 <= n,m <= 255

Assignments must use basic types.

In addition, it must be noted that with the types string and real, a length adjustment occurs. That is, if a longer type is assigned to a shorter one, the longer value will be truncated.

When a real is assigned to an integer, a run time error may occur (see appendix B), if the value of the real is so large, that it can't be converted to an integer.

Assignments to function names can only be done inside the function block definition. It will thereby determine the return value of the function.

5.2.9.2. The procedure call

<Procedure call>
: Procedure name (<Actual parameter list>)

The actual parameter list must have the same number of parameters as the corresponding formal parameter list in the declaration of the procedure in question.

<Actual parameter list>
: <Actual parameter>
: <Actual parameters> … <Actual parameter>
<Actual parameters>
: <Actual parameter> ,
<Actual parameter>
: <Expression>
: <Variable>

When a formal parameter is declared as a reference parameter (using the keyword VAR), then the corresponding actual parameter must be a variable, not an arithmetic expression.

Basically, the types of the formal and actual parameters must match. However, a length adaption occurs with the basic types string and real, if required, but only for value parameters. For reference parameters and arrays the type must match exactly (also the lengths).

When an array is passed as a parameter will only the variable name for the array (without indexes) be passed as the actual parameter. It's recommended to pass arrays as reference parameters, to avoid a large allocation of runtime memory and execution time.

File variables are always passed by referencing the file name.

5.2.9.3. The goto statement

<goto statement>
: GOTO label

The label must be in the same or a superior block.

TURBO-PASC'99 supports structured language elements like WHILE, CASE, REPEAT and so on, so that the goto statement in most cases is superfluous. Goto statements usually deteriorates in many cases the ability to understand the program's structure. I don't think I've ever used goto in a Pascal program (translator's comment).

5.2.9.4. The statement parenthesis

<statement parentheses>
: BEGIN <Statements> … END

Using this command several statements can be joined as a unit (comparable to using parentheses).

5.2.9.5. The if statement

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

This expression must be of the type Boolean. It should be noted that before ELSE there should be no ;. In addition, the IF statement is considered closed where the THEN statement ends, so from possible syntax error's point of view, what comes after ELSE is an independent statement.

5.2.9.6. The repeat statement

<repeat statement>
: REPEAT <statements> … UNTIL <expression>

The expression must be of type Boolean. The repeat loop is executed until the Boolean expression becomes TRUE. The REPEAT loop is always executed at least once.

5.2.9.7. The while statement

<while statement>
: WHILE <expression> DO <statement>

The expression must be of Boolean type. The while loop is executed until the expression becomes FALSE.

If the expression evaluates to FALSE on the first encounter, the while loop is not executed at all (contrary to the REPEAT loop).

5.2.9.8. The for loop

<for loop>
: FOR variable := <expression> TO <expression> DO <statement>
: FOR variable := <expression> DOWNTO <expression> DO <statement>

The loop variable and the expressions must be of type integer.

The keyword TO means that the loop runs in ascending order and DOWNTO that it runs in descending order (always counting by one).

To avoid logical program errors it's enforced that the loop variable must be local (see chapter 5.4 – Scope of names) and a simple integer variable.

5.2.9.9. The case statement

<case statement>
: CASE <expression> OF <case body> END

The expression must be of type integer.

<case-body>
: <case-alternative>
: <case-alternative> …

<case-alternative>
<case-label list : <assignment>

<case labellist>
: <case label>
. <case labels> … <case label>

<case-labels>
: <case-label> ,

<case-label>
: integer constant
: <range>

<range> : integer constant .. integer constant

Case alternatives are processed sequentially. An alternative is true as soon as the value of the expression equals one of the constants or fits inside the range. If this is the case for one alternative, the corresponding statement will be executed and the case statement is then left.

If none of the alternatives match, then a run time error will occur. If you need a fall-through alternative, in case none of the listed alternatives match, the following trick can be deployed:

Make the last alternative as:

Minint..maxint : <statement>

Minint and maxint are standard constants, representing the smallest and largest integer the computer can handle (see appendix A).

5.2.9.10. Expressions

Expressions consist of operands (constants, variables etc), operators (+, -, *, / and so on) and rounded parenthesis. A type is always associated with an expression. Operators are rated in the hierarchical levels 0, 1, 2 and 3.

Evaluation of an expression

- Operators on a lower level are evaluated before those on a higher level.
- Operators are evaluated from left to right.
- Expressions inside parentheses are evaluated first.

Level 3:

<expression>
: <simple expression> <relation> <simple expression>

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

Level 2 :

<simple expression>
: <leader> <term>
: <leader> <term> <addition> ...
: <term>
: <term> <addition> ...

<addition>
: <addition operator> <term>

<leader> : + : -

<addition operator> : + : - : OR : &

Level 1:

<term>
: <factor>
: <factor> <multiplication> …

<multiplication>
: <multiplication operator> <factor>
<multiplication operator> : * : / : DIV : MOD : AND

Level 0:

<factor>
: <variable>
: <constant>
: (<expression>)
: NOT <factor>
: function name (<actual parameter list>)

For a function call the same rules for parameter transfer as for procedure calls are valid (see chapter 5.2.9.2 – procedure calls).

5.2.9.10.1. Arithmetic expressions

Arithmetic expressions have a value of the basic type integer or real.

The operators must be type compatible with the operands they apply to.

The following combination rules apply:

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 base type string.

The following combination rules apply:

Level 2:

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

The operator '&' performs concatenation of strings.

5.2.9.10.3. Logical expressions

Logical expressions have a value of type Boolean, either TRUE or FALSE.

The following combination rules apply:

Level 3:

Integer	=,<,<=,>=,<>,>	integer	->	Boolean
real[n]	=,<,<=,>=,<>,>	integer	->	Boolean
real[n]	=,<,<=,>=,<>,>	real[m]	->	Boolean
string[n]	=,<,<=,>=,<>,>	string[m]	->	Boolean

Level 2:

Boolean	OR	Boolean	->	Boolean

Level 1:

boolean	AND	Boolean	->	Boolean

Level 0:

---	NOT	Boolean	->	Boolean

5.3. Modularizing the programs

In this chapter we'll explain how one program can be split in to modules, or program building blocks, which can be stored in different data files and be compiled separately. The possible communication paths between the modules will also be explained.

There is a distinction between main module and library module.

5.3.1. Main module

How to make a main module you can see in chapter 5.2 – Language elements of TURBO PASC'99. Such a module must start with the keyword PROGRAM.

Every program must contain this module, since the execution always starts and ends there.

5.3.2. Library module

The essentials of such a module are:

- A collection of procedures and functions with a common denominator, e.g.graphics utilities, file processing utilities and so on.
- The module has no executable part.

The syntax of a module:

<module>
: <module header> <module definition>

<module header> : MODULE module name;

The module name is a name and has only documentational purpose.

<module definition> :
<constant declaration>
<variable declaration>
<procedure and function declaration> …

These objects are declared exactly as in chapter 5.2 – Language elements of TURBO PASC'99.

