HighSpeed Pascal

Program by: Christen Fihl (compiler and run time), Jacob V. Pedersen, Martin Eskildsen
Manual by: Christen Fihl, Jacob V. Pedersen, Martin Eskildsen, Lars Lindegren, Steen Soendergaard.
Cover and lay-out by: Hans Asmussen
Project manager: Lars Lindegren

HighSpeed Pascal is distributed in Germany, Switzerland and Austria under the name:
MAXON PASCAL
by: Maxon Computer GMBH
Industriestrasse 26
D-6236 Eschborn, Germany
Phone: (+49) 06196/481811 , Fax : (+49) 06196/41137

Second edition, August 1990

ISBN 87-983591-1-8

HighSpeed Pascal License Statement

This software is protected by law and international treaty provisions. You may copy the software solely for back-up purposes, all other copying of the software is expressly forbidden.

This software must only bee used by one person at one computer at the same time If the software is to be transfered to another computer it must be removed from the previous computer, so their is no possibility of the software being used at more than one location at a time.

Programs written with the HighSpeed Pascal Compiler may be used, given away, or sold without any license or fee to D-House or their agents.

The sample programs included on the HighSpeed Pascal diskette may be used freely as part of your compiled programs. You may not sell, or give away those sample programs as stand alone programs or as source.

Limited Warranty

As the only warranty under this agreement, and in the absence of accident, abuse or misapplication, D-House warrants, to the original Licensee only, that the disk in this package is free from defects in materials and workmanship under normal use and service for a period of 90 days from the date of payment as evidenced by a copy of the receipt.

D-House's only obligation under this agreement is, at D-House's option, to either:

1) Return payment as evidenced by a copy of the vendor's receipt

2) Replace the disk that does not meet D-House's limited warranty and which is retumed to D-House with a copy of the vendor's receipt.

The software and written material is provided „as is" without warranty of any kind including the implied warranties of merchantability and fitness for a particular purpose, even if D-House has been advised of that purpose.

D-House specifically does not accept any liability for the operation of the software neither indirect, consequential or incidental damage arising out of the use or inability to use such product, even if D-House has been advised of the possibility of such occurrence.

Registration

For your own and our benefit please register your HighSpeed Pascal.
Return the enclosed Registration Card or the registration side in this manual to qualify for:

Add-ins and application:
A number of very interesting add-ins and application to the HighSpeed Pascal will follow. You only have the possibility to integrate them in your HighSpeed Pascal if you are a registered user.

Product Upgrades:
We are continually improving our HighSpeed Pascal with enhancements and new features, only registered users can achieve these upgrades.

Technical Information:
Registered owners are among the first to hear about new products, developments, enhancements add-ins and applications from D-House.

HighSpeed Pascal User's Magazine:
We plan to publish a magazine for registered HighSpeed User's.

Free Technical Support:
Besides any support you may get instantly from your dealer or distributor, D-House gives you unlimited access to customer support by writing or sending a fax if you are a registered user.
(We offer technical support, however, we are not able to tutor in programming, please keep this in mind !.)
To reach our Technical Support by mail, write (or fax):

D-House 1 ApS.
Transformervej 29
Dk-2730 Herlev
Denmark
Fax.: (+45) 44 53 95 44

Registration Card

Product name: **HighSpeed Pascal** Serial number:

Date:

Mr. Mrs.

Last name:

First name:

Company:

Department:

Address:

Zip/Postal Code: State:

City:

Country:

Phone: Fax:

Which other compilers do you also use:

Comments:

Postage

D-House
Transformervej 29
DK-2730 Herlev
Denmark

Fax.: (+45) 44 53 95 44

Contents

Chapters:

Reference:

Appendices:

Index

Chapters:

Reference:

Appendices:

Introduction

No doubt that Pascal is a success as a programming language. It is widely used all over the world, and used as a teaching language in many schools.
In the PC world the "state of the art" Pascal Compiler is Turbo Pascal from Borland mainly written by Anders Hejlsberg.
In the Atari ST World their are many Pascal Compilers but no "state of the art" . We have tried to make a "Turbo Pascal look alike" for the Atari World. We call it HighSpeed Pascal.
Our managing software engineer, Christen Fihl has been working with Anders Hejlsberg and the Turbo Pascal team for six years, now he is working with HighSpeed Pascal.
HighSpeed Pascal is more than just an extremely fast Pascal Compiler. We know that the compiler is the programmers work place, so we try to make this work place as efficient as possible.
We do not give you three stand-alones, editor, compiler and linker to:

first edit -
then compile -
then link -
then run - your programs

In HighSpeed Pascal we put it all together in an easy-to-learn and easy-to-use integrated development environment. With HighSpeed Pascal you simply:

edit and run your programs.