NOTE! The variable declaration has a different meaning here: it serves as a communication path with other modules through the concept of a COMMON domain (see chapter 5.3.4 – Communication through COMMON domain)

5.3.2.1. Export of procedures and functions

The procedures and functions in the module declaration are exported either to another library module or to a main module (thereby made accessible by other modules). The correspondence is established with the keyword EXTERNAL.

The following must be noted:

- When exporting as well as importing names only the first 6 characters – the character '_' is excluded – are considered.
- The procedure and function names that are exported must be unambiguous.
- If a name is imported into the main module or into a library module with EXTERNAL, then it must also be exported by a library module.
- If a procedure or function is declared EXTERNAL in a library module, it will be imported but not exported.

If these precautions aren't considered, then linking of modules (see chapter 6 – the linker) may result in errors.

5.3.3. Communication with parameter transfers

The parameter transfer works as is described in chapter 5.2.9.2 – the procedure call.

However, no interface verification is performed, thus the programmer must himself make sure that the formal parameter list in the module matches the actual parameter list.

5.3.4. Communication through the COMMON domain

The variable declarations in the modules declaration part are located in the same main memory address as the variable declaration in the main program module. If for example an integer variable is declared in both modules, this variable can be assigned a value in the main program and then this value can be used in the library module.

This type of communication is very fast, since no parameter needs to be transferred, but direct access to the memory cell is performed. This however requires considerable caution. In no case can more variables be overlaid on the main module than are declared in the main module, or the program will soon come to a crash.

The safest method is to declare the same variables which are declared in the declaration level of the main module. Other names could of course be used. It's important that the order of the type declarations match.

The following example should make the concept of modules more obvious:

Communication with parameter transfers:

Main module:

```
PROGRAM main;
      var result: INTEGER;

      {procedures and functions that are imported from a library module}
      PROCEDURE add (a,b: INTEGER; VAR sum: INGEGER); 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.
```

Library module:

```
MODULE maths;
      { procedures and functions that are exported to another module }
      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 using the COMMON domain

Main module:

```
PROGRAM main;
      VAR n: integer;
            field: ARRAY [100] of STRING[10];
            i: integer;

      PROCEDURE comout; EXTERNAL;
      BEGIN
        readln(n);
        for i := 0 to n do readln(field[i]);
        comout;
      END.
```

Library module:

```
MODULE out;
      { common domain: The common variables must be declared
                       in the same order as in the main module.
                       The names themselves aren't required to
                       be the same.}

      VAR k: INTEGER; { will be overlaid with variable n in the
                         main module}
          f: ARRAY[100] of STRING[10]; { f will overlay field
                                         in the main module }

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

5.4. Scope of names

1) Names could either be local or global.
   To be able to defined the scope of names in an unambiguous way, certain rules must be defined:
   - A name is only valid in the procedure (function) where it's declared (local to the procedure), as well as in all procedures and functions declared therein (global for these procedures). In superior procedures/functions and in the main program it's not defined.
   - Names declared in the main program are accessible in all procedures and functions.
   - Reserved names (see appendix A) are "declared" one level above the main program and are thus accessible both in the main program and in all procedures and functions.

2) When a name is declared again in a lower level, the original meaning is lost on that and lower levels. For the new name these two rules are valid again.

Example:

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

        procedure Level2;
          VAR i,j,l: Boolean;

          BEGIN { Level2 }
            { valid are i,j,l: Boolean
                      k: INTEGER }
          END;

      BEGIN { level1 }
```

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

TURBO-PASC'99 contains a so called dynamic memory management model. Thus by applying a clever structure to the program you can save a substantial amount of memory
.

Since the validity of the variables are void anyway, when the procedure is left, the memory these variables occupied is released and can be used for other purposes. This also implies that the value of these variables upon a second call of a procedure are undefined, since other procedures that may have run in between can have used the same memory area for other purposes.

If you for some reason want to keep a value in a variable across separate calls to a procedure, then this variable must be declared at a superior procedure/function level or at main program level.

5.5 Standard procedures and functions

This chapter describes all procedures and functions that are hosted by the run time system (in RUNLIB on the TURBO PASC'99 disk), and thus are available for use.

This run time system is in reality a library (see chapter 5.3 – Modularizing programs), but you don't have to link this module specifically to your program. Since every program you create with TURBO PASC'99 uses this run time library, the necessary linking is done automatically by the TURBO PASC'99 compiler.

As you may have noticed already, the RUNLIB is a memory image file. That's a further difference compared to the normal modules, which are DF80 files produced by the assembler. Memory image files have the advantage that they load faster into memory.

The run time system contains the following function groups:

- SCREEN – Special routines for screen management
- CONSOLE-I/O – Procedures for keyboard input and screen output
- FILE-I/O – Procedures and functions for file management
- MATH – Mathematical functions
- STRINGS – Functions for manipulation of character strings
- CONVERSION – Functions for data type conversions
- MISCELLANEOUS – Precisely that

Below are the declarations of the standard procedures, their function, a short explanation and an example listed.

For certain standard functions the declaration is a bit problematic, since it concerns generic procedures, that is procedures that has parameter lists of arbitrary length and can have objects of arbitrary types.

Example:

```
write(3.14);
```
is just as correct as
```
write("Hello ", true, 12, 3.14);
```

For these procedures the parameter list is specifically listed as GENERIC.

5.5.1. SCREEN

SCREEN contains procedures for screen display. As you know, the screen used by BASIC supports 768 characters (24 * 32) but 960 characters (24 * 40) when used by the E/A editor.

The 32 character line screen supports the ability to assign one of 16 colors to character groups of eight characters. However, only a limited amount of information can be displayed.

The 40 character line screen supports only two colors (characters and background), but allows for displaying more information at the same time.

To give you the possibility to use any of the two screen modes, the following procedures are implemented:

5.5.1.1. Graphics

Declaration: PROCEDURE Graphics;

Explanation: Enables the 32 character line screen. If that screen mode is already active when the procedure is called, it's ignored. Otherwise the screen is cleared.

Example: Graphics;

5.5.1.2. Text

Declaration: PROCEDURE Text;

Explanation: Switches to the 40 character line screen. If that screen mode is already active when the procedure is called, it's ignored. Otherwise, the screen is cleared.

Example: Text;

5.5.1.3. Cls

Declaration: PROCEDURE Cls;

Explanation: Clears the screen. The procedure adapts automatically to the presently used screen mode (32 or 40 character lines).

Example: Cls;

5.5.1.4. Screen

Declaration: PROCEDURE Screen(foreground_color, background_color: integer)

Explanation: Sets the general fore- and background colors.

When using the 32 character line mode, all character groups as well as the frame are set to the color combination given. The 40 character line screen gets the fore- and background colors. Both parameters must be within the 1 - 16 range.

Example: Screen(16,5);

5.5.2. CONSOLE-I/O

CONSOLE-I/O gives you the ability to communicate with your program. The adaption to the current screen mode is automatic.

5.5.2.1. Cursor

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

Explanation: Positions the cursor for in- and output operations.
Row must be in the range 1 - 24.
Column must be in the range 1..32 for 32 character line screen mode and within the 1 - 40 range for 40 character line screen mode.

Example: Cursor(12,20);

Translators note:
This is contrary to most Pascal implementations, which instead uses the gotoxy command. TURBO-PASC'99 can be made compatible by adding a gotoxy procedure to your programs.

```
procedure gotoxy(x,y: integer);
begin
  cursor(y+1, x+1);
end;
```

Note that gotoxy has the arguments in the other order and uses the ranges 0..23, 0..31 or 0..39, respectively.

5.5.2.2. Write

Declaration: PROCEDURE Write(GENERIC);

Explanation: Outputs expressions of different types on the screen.
You may add a format specifier to the expression,
with the following meaning:

- integer: field width (1 Format)
- boolean: field width (1 Format)
- string: field width (1 Format)
- real: field width (1 Format)
- real: field width:number of decimals
- (2 Formats)
-

Truncation due to too short field widths does not occur. In these cases the format specification is ignored.
Example:

Without formatting

```
Write(12);
```
Outputs: 12

```
Write(1.23);
```
Outputs: 1.23E+000

```
Write("This is a text");
```
Outputs: This is a text

```
Write(true);
```
Outputs: TRUE

With formatting (somewhat edited in translation, for clarity)

```
Write("X",12:4);
```
Outputs: X 12

```
Write(-1.23:5:2);
```
Outputs: -1.23

```
Write("Expansion":15,"More");
```
Outputs: Expansion More

5.5.2.3. Writeln

Declaration: PROCEDURE Writeln(GENERIC);

Explanation: Same as write, but after output of all parameters, the cursor is positioned at the beginning of the next row on the screen.

Example: writeln(1,2.0:5, "three":20);

5.5.2.4. Read

Declaration: PROCEDURE Read(VAR GENERIC);

Explanation: Assigns various objects values from the keyboard. The actual parameter list must contain reference parameters only.

The lexical validity of integer and real constants (see chapter 5.1 – Lexical structure of the TURBO PASC'99 language) will be validated during entry. In case of an error the entered value will be ignored and the input must be re-done. (Translators remark: This is different from most Pascal implementations, which tend to instead generate a run time error if the input has the wrong syntax.)

Also when doing input a format can be specified; it will specify the length of the entry field. For string constants – which must be entered without quote marks – the format specification has an additional meaning:

For entries without formatting, all trailing spaces are disregarded, but if a format is specified, the entry will be padded with trailing spaces if necessary, to reach the specified string length.

Every input must be terminated by pressing ENTER, since that's the only valid separator character. Thus for example three parameters also requires three presses of ENTER.

Example: Read(Name, Address:20, Age:3);
Input: Anders <ENTER>
Sweden <ENTER>
50 <ENTER>

5.5.2.5. Readln

Declaration: ReadLn(VAR GENERIC);

Explanation: Like Read, but after the input has been completed, the cursor will move to the beginning of the next screen row.

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

5.5.3. File-I/O

File-I/O contains all the routines you need for convenient file processing. Both sequential data (stream, sequential) as well as random access data (relative, direct access) is supported. In chapter 5.2.7 – Variable declaration – you've seen how to declare file variables. They have the following meaning:

STREAM – Sequential files with variable record length.
An integer constant is used to specify the logical record length when the file variable is declared.
To a file of this type can only objects of the type string be written. If you want to store arithmetic expressions in a stream file, then you must use the supplied type conversions (see chapter 5.5.6 – Conversion) to convert them to strings.
Likewise, from a stream file only strings can be read.

BLOCK – Sequential files with fixed record length.
The logical record length is defined by an integer constant in the file variable declaration. Objects of all basic types can be written to this kind of file. The value of the object will be represented by its internal representation, thus it occupies the space that was defined in the declaration (see chapter 5.2.7 – Variable declaration). Since the types of the objects are not saved with the object, it's up to the programmer to make sure that when a record in the file is read back in again, type compatibility is maintained. The system will monitor the record length, so in case too little or too much information is read from a record, then a run time error will occur (see appendix A).

RELATIVE – Direct access files with fixed record length.

The same applies as for the file format BLOCK.

5.5.3.1. Open

Declaration: PROCEDURE Open (VAR filevar: filetype;
filename: string[n];
openmode: integer);

Explanation: Assigns a physical file to a logical file definition of suitable type and opens the file.

Filevar is the logical file name and must be of type STREAM, BLOCK or RELATIVE (see chapter 5.2.7 – Variable declarations).

Filename is the physical file name and defines the name of the peripheral unit as well as a file located on that device.

Openmode defines the direction of data transfer (INPUT, OUTPUT, APPEND). These names are available as standard constants (see appendix A).

Example:
```
VAR file: STREAM[80];
…
Open (file, “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 and RELATIVE

Explanation: Output of expressions to a file.

All parameters will be written after each other as logical items to the file. For the file type STREAM, an automatic line feed occurs if the logical item exceeds the logical record length. For files of types BLOCK and RELATIVE, a run time error will occur.

The type of the parameter is dependent upon the data type:
- STREAM: only strings allowed.
- BLOCK: all basic data types allowed.
- RELATIVE: all basic data types allowed.

For BLOCK and RELATIVE the parameter list must contain variables only (reference parameters), to insure a better reliability for the file communication.

Example: You have when computing the logical item length in a field reserved the type real[8]. When attempting to write the constant 3.1 to the file, a run time error occurs, since the compiler will assign such a real constant the type real[4].

Example: Put(file, “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 and RELATIVE

Explanation: The same rules apply as for Put.

In addition, the size of the combined parameters (and the parameters in a possible previous Put statement) will be verified against the declared fixed record length. If they don't match, a run time error will be displayed on the screen.

Coincidentally, the next record in the file is prepared for processing.

Example: Putln(file, "This is the end.");

5.5.3.4. Get

Declaration: PROCEDURE Get (VAR filevar: filetype; VAR GENERIC);

Explanation: Reads applicable values from a file.

For the file type STREAM the next data item will automatically be read, when the logical record end is reached, before all parameters can be assigned values. For file types BLOCK and RELATIVE a run time error is issued if reading beyond the record length is attempted.

A run time error also occurs at attempts to read beyond the end of the file. The function EOF (see chapter 5.5.3.6 – EOF) can be used to test if further data access is possible, or the end of the file has been reached.

The parameter types depends on the file type:

- STREAM: only strings are allowed.
- BLOCK: all basic types are allowed.
- RELATIVE: all basic types are allowed.

Example: Get (file, name, address, age);

5.5.3.5. Getln

Declaration: PROCEDURE Getln (VAR filevar: filetype;
VAR GENERIC);

Explanation: The same rules applies as for Get.

In addition, for files of type BLOCK and RELATIVE, a test is carried out to verify correspondence between the given parameters (and possibly parameters given with previous Get statements) and the in the declaration established logical record length. Should such a correspondence not exist, a run time error will be issued and shown on the screen.