We call this integrated development package The HighSpeed Integrated Development Environment (HIDE).
The first version of HighSpeed Pascal is available herewith. We will continually improve our HighSpeed Pascal with enhancements and new features. So if you ask "Is there a new version of HighSpeed Pascal on it's way ?", the answer will always be "Yes, we are working on it !".
To make this work as good as possible we will need your help so please report any bug you identify as well as proposals for enhancements to:

D-House Aps.
Att: HighSpeed
Transformervej 29
DK-2730 Herlev
Denmark
Fax: (+45) 44 53 95 44

For usable proposals we promise to send a gift (Free upgrade, free add-in etc.)

We have tried, and will in the future try to make HighSpeed Pascal compatible with source code from Personal Pascal and ST-Pascal by including the unit STPascal. Also we will try to make it as compatible with Turbo Pascal source codes as possible.

Please write/fax to us with any of your experiences with the compiler.

Please don't try to learn the Pascal Language from this manual. And although we have included a GEM chapter, please don't try to learn programming in GEM from this manual. This was not our intention, if you are a novice please find a suitable book in our appendix book list.

Also we take for granted that you are familiar with using the GEM environment, if not, please read the „Atari owners manual" supplied with your Atari Computer.

Installation

In this chapter we will get you started with HIDE (HighSpeed Integrated Development Environment). We will tell you how to install the program, and a little about the files contained on your disk(s).
The disk(s) contained in the package are formatted for standard 3 1/2" Double Sided disk drive.
!! Remember to make a back-up copy of the disk(s). (This is the only legal copy of the program you are allowed to make)
Minimum system configuration to run the HighSpeed Pascal is:
520 Kb. 1 double sided floppy (preferably). A single sided floppy can also work.
The disk that comes with HighSpeed Pascal includes two versions of the Pascal Compiler: a HighSpeed Integrated Developing Environment (HIDE), and a command-line version.
You won't need all the files from your distribution disk(s) on your personal HighSpeed Pascal system disk.
Actually you can work with as little as the HSPASCAL.PRG and still use the HIDE system.

Here is a summary of the files contained on the distribution disk(s):

Information files:
READ.ME
To see any last-minute notes and corrections. Read this !!!!!

The compiler files:
HSPASCAL.PRG
The HighSpeed Integrated Developing Environment version of the HighSpeed Pascal. Use this to edit and run your programs.

HSPC.TTP
The command-line version of HighSpeed Pascal. If you would like to write source text in your own editor you will probably want to use this

PASCAL.HLP
This contains the on-line help text used by HIDE.

PASCAL.DAT
Defaults settings, made by the HIDE version

PASCAL.CFG
File made by the user. For use with HSPC.TTP

PASCAL.LIB
A collection of unit files in one resident library.

Utility programs:

UNITS\LIBMAKER.TTP
When you compile a unit, the resulting code is placed in a *.UNI file. With the program LIBMAKER.PRG you can chain several units into a library, and then using the editor command OPTIONS/General make this library resident.
UNITS\LIBMAKER.PAS
Source code to the unit to library converter. (You can change this if you need to move a lot of files).

Unit files:

UNITS\GemDecl .UNI Standard Gem declarations (and some utilities)
UNITS\GemAES .UNI Standard Gem routines
UNITS\GemVDI .UNI Standard Vdi routines
UNITS\STPASCAL.UNI Implementation of the routines from ST-Pascal
UNITS\EasyGraf.UNI A demonstration unit, using the VDI library.
UNITS\Dos.UNI A lot of routines for TOS interface
UNITS\BIOS. Interface for the BIOS and XBIOS routines

All unit files are already included in the PASCAL.LIB file.
The Dos unit is included in the compiler itself.

Demonstration files

DOSDEMO*.* Demonstration programs
GEMDEMO*.* Gem demonstration programs
GRAFDEMO*.* Graphic demonstration programs without use of windows

*.DOC files

Documentation files. Read these !!!

Decide which files you need, choose which version of the compiler you want to use (probably the HIDE version), the .HLP file if you want on-line help available, LIBMAKER if you want to. If you do not know which then take all, and copy them to your personal "HighSpeed Pascal Systemdisk".
If you have a disk based system copy them to a floppy disk. If you have a system with harddisk copy them to a folder (directory) on your harddisk.