In conjunction with this the next record in the file is prepared for processing.

Example: Getln (file, nearend, theend);

5.5.3.6. Seek

Declaration: PROCEDURE Seek (var filevar: tiletype; recordno: INTEGER);

Explanation: Prepares record number recordno for processing.

This procedure can only be applied to files of type RELATIVE.

Example: seek (directacess_file,12);

5.5.3.7. Eof

Declaration: FUNCTION EOF (VAR filevar : filetype)
: BOOLEAN;

Explanation: Returns if the end of a sequential file (types STREAM or BLOCK) has been reached. This function is meaningful for input files only. When Eof returns TRUE, the next read access will result in a run time error.

Example:
```
IF EOF (file) THEN
  WriteLn("All records have been read!");
```

5.5.3.8. Eoln

Declaration:
```
FUNCTION EOLN (VAR filevar: filetype)
                                    : BOOLEAN;
```

Explanation: Returns if the end of a record in a sequential file of type STREAM has been reached. This function is necessary, since such files have variable record lengths. This function is only useful for input files.

Example:
```
IF EOLN(file) THEN
  WriteLn("Record end reached!");
```

5.5.3.9. Close

Declaration:
```
PROCEDURE Close(filevar: filetype);
```

Explanation: Closes a file. If this procedure isn't called, then all open files will be closed when the program terminates its execution.

Example:
```
close(file);
CLOSE(directaccess_file);
```

5.5.4. MATH

MATH contains all the floating point functions, which you've become accustomed to through BASIC. In the following table the legal parameter types and the type of the returned function value after the calculation:

Function	Parameter	Value of function
ABS	INTEGER	INTEGER
ABS	REAL[n]	REAL[n]
ARCTAN	INTEGER	REAL[4]
ARCTAN	REAL[n]	REAL[n]
COS	INTEGER	REAL[4]
COS	REAL[n]	REAL[n]
EXP	INTEGER	REAL[4]
EXP	REAL[n]	REAL[n]
INT	INTEGER	REAL[4]
INT	REAL[n]	REAL[n]
LN	INTEGER	REAL[4]
LN	REAL[n]	REAL[n]
SIN	INTEGER	REAL[4]
SIN	REAL[n]	REAL[n]
SQRT	INTEGER	REAL[4]
SQRT	REAL[n]	REAL[n]
TAN	INTEGER	REAL[4]
TAN	REAL[n]	REAL[n]

The meaning and use of these functions are explained in your BASIC user's manual.

5.5.5. STRINGS

STRINGS contains support for processing character strings.

5.5.5.1. ASC

Declaration:
```
FUNCTION ASC(ch: string[1]): integer;
```

Explanation: This function returns the ASCII code of the character.

Example: WriteLn(ASC(“A”));
The value 65 will be output.

Translator’s remark: Such a function usually returns the code of the first character if the string contains two or more, but I don’t know if this implementation does the same.

5.5.5.2. CHR

Declaration: FUNCTION CHR(charactercode: integer):string[1];

Explanation: This function returns the character corresponding to the character code given.

Example: WriteLn(CHR(65));

The character A will be output.

5.5.5.3. LEN

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

Explanation: This function outputs the current length of the source string.

Example: WriteLn(LEN(“Hallo”));
The value 5 will be output.

5.5.5.4. SEG

Declaration:
```
FUNCTION SEG(source   : STRING[n];
             Startpos : INTEGER;
             Length   : INTEGER)
                      : STRING[n];
```

Explanation: This function outputs a string, which is the segment with given length and start position from the source string. A more detailed description is available in the BASIC user’s manual.

Example: WriteLn(SEG(“Hallo”, 2, 3));
The text all will be output.

5.5.6. CONVERSION

CONVERSION has powerful routines for converting objects of various types to other types. The conversion of real- and integer types to strings can be further controlled by the same kind of formatting commands which are available for the output procedures in CONSOLE-I/O (see chapter 5.5.2).

5.5.6.1. CIR

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

Explanation: Converts an Integer expression into a Real value, four bytes long.

Example: WriteLn(CIR(12));

5.5.6.2. CIS

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

Explanation: Converts an integer expression into a string of length n. The format can be specified as when using write (see chapter 5.5.2.2 – Write).

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

5.5.6.3. CRI

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

Explanation: Converts a real expression of type real[n] into an integer, where rounding occurs when applicable. If the input value is too large to be represented by an integer, a run time error will occur.

Example: WriteLn(CRI(1.2));

5.5.6.4. CRS

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

Explanation: Converts a real expression of the type real[n] into a character string of length m. The format can be specified as when using write (see chapter 5.5.2.2 – Write).

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

5.5.6.5. CSI

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

Explanation: Converts a string expression with length n to an integer number. The string representing the integer must be lexically correct (see chapter 5.1.1 – Structure of integer constants)

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

5.5.6.6. CSR

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

Explanation: Converts a string expression with length n to a real number of type real[m]; The string representing the real number must be lexically correct (see chapter 5.1.2 – Structure of real constants)

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

5.5.7. MISCELLANEOUS

MISCELLANEOUS contains three useful routines:

- One direct keyboard input routine
- Two routines for random number generation

5.5.7.1. KEY

Declaration: PROCEDURE Key (keyboard_number,
VAR keycode,
VAR Status : INTEGER);

Explanation: Reads the keyboard. For a more detailed explanation, read about CALL KEY in your BASIC handbook.

Example:
```
REPEAT Key(0, K, s)UNTIL s>0;
```

5.5.7.2. RANDOMIZE

Declaration: PROCEDURE Randomize;

Explanation: Initializes the random number generator. This procedure should be called at program start, when random numbers are to be used in the program. By using randomize, a new row of pseudo random numbers is generated each time the program is run.

Example: Randomize;

5.5.7.3. RND

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

Explanation: Returns a random number between zero and the integer expression giving the upper limit.

Example: WriteLn(rnd(32767));

Section 3

*Program execution
*The linker
*Starting user programs

6. The linker

TURBO PASC'99 contains a high performance program, consisting of two components:

- Loader for compiled modules
- Program file generator

The first component, a loader for tagged object code, is comparable to the LOAD AND RUN option in Editor/Assembler, but more comfortable and with higher performance.

The second component, the program file generator, works similar to the Save utility on the Editor/Assembler disk.

6.1. Loading the linker

Insert the diskette PROGRAMMAUSFUHRING in drive 1. Select the Editor/Assembler menu option "RUN PROGRAM FILE" and enter the name DSK1.LK99.

The Linker's logotype will show on the screen.

6.2 Loader for tagged object code

At the left edge of the screen you can see the flashing cursor. A complete line-oriented editor is available for entering the module names. The following editing commands are supported:

<ENTER>: The module name is entered and the corresponding module is loaded from the diskette.

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

<FCTN> D (Cursor right): Moves the cursor one position to the right.

<FCTN> 1 (Delete character): Deletes the character under the cursor.

<FCTN> 2 (Insert character): Switches to insert mode. From this moment all characters are inserted. The insert mode is cancelled by pressing another function key.