Getting started

In this chapter we will get you started with HIDE (HighSpeed Integrated Developing Environment).

You will learn how to load the HighSpeed system into memory, how to enter, edit, save and run your programs and how to use the on-line HighSpeed help system.
On top of that you will get an introductory to what a unit is and how to make your own units.

During the reading of this chapter you will be requested to enter four Pascal programs into the editor. None of these programs will be all that useful, they are merely examples of simple and easy-to-enter programs.
If you spot any syntactic errors in the listed programs, don't correct them. (They were created on purpose, so that you can get started on using the on-line help system.)

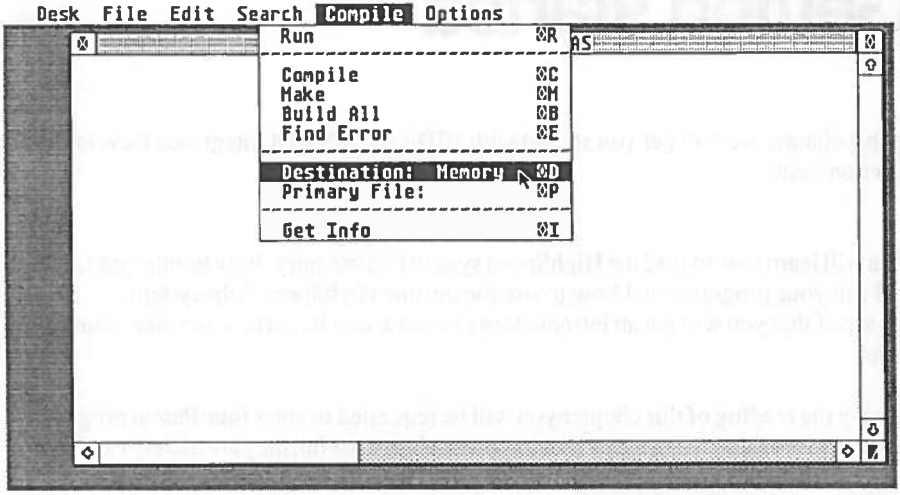
To see all about the menu selections explained in detail read the appendix „Menu"

Loading HighSpeed

Loading from a disk drive:Insert your HighSpeed diskette into your disk drive, Double click on the icon for the drive to open its file window. Start HighSpeed by double clicking on the file named HSPASCAL.PRG.

Loading from a hard-disk:Double click on the icon for your hard-disk. If you have copied the HighSpeed files into a subdirectory you must go to that subdirectory. When this has been done, you can start the HighSpeed system by double clicking on the file HSPASCAL.PRG.

The menu



Look closely at the screen, it consist of two parts, the main menu bar, and the editor window. You choose a menu selection from the menu by placing the arrow over a choice. By doing so another menu will drop down, from this window point to the choice you want, and select it by clicking the mouse.

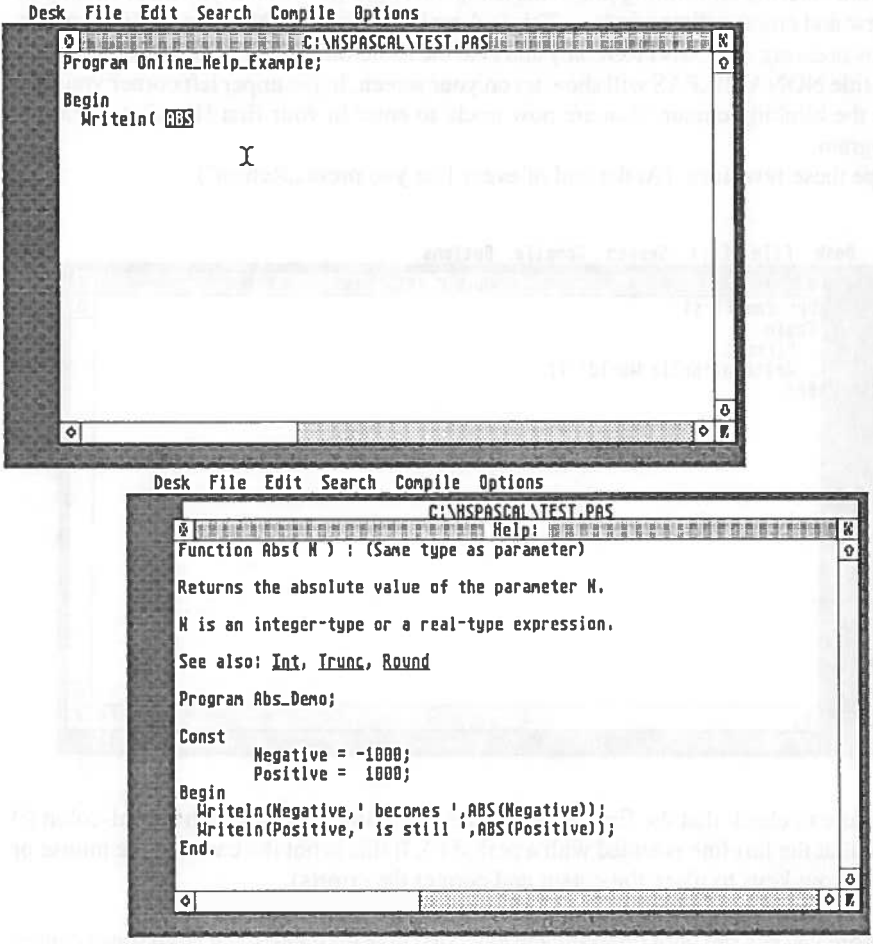
Short-cuts

Instead of selecting with the mouse you can use a short-cut. You can see the short-cuts to the right of the choice's in the drop down menus.

New ^N Means that you should hold down the „Control" key and at the same time press the key „N".

Run ◊R Means that you should hold down the „Altemate" key and at the same time press the key „R"

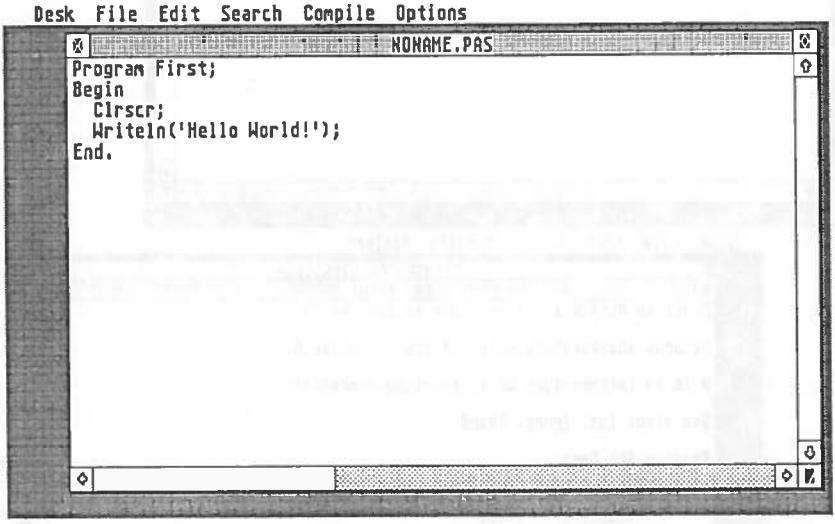
Help



From anywhere in the program (except in the dialogue boxes) HIDE provides on-screen help at the touch of the HELP key. Pressing HELP will open a help window, the help window can contain one or more keywords (an item with an underscore) on which you can get more information. To get more information just double-click a keyword and a new help window will appear. Pressing the „Undo" key close the Help window. You can also click on a word in the editor for example „Abs", press the „Help" key and get help on the Abs function.

Creating your first program.

Before entering and editing programs using the HighSpeed editor you must first open a new and empty editor window. This is done by clicking on NEW in the FILE menu or by pressing the CONTROL key and N at the same time. Either way a window with the title NONAME.PAS will show up on your screen. In the upper left corner you will see the blinking cursor. You are now ready to enter in your first HighSpeed Pascal program.
Type these few lines: (At the end of every line you press „Return")



Be sure to check that the first, third and the fourth line is ended with a semi-colon (;) and that the last line is ended with a period (.). If this is not the case use the mouse or the arrow-keys to place the cursor and correct the error(s).

Before you can run your program you must first give it a name since HighSpeed cannot run a NONAME.PAS program. To name a file choose SAVE or SAVE AS in the FILE menu. When you have done this an item selector box will appear on your screen. Now type in the name under which you wish to save your program (i.e.. FIRST.PAS) and press the RETURN key. When HighSpeed has finished saving your program you will notice that the title on the editor window now corresponds to the name you just entered. You are now ready to run your program. Do this by choosing RUN in the COMPILE menu (or by pressing the ALTERNATE key and R at the same time). The screen will now be cleared and a line reading „Hello World!" will be displayed. The next line will read „Press a key...". Do so and you will again be able to see the editor window.
If you do not wish the „Press a key..." message to appear every time you run a program from within the editor, you can disable it in the OPTIONS/Run menu.
When your program has finished executing, you are retumed to the place in your program where you started. You can now continue the editing.