<FCTN> 5 (BEGIN): Moves the cursor to the beginning of the entry field.

<FCTN> 8 (REDO): All hereto entered module names are ignored. The cursor is repositioned at the first input field.

<FCTN> 9 (BACK): Exits the linker.

During the loading process, two types of errors may occur:

Errors, which require a renewed input of the last module name.

- I/O-Error #0-7 The meaning of these is detailed in the Editor/Assembler handbook.

Errors, which required renewed input of all module names

- Duplicate symbol. A module was loaded more than once or modules contain the same external definitions.

- Memory full. Too many modules have been loaded. The memory is full.

After loading the desired modules in the correct order, an empty input is used to end the linking process.

This may lead to one of the following errors:

- At least one module must be loaded: You made the empty input as the first entry, without any previous valid module name being entered.
- Main module not loaded: The main module wasn't loaded, only library modules. The cursor will be positioned at the first free entry field, so the missing name can be entered.
- Unresolved reference: One or more library modules weren't loaded. The cursor is positioned on the first free input field, so the missing name or names can be entered.

Provided the input eventually is completed correctly, then the second component of the linker is activated.

6.3. Program file generator

At the lower part of the screen the question about if a program file should be created is shown. If you answer Y, then the same line editor as was described above is available to enter the name of the program file.

Notice then that the program files are split in 8K chunks and the last letter in the file name is incremented automatically (see Save utility in the Editor/Assembler handbook). As the last letter in the file name a digit is recommended (e.g. DSK1.MYPROGRAM1).

Following the entry of the name the original relocatable object code will be converted to a memory image file, which loads via the “RUN PROGRAM FILE” option from Editor/Assembler.

The convenience of loading a memory image file isn’t used during the test phase of software development, where for example a run time error (see appendix B) would lead to bug fixing, new compilation and then a new linking of the program.

The program file is created only when the program runs without errors, so the burden of loading of the linker and entry of all names is otherwise not required.

6.4. First educational example

On the TURBO PASC’99 diskette are two files:

- WURM*
- VDPLIB*

As mentioned in chapter two, WURM* is a game (a main module) and VDPLIB* contains screen utilities (a library module).

To create an executable program out of these modules, you need to take the following actions:

1. Load the LK99 as described in chapter 6.1.

2. Put the TURBO PASC’99 disk in an empty drive (if you have only one drive, then remove the disk PROGRAMMAUSFUHRUNG).

3. Enter one of the two module names together with the drive name (e.g. DSK1.WURM* if the TURBO PASC’99 diskette is in drive 1).

4. As you can see, the drive is immediately engaged. When the loading procedure is completed (which it should be), the cursor is positioned on the next input field.

5. Enter the name of the second module (e.g. DSK1.VDPLIB*).

6. Since now all necessary modules are loaded, the empty input can take place. Now – at the third input field – press only <ENTER>.

7. Now enter the program name, e.g. WURM1. Please specify the device name of the drive containing the diskette PROGRAMMAUSFUHRUNG (if you have only one drive, remove the TURBO PASC’99 diskette and insert the diskette PROGRAMMAUSFUHRUNG instead).

8. Select e.g. the name DSK1.WURM1.

9. The program file generator creates the memory image file.

10. At the question if the program should be started, answer N. The cursor will be positioned at the first input field, so you can link new modules. Since your work is now completed, press <FCTN> 9.

11. In Editor/Assembler, select the option “RUN PROGRAM FILE” and enter the name of the memory image file (e.g. DSK1.WURM1).

12. Good luck! (V I E L V E R G N U G E N)

7. Running user's programs

There are two different possibilities (7.1 and 7.2) for running your compiled programs.

7.1. Starting from the linker

To use this method, you must first load the linker (see chapter 6.1), then in connection to this load your compiled modules (see chapter 6.2). To create a program file is voluntary.

When you can confirm, the screen color changes to red. At the bottom row is a question about if you want to run your program. When you press Y, your program is executed.

First make sure that you have the diskette PROGRAMMAUSFUHRUNG in drive 1, since the file RUNLIB is located on that disk. This file is a memory image file, which contains the complete run time system of TURBO-PASC'99 (i.e. all standard functions and procedures, setup routines etc.).

If the file RUNLIB isn't on the diskette in drive 1, an error message is issued:

INSERT "RUNLIB" IN DRIVE 1!

Take corrective action and press any key except <FCTN> 9 (BACK). The program will make a new attempt to load RUNLIB. If you press <FCTN> 9, you'll be brought back to the TI logotype screen.

7.2. Starting with the E/A module

This method is undoubtedly the most convenient of the two possibilities. However, it does require that you after loading the module (see chapter 6.2) have created a program file (see chapter 6.3).

From the E/A menu, select option "RUN PROGRAM FILE". Before entering the file name, make sure that drive 1 contains the diskette PROGRAMMAUSFUHRUNG. This should be the case anyway, since you should save your programs on the PROGRAMMAUSFUHRUNG diskette only. After entering the file name the program starts by itself.

Section 4

***Appendix A**
***Appendix B**
***Appendix C**

Appendix A

A.1 TURBO-PASC'99 keywords.

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	arctan	chr	cir
Cis	Close	cis	cos	crt	crs
Csi	Csr	cursor	e	eof	eoln
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

&	..	%	.	(	)
0	-	*	/	,	:
;	<	=	>	<=	<>
:=					

Appendix B (error codes)

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	Encapsulating more than 9 levels of functions/procedures
52	Identifier declared twice
53	Forward reference not resolved
54	Too little memory available for compiled program
55	Declared label not used
56	Wrong dimension
57	Disallowed real type
58	Disallowed string type
59	Identifier not declared
60	Constant not within allowed range
61	Type conflict
62	Wrong use of identifier
63	Integer expression expected
64	Boolean expression expected
65	String expression expected
66	Integer or real expression expected
67	Simple expression expected
68	Too little memory for the compiler
69	The same label used more than once
70	Assignment of function identifier not within scope of function
71	Loop variable not declared locally
72	Disallowed data type

B.4 Run time errors

0-7	Standard I/O-errors. See Editor/Assembler handbook.
8	File not opened
9	Logical file name assigned to multiple physical files
10	More than 9 files open
11	Logical record length not observed
12	Division by zero
13	Over- or underflow in arithmetic calculation
14	Error in mathematical function
16	Disallowed parameter value for standard procedure or function
17	Invalid format specification (must be between 0 and 255)
18	Field index outside of bounds
19	No valid case option
20	Stack overflow

Appendix C

By using the compile time options, you can influence the result of the code generation. These options are handled using a mechanism that's similar to a switch. In the source code, at selected places these switches can be turned on or off.

As is shown in chapter 5.1.6 – Comments – these switches are stored inside comments. In the case of staggered comments, compiler options are only recognized when they occur at the first level.

Rules for inserting compiler options:

<Option> : $ <letter> <prefix>

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

<prefix> : + : -

If these rules aren't followed, the compiler option will not be recognized as such, but will be treated as a normal comment (i.e. disregarded).