The User Screen

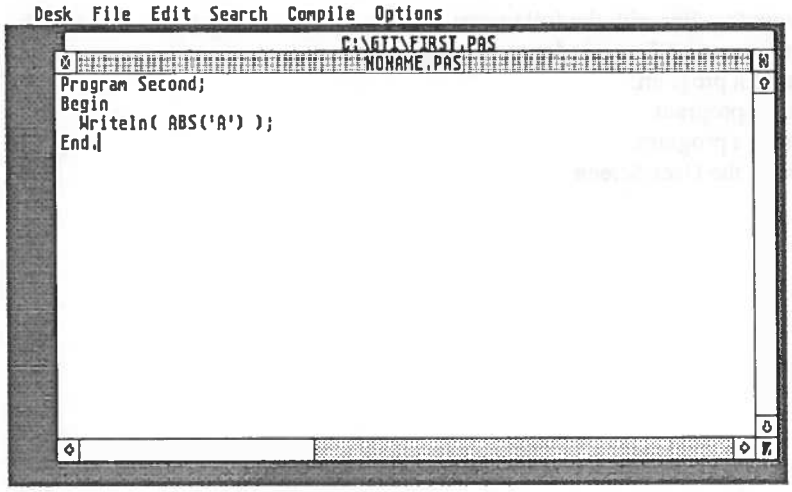
Once you're back in the HIDE after executing the program, you can view the program output by choosing the SEARCH/User screen (Esc) command. By pressing any key you return to HIDE.

You are now familiar with the following:

- Opening a new editor window.
- Entering a program.
- Saving a program.
- Running a program.
- Showing the User Screen

Using the on-line help system.

Open a new editor window by pressing CONTROL and N at the same time. When the editor window appears start entering this program:

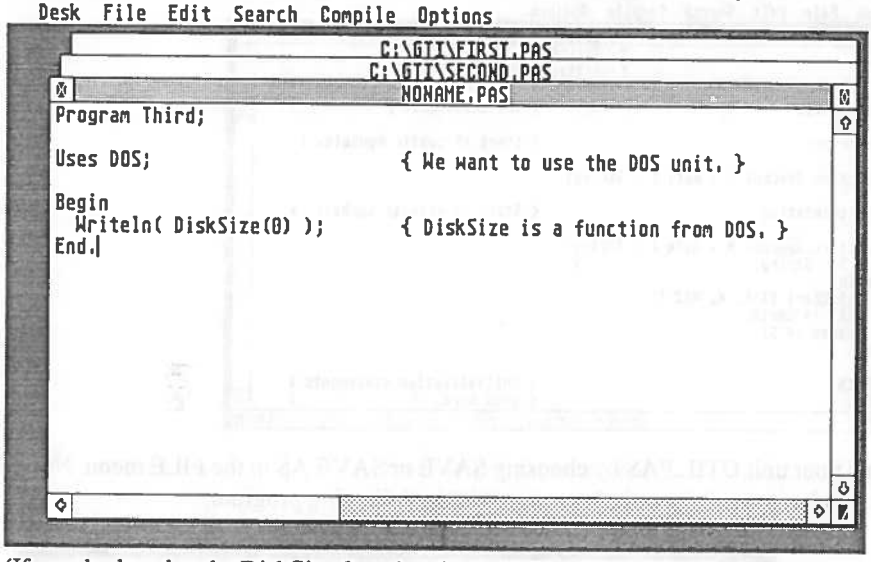


Make sure that you entered each line correctly and then name your program by choosing SAVE or SAVE AS in the FILE menu.

When you try to run the program HighSpeed will immediately stop the compilation and display an error message. Acknowledge the message by pressing the RETURN key. As you can see from the location of the cursor, the problem is the parameter to the ABS function. In order to get some aid from the help system simply double click on the word ABS and press the HELP key.
When you have done this a help screen will appear, showing you the syntax of the ABS function, a small explanation of what it does, together with a small demo program. On top of all this there is a „See Also" line with a number of underlined keywords. (Double clicking on one of these will bring up new help pages).
As you can see from the ABS function help page the ABS function takes an integer-type or real-type parameter and not a char-type one as we entered earlier. Now close the help window by pressing the UNDO key, correct the error and your second program will be able to run.

Using a standard HighSpeed unit.

In your third program you will learn how to use units. A unit is a collection of pre-compiled procedures, functions and data. The smart thing about units is that they don't have to be compiled every time you compile a program that uses them. They only have to be linked with your program. A fact that should encourage you to use units whenever possible, thereby increasing compilation speed.
Now open a new editor window, and start entering the following program:



(If you doubt what the DiskSize function does, double click on the word DiskSize and press the HELP key.)

Name your program, and then run it.

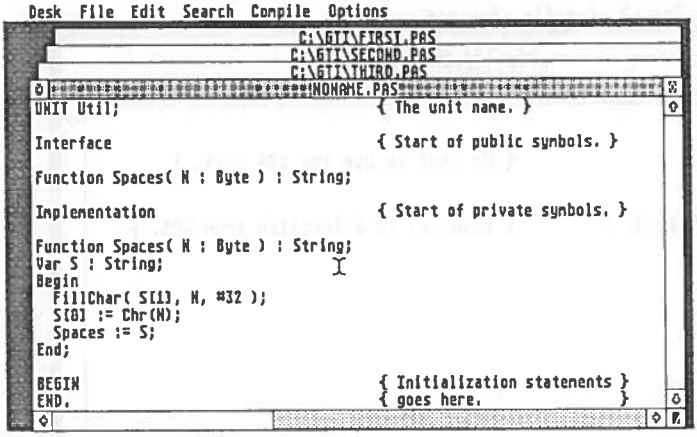
Creating your own unit.

The following will show you the steps necessary to create your own unit.
Let's say that you have created a function called Spaces:

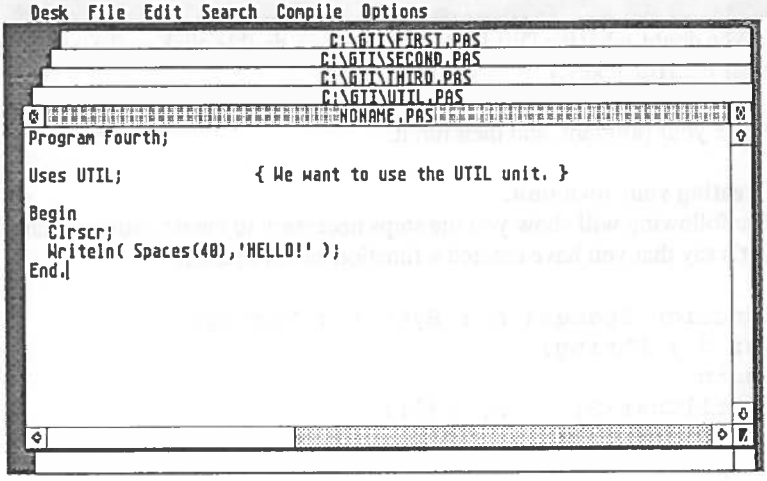
```
Function Spaces( N : Byte ) : String;
Var S : String;
Begin
  FillChar(S[1], N, #32);
  S[0] := Chr(N);
  Spaces := S;
End;
```

If you use this function in several of your programs and consequently you have a number of identical copies of the function.
What happens if you need to make a change to the Spaces function?

You will have to locate all the programs that uses the Spaces function, and then make the same change in all of these programs!
Why not put the Spaces function into a unit?. Doing so will make the future changes to the Spaces functions much easier since the only place you will have to make the changes is in the unit.
Open a new editor window and start entering the following, which is a unit called UTIL, containing the Spaces function:



Name your unit UTIL.PAS by choosing SAVE or SAVE AS in the FILE menu. Now that you have created a unit, let us use it in the following program:



Name your program FOURTH.PAS.
Since you have created a new unit which has not yet been compiled you must now choose BUILD or MAKE in the COMPILE menu. This will make HighSpeed compile your UTIL unit as well as the program you just entered.

The menu

To see all about the menu selections explained in detail see the app. "Menu".

Desk:
Information about copyright and compiler version.

File:
Loading existing files ..
Creating new files ..
Saving the file ..
Printing a file - in the editor
Quitting HighSpeed
Execute a program outside HighSpeed

Edit:
Cut/Copy/Paste text
Indent and outdent text
Get info on source program

Search:
Find and replace text.
Toggle between windows

Compile:
Compile, Make or Build a program.
Set the destination of the compiled code (disk or memory).
Find a run-time error.
Set the primary file.
Get information about the compiled file.

Options:
Contains general settings that control how HIDEks:
- General settings.
- Compiler settings.
- Linker settings.
- Run settings.

Save all options to disk.
Also contains the help system.