The letter identifies which compiler option is used. The prefixes are used to enable the option '+', or disable it '-'.

The meaning of the compiler options

Option B : (default $B-) -- Boolean

With the option $B+ you can specify that the evaluation of a Boolean expression will end as soon as the result of the expression has been determined.

Example:

```
{ $B+ }

IF (i>10) OR (f[i]<>0) THEN ;
```

When (i>10) is true, the second part of the expression (f[i]<>0) will not be evaluated at all, since it's already known that it will not change the result of the expression.

Using this option when the code contains complex Boolean expressions can result in a substantial generated code. Still this will give a better run time performance, when not all of the Boolean expressions always must be evaluated.

Translators note: Beware of this option in cases where part of the expression involves calls to functions!

Consider an example like this:

```
function f(var x: integer): integer;

begin
  f := x*x*2-80;
  x := x+2;
end;
…

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

Here the second part of the expression involves calling the function f. If the first part of the expression is true, then the second part is not executed. Thus function f is never called. Since function f not only returns a result of the function, but also modifies the value of the actual parameter in the call (variable i in this case), the value of i after the last code line above will not be incremented by two if it's already above 10. Although this may be the desired action, such code is very difficult to understand and debug.

Option E : (default $E-) –Extended Check

With option $E+ will the results of integer addition and subtraction be tested for over- or underflow, respectively. Activating this function generates a substantial amount of extra code. Thus it should only be used when a strict evaluation of the result being within the value range is desired.

Option A : (default $A-) -- Array

By using $A+ array indexes will be checked against their bounds, both upper and lower. When active the code generated will be significantly longer and somewhat slower, but also in a was safer. Turning this option off may in the worst case result in a program crash!

Translators note: This option is more often called $R+ (range check), and is also more often defaulted as being active.

Option S : (default $S-) -- Source

By using $S+ the assembler source code, which is generated by the compiler, will contain the original source code lines as comments. This option will not change neither the code length nor the execution time of the code.

It does produce a longer assembler source file, which in turn implies a longer assembly time.

Option I : (default $I-) --Initalize

By using this option, variables with the corresponding types will be initialized with 0, 0.0, false or "" respectively. This option has no effect on the code length. The execution time will be longer if large amounts of data it initialised – especially when large arrays are used – inside procedures or functions, since the dynamic memory model implies that these variables require initialisation each time the procedure or function is called (see chapter 5.4 – scope of identifiers).

Using TURBO-PASC'99

***Diskette handling with one drive**
***Diskette handling with two drives**
***Sample program SIEVE**
***Sample program RECURSIVE_FUNCTION**
***Sample program FILE_LISTING**
***Sample program WURM**

Diskette handling with one drive

- Here you'll see how to use your prepared disks in daily use. Still, it's just a suggestion, to give you a starting point when you look for the most efficient solution for your need.
- Place the TURBO PASC'99 disk in the drive and start the compiler system.
- Put the disk PROGRAM DEVELOPMENT in the drive. On this disk you save only your Pascal source files and the assembler source files the compiler creates.
- Leave TURBO PASC'99 and assemble the assembly source files, again to the disk PROGRAM DEVELOPMENT.
- Now insert the PROGRAM EXECUTION disk in the drive and start the linker.
- If you have more disks of the type PROGRAM DEVELOPMENT (perhaps you've already been a busy creator of modules), you must for a moment act as a "disk jockey". Only insert the disks that contain the desired library modules.
- If you want to create a program file, or want to run the program, then you must again insert the disk PROGRAM EXECUTION, since that's where the file RUNLIB is stored. RUNLIB is required by all user programs.

Diskette handling with two or more drives

- Insert the TURBO-PASC'99 diskette in drive 1 and start the compiler system.
- Insert the diskette PROGRAM DEVELOPMENT/PROGRAM EXECUTION in drive 1.
- Put another diskette in drive 2. On this diskette the TURBO PASC'99 source files and the assembly files created by the compiler will be stored.
- Leave TURBO-PASC'99 and assemble the source files from the diskette in drive 2. Assemble them to the same disk.
- Load the linker and link the assembled files together. When you want to create a program file, you should store it on the PROGRAM DEVELOPMENT/PROGRAM EXECUTION disk, since on each of these disks you have the file RUNLIB. RUNLIB is required by all user programs.

Program Example SIEVE

```
program sieve;
{ Erathostenes sieve
  Demo for TURBO PASC'99 }

const
  size = 8190;
  loopings = 10;

var
  i,
  k,
  iter,
  count,
  prime: integer;
  flags: array[size] of boolean;

begin { sieve }
  cls;
  cursor(3,1);
  writelns("** Erathostenes' sieve");
  cursor(7,1);
  writeln("Array size ",size);
  writeln("Loops ",loopings);

  cursor(10,1);
  writeln("Iteration step");

  for iter := loopings downto 1 do
  begin
    cursor(10,19);
    write(iter:3);
    count := 0;

    for i := 0 to size do
      flags[i] := true;

    for i := 0 to 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 }

        count := count+1;
      end; { if }
    end; { for }

    screen(2,iter+6);
  end; { for }

  cursor(22,1);
  writeln(loopings," times ",count:1," primes found");
end. { sieve }
```

Program example WURM

```
program wurm;

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

var
  dir, z, s: array[2] of integer;
  display: array[2] of string[1];
  reply: string[1];
  point1, point2: integer;
  grad: integer;
  mistake1, mistake2: boolean;

procedure peekv(row, col: integer; var byte: string[1]); external;
  { This is an external assembly procedure }

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(o,k,s);
  until s>0;
end; { instructions }

procedure pokev(row, col: integer; byte: string[1]); external;
  { External assembly procedure }

procedure score;

begin
  cursor(1,1);
  write("**Score**   A:",point1:1,"   B:",point2:1);
end; { score }
```

```
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 (s1<>0) and (s2<>0);
  cursor(18,7);
  write("                            ");
end; { wait till players ready }

procedure init;

var
  i: integer;

begin
  cls;
  cursor(2,1);
  write("+--------------------------------------+"); { 38 "-" }
  for i := 3 to 22 do
  begin
    cursor(i,1);
      write(":");
      cursor(i,40);
      write(":");
  end;
  cursor(23,1);
  write("+--------------------------------------+"); { 38 "-" }
  display[1] := "A";
  z[1] := 10;
  s[1] := 5;
  dir[1] := right;
  pokev(z[1],s[1],display[1]);
  display[2] := "B";
  z[2] := 10;
  s[2] := 36;
  dir[2] := left;
  pokev[z[2],s[2],display[2]);
end; { init }

procedure direction(se1: integer);

var
  i, k, s: integer;

begin
  i := 0;
  repeat
    i := i+1;
      key(se1,k,s);
  until (s>0) or (i>(grad-1)*10);
  if s>0 then
    case k of
        left, right, up, down: dir[se1] := k;
        minint, maxint:
      end;
```