Language Elements

Basic Symbols

The smallest units of text in a Pascal program are called tokens. They are classified into special symbols, word symbols, numbers and character strings.

These are the basic building blocks:

Letter =
A to Z, a to z and underscore '_'.

Digits=
The ten digits from 0 to 9.

HexDigits =
The normal digits, A to F and a to f.

Blank =
All control characters and the space characters.

Special =

```
+ - * / = < > ( ) [ ] { } , . ; : ' ^ @ $ #
<>   <=   >=   :=   ..   (*   *)   (.   .).
```

The last line consists of nine symbols, two characters each.

Below mentioned symbols have the same meaning:

```
(* and {
*) and }
(. and [
.) and ]
```

Separators

Separators can be a blank, a newline or a comment. Actually all control characters will work. Two word symbols must be separated by at least one separator. Otherwise the compiler will see only one symbol.

Comments

Comments can be inserted anywhere where you are allowed to insert a blank or new line. A comment is surrounded by { and } or by (* and *). But not by { and *) and not by (* and }.
A comment can span over several lines.

```
{ This is a comment }
(* This is also a comment *)
{ This is also a comment *) }
{ But this comment never ends! *)
```

If the first character right after the opening { or (* is a $ character, then the comment is compiler directive.

Program lines

Each program line can be 127 characters long. The rest of the line is simply discarded. This can be dangerous for your programs, because you might not necessarily get a compile error. One example is starting a comment in position 1 and ending it in position 135 of the line. This comment will last forever, or until the end of another comment.

Reserved Words

These word symbols are reserved in HighSpeed Pascal, and cannot be redefined.

WordSymbol = and, array, begin, case, const, div, do, downto, else, end, external, file, for, forward, function, goto, if, implementation, in, inline, interface, label, mod, nil, not, of, or, packed, procedure, program, record, repeat, set, shl, shr, string, then, to, type, unit, until, uses, var, while, with, xor.

Identifiers

Identifiers denote labels, constants, variables, procedure, functions, units, programs and fields in records. An identifier consists of a letter symbol followed by letter and digit symbols.

Identifier =
    Letter { Letter | Digit }

Numbers

The compiler is made for number crunching, so we need numbers too.
Some examples:

```
7   9   13
$12 $AA55 $000F
1.2 3.1415927 1.23e17 1e-9
```

Numbers used for integer types are those made by using digits only. They can also be written using hex notation, by prefixing it with a $ character.

Numbers for reals are made by using the normal engineering notation format. 1.23e4 is the same as 12300.00 or 12300.

```
UnsignedNumber =
   UnsignedInteger | UnsignedReal.

UnsignedInteger =
   DigitSequence | "#" HexDigitSequence.

UnsignedReal =
   UnsignedInteger "." DigitSequence ["e" SCaleFactor] |
   UnsignedInteger "e" ScaleFactor.

ScaleFactor =
   [ Sign ] UnsignedInteger.

Sign =
   "+" | "-"

DigitSequence =
    Digit { Digit }.

HexDigitSequence=
   HexDigit { HexDigit }.

SignedNumber =
   SignedInteger | SignedReal.

SignedInteger =
   [ Sign ] UnsignedInteger.

SignedReal =
   [ Sign ] UnsignedReal.
```

Strings

A character string are sequences of characters within quotation marks. A quotation mark can itself be included by entering it twice.

A string can have a length of zero, one or more. It is always compatible with variables of type strings. With a length of one, it is also compatible with variables of type char. With a length of N it is compatible with packed array [1..N] of char.

Like Turbo Pascal, HighSpeed Pascal can include control characters to be entered into the string. They can be included as #nn or as ^C. #nn enables you to write any ASCII character in the range 0 to 255. ^C enables you to write normal control characters in the range from C='A' to C='Z'. It is easier to write ^Z instead of #$1A or #26.

```
CharacterString =
  "'" { StringElement } "'"

StringElement =
  "''" | AnyCharacterExceptApostrophe.
```

Some examples:

```
'Plain vanilla'
'It''s Ok'#13#10
''
'.'
'a'
'A'
```

The control characters entered into a string must be entered outside of the apostrophes without using any spaces.

Constant Declarations

A constant declaration declares a constant identifier, and gives it a value.

```
ConstantDeclarationPart =
 [ "const" ConstantDeclaration
 { ConstantDeclaration } ].

ConstantDeclaration =
  Identifier "=" Constant ";".

Constant =
  [ Sign ] UnsignedNumber |
  [ Sign ] ConstantIdentifier |
  CharacterString.

ConstantIdentifier =
  Identifier.
```

These constants are predefined in HighSpeed Pascal:

```
false, true          of type boolean,
maxint, maxlongint   of type integer (longint)
pi                   of type real (extended)
```

Compiler Directives

Source code control is done by using compiler directives.
A compiler directives is a comment where the first character is a $-sign. No spaces are allowed before the dollar sign.

```
{$Define Debug}
(*$Ifdef Debug *)
{$R-}
{ $Not a compiler directive! }
```

Types

Every variable in Pascal has a type. The type of a variable determines the values that the variable can assume and the operations that can be performed on it.

Type declaration part

The type declaration follows right after the "type" keyword. In HighSpeed Pascal there can be more than one "type" declaration block, and it can be mixed with constant, label, procedure, function and variable declarations.

```
TypeDeclarationPart =
  "type" TypeDeclaration { TypeDeclaration }.
```

Example:

```
Type
  NewInt  = Integer;
  BestInt = NewInt;
  MyChar  = Char;
  Color   = (Red, Green, Blue);
  Byte    = 0..255;
  Weight  = 10..25;
  PRect   = ^Rect;
  Rect    = Record
              x, y, w, h: Integer;
            end;
  Rects   = Array [1..99] of Rect;
  Mixed   = Array[1..4] of Record a,b: Integer  ;
```

Type declaration

```
TypeDeclaration =
  Identifier "=" Type ";".

Type =
  TypeIdentifier |
  SimpleType |
  StructuredType |
  StringType |
  PointerType.
```

Simple Types/Ordinal Types

```
SimpleType =
  OrdinalType | RealType
```

A simple type defines an ordered set of values. This means that you can compare to variables (of same type) and see if one is greater than the other.

```
OrdinalType =
  SubrangeType | EnumeratedType | OrdinalType
```

Ordinal types can be worked on using these five routines:
- Ord returns the ordinality of a value
- Succ returns the value right after the one given
- Pred returns the value just before the one given
- Inc increments the value (default to next value)
- Dec decrements the value

```
Succ(Pred(y)) = y
```

The opposite function of Ord can be made by using type casting.
The boolean value True=Boolean(1). Do not make constructions like this:

```
if Boolean(2)=true then ThisMightBeExecuted;
```

These other words for types: Integer, Boolean, Char, Enumerated and subrange are all of ordinal type.

Integer

HighSpeed Pascal gives you five predefined integer types: ShortInt, Byte, Integer and LongInt. They are defined as:

```
ShortInt  = -128..127;
Byte      = 0..255;
Integer   = -32768..32767;            {-MaxInt-1..MaxInt}
LongInt   = -2147483648..2147483647; {-MaxLongInt-1..MaxLongInt}
Word      =  0..65535
```

A variable of type ShortInt uses one byte of memory. Byte and Integers will use 2 bytes, and LongInt will use 4 bytes.

Boolean

The Boolean type defines two constants, True and False. They are defined as:

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

Note that: Ord(False)=0, Ord(True)=1 and that False<True.

Char

The Char type gives 256 different characters values. The character set used is the one used on the Atari. The biggest problem you will get with most character sets, is the way they sort national characters. You cannot easily sort characters if they uses 8-bit characters as all national letters does. This does always hold on the Atari:

```
'0' < '1' < '2' < '3' < '4' < '5' < '6' < '7' <
'8' < '9'
'A' < 'B' < ... 'Y' < 'Z'
'a' < 'b' < ... 'y' < 'z'
'9' < 'A' < 'Z' < 'a' < 'z'
```

Enumerated

```
EnumeratedType =
    "(" IdentifierList ")".

IdentifierList =
    Identifier { "," Identifier }.
```

Examples of enumerated types:

```
Game    = (Club, Diamond, Heart, Spade)
Weekday = (Monday, Tuesday, Wednesday, Thursday, Friday,
           Saturday, Sunday)
```

The first identifier has an ordinality or zero. The following one, and so on.

Subrange

A subrange, as the name says gives you access to a limited range of the values in a host type. The subrange is specified by writing the lower and the upper limit wanted.

```
WorkDay  = Monday..Friday;
ShortInt = -128..127;
```

Note that Ord of an enumerated type variable cannot be less than zero, while Ord of a subrange type can be anything.

Real

So much about ordinal type. The last type in SimpleType is RealType. Real types can like normal ordinals be compared, but they are not very useful for counting, because there is no ordinality on reals. You cannot be sure that you can change a real just by adding one to it!

All real types are predefined and cannot be extended by new user types. Three kinds of reals are implemented