```
end; { direction }

procedure player(se1: integer; var mistake: boolean);

var
  byte: string[1];

begin
  mistake := false;
  direction(se1);
  case dir[se1] of
    left: s[se1] := s[se1]-1;
      right: s[se1] := s[se1]+1;
      up: z[se1] := z[se1]-1;
```

TURBO-PASC’99 Version 2.0

TURBO-PASC’99 Version 2.0 supports a new command

GO xxxx Allows you to position the cursor on a specific line of a program in memory.
If there are not enough lines, the cursor moves to the last one.
If no line number or about 0 is specified, the cursor is positioned on the first line of text.

This command is needed to make it easier to determine the incorrect line in the event of a runtime error.

Turbo-Pasc'99 Version. 2.0 further increased the compiler speed and reduced the space requirement of the assembly code generated by the compiler on the diskette.

Turbo-Pasc'99 Version 2.0 now supports the following controllers:

- ATRONIC disk controller
- CORCOMP disk controller
- TI disk controller

TURBO-PASC’99 VERSION 2.0

TURBO PASCAL 99 Version 3.0

This disk image and the zip file contains the program Turbo Pascal 99 version 3.

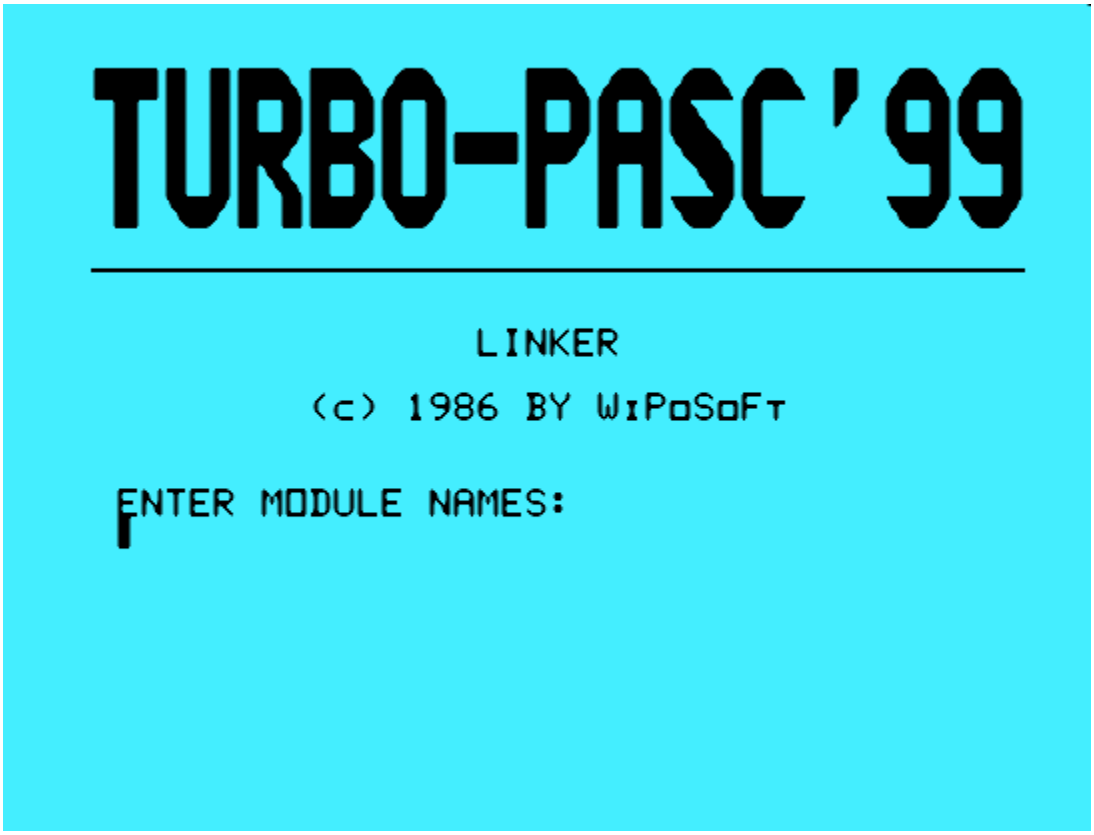
The program is not copy protected and runs from DSK1.

TP99V3A.dsk 360 kB · 10 downloads

TP99_V3A.zip 61.32 kB · 9 downloads

In classic99 you can map the disk image TP99V3A to DSK1.
Or you can unpack the zip file TP99V3A in a FIAD folder and map the folder to DSK1.


TURBO-PASC'99
EDITOR/COMPILER
(c)1989 by WiPoSoFt
VERS. 3.0
NO WORKFILE PRESENT


TURBO-PASC'99
LINKER
(c) 1986 BY WiPoSoFt
ENTER MODULE NAMES:

I don't know about the differences between version 2 and 3 but during my tests the editor / compiler and the linker works the same way.

The colors in the compiler and in the linker are different to version 2, more professional, I think.

The Compiler loads from EA option 5 RUN PROGRAM FILE and the FILE NAME: DSK1.TP3

The Linker loads from EA option 5 RUN PROGRAM FILE and the FILE NAME: DSK1.LK3A

You can use this Turbo Pascal version with the DEMO programs showed in the posts before (POST #2=TP-DEMO, POST #52=TP-WIN and POST #63=TP-LINES) or any other Turbo Pascal Program.

There are some more libraries for Turbo Pascal 99 included:

CHARS@	CHAR, CHARS, CHARP, COLOR, GCHAR, HCHAR, VCHAR, JOYST
SOUND@	SOUND, SOUND2, SOUND3, SOUND4
SPEECH@	SAY, SPGET
SPRITES@	SINIT, MAGNIF, MOTION, PATTRN, LOCATE, POSITN, SPRITE, SCOLOR, DELALL, DELSPR, DISTSS, DISTSL, COINC
PACKBIT@	CLEAR, BITMAP, PIXEL, LINE

The documentation is currently in German:

CHARSHELP	Screen and Graphic documentation
GS-BEFEHL	External Procedures from the libraries CHAR@, CHARS@, SPRITES@, SOUND@ and SPEECH@

There is also a RUNLIB80 file and a RUNLIBHELP documentation included. It seems that Turbo Pascal can be used with a 80 character card or it can produce programs for a 80 character card.

** Have fun programming with Turbo Pascal! **

Planed for but either un-written or un-released (as noted in the version of the manual from L.L. Conner Enterprises.

- TurboPasc'99 Training Guide.
- TurboPasc'99 Guide to Assembler Interfacing.
- Windows'99 – A library to create Windows in your software.
- Graphics and Sound Toolbox.

Reference Material More – added by DHE

I'm a child of the 60's, so additional information that might make your journey more rewarding an interesting.

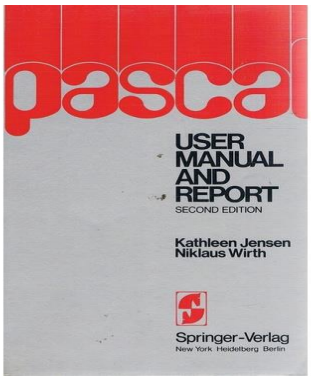
The first program written these days are usually called Hello World Programs, here it goes for Pascal:

```
program HelloWorld(output);
begin
    Write('Hello, World!')
    {No ";" is required after the last statement of a block -
        adding one adds a "null statement" to the program, which is
ignored by the compiler.}
end.
```

The Father of Pascal is **Niklaus Emil Wirth**

The Holy Bible if you will for Pascal

The Pascal User Manual and Report,



RAMBLES by Stephen Shaw January 1988

First Comments TURBO-PASC'99

At press time, I was still awaiting a positive price quotation from Texaments, prior to ordering. However, I have managed to obtain a German copy of the program and have done some benchmark tests as below.

A full review will appear as soon as I have the English version, with English documentation. I understand that Texaments have the source code and are making one or two amendments - these should not alter the benchmarks or general comments too much.

[Historic Note: Subsequently Texaments advised that they only had the rights for the USA and Canada, but this was then extended to the U.K. Although they advertised the program, the documentation appears to have never been translated nor the program issued. So this little piece really is historic. I even wrote a good fast version of Conway's Game of Life in Turbo Pasc99, not bad with German documentation... ah well].

Error handling is excellent, with many syntax errors caught on compilation. Use of defaults when you ask for the impossible mops up many run-time errors, and the range of error messages for run-time is greater than that produced by the Editor-Assembler alone. An error caught on compilation will place you back in the editor with the cursor near the error. Most helpful!

-This is NOT Borland's TURBO PASCAL(tm). It is close to "standard" ISO Pascal, amended to make it more friendly for programmers used to the TI99/4a.

ISO Pasca1 items NOT supported are:
file, in, packed, record, set, type, with, char, ord, pred, round, sqr, succ, trunc,odd, reset, rewrite, dispose, new, pack, unpack.

Items included but not in ISO PASCAL (some replacing items above):
Block, Module, Relative, Stream, String, Open, Seek, Append, Close, Asc, Cursor, Key, Screen, minint, pi, graphics, text, putln, cls, cir, cis, cri, crs, csi, csr, len, int, rnd, seg, tan, randomize.

Variations on ISO Pascal are:
Strings are within double quotes "string" instead of 'string'
REAL must be specified as 4, 6 or 8 bytes- if you select 8 you have normal Ex Bas, precision.

Some of the portability has gone with these changes of course, but they seem to make the language easier to enter for a TI-user with NO Pascal experience!

In addition, disks containing additional procedures are available- one for Windows, and one for our familiar TI extensions with sprites, sound and so on. If the language is popular, expect more in due course.

NB: The German version I have REQUIRES the Editor Assembler MODULE to function.

Pascal Books Worth looking at:

A CRASH COURSE IN PASCAL (about 8 pounds) by D M Munro. Publisher:Arno1d ISBN: 0 7131 3553

PASCAL USER MANUAL AND REPORT. Jensen and Wirth. SPRINGER-VERLAG ISBN: 0-387-96048-1 (specify third edition)

There are also books by Peter Lottrup (COMPUTE) and Zaks (Sybex) which may be worth a look. Take care: Most Pascal books these days seem to be for Borlands Turbo Pascal (NOT TURBO-PASC'99 or ISO Pascal).

TURBO-PASC'99 BENCHMARKS

These benchmarks were detailed in TI*MES issue 15, and have appeared several times in PERSONAL COMPUTER WORLD magazine.

For TURBO-PASC'99, the PASCAL code will first be given, followed by the time and possibly some notes.

Remember: You do not need a P Code card to run TURBO-PASC'99.

Integer Math

```
PROGRAM intmath;
  VAR t,
```

```
  i,x,y : INTEGER;
    BEGIN
    writeln(".........");
    t := 0;
      FOR t := 1 TO 100 DO
      BEGIN
      x := 0;
      y := 9;
      writeln("start");
         FOR i := 1 TO 1000 DO
         BEGIN
         x := x + (y * y - y) DIV y;
         END;
      END;

    writeln("---",x);
    END.
```

The timing on this program equates to a benchmark of 0.337 seconds for 1000 loops, which compares well with a benchmark of 0.48 seconds for C99.

Real Math

```
PROGRAM realmath;
  VAR t,
  i : INTEGER;
  x,y : REAL[4];
    BEGIN
    writeln( ".........");
    FOR t := 1 TO 5 DO
      BEGIN
      x := 0.0;
      y := 9.9;
      FOR i := 1 TO 1000 DO
         BEGIN
         x := x ( (y * y - y) / y;
         END;
      END;
    writeln("***END...",x);
    END.
```

REALMATH using numbers of 4 bytes, took 8.20 seconds for 1000 loops, compared to 17.7 seconds for plain ordinary TI Basic!

Trigonometry

```
PROGRAM triglog;
  VAR i : INTEGER;
  x,y : REAL[4];
  BEGIN
    writeln(".........");
    x := 0.0;
    y := 9.9;
    FOR i := 1 TO 400 DO
      BEGIN
      x := x + sin( arctan( cos( ln(y))));
      END;
```

```
  writeln("***",x);
  END.
```

This was SLOOOW and the equivalent of 1000 loops took 625 seconds, which compares badly to 360 seconds in Extended Basic!

Text to screen

```
PROGRAM textscrn;
VAR i : INTEGER;
   BEGIN
   text;
   writeln("START");
     FOR i := 1 TO 1000
     writeln ("1234567890q wertyuiop" ,i);
     writeln("***",i);
END.
```

This one took 69 seconds for a 1000 loop, again comparing badly with C99, which took just 38.7 seconds.

Storing text to disk

```
PROGRAM store;
VAR  i : INTEGER;
        f : STREAM[80];
BEGIN
   writeln("START");
   open(f, "DSK2.TEXT", output);
     FOR i := 1 TO 1000
     putln(f, "1234567890q wertyuiop ");
     close(f);
     writeln("*****");
END.
```

This took 61.4 seconds, compared to 83 seconds in Myarc XB and 131 seconds in TI XB. Notice how easy disk access can be!

TURBO-PASC’99 Benchmark and accuracy RESULTS

Benchmarks are not the be all and end all of a language, although an advertiser can give the impression a language is incredibly fast by telling you what it is fast at, and not telling you what it is sloooow at! TURBO-PASC’99 seems to be comparable with c99 overall, sometimes better sometimes not.

Personally I found it much easier to use TURBO-PASC’99- I even made it write to disk! Look how short the STORE program is!

If you are a 'TI P-Code' user you may find these codes a little odd.. I can assure you they work EXACTLY as printed here. If YOU have a TI P Code card, why not run comparative tests and let me know the times? (update: nobody did, and I never heard of anyone owning the TI P-Code Card).

ACCURACY:

The answers printed out by the math results were: intmath 8000, realmath 8.9E3 (both same as TI Basic) and triglog -2.2021E2, compared to -2.20497E2 from TI Basic- this minor inaccuracy is due to using only 4 bytes for the variable rather than 8, but we could have used 8 if we had wanted such accuracy.

<EOF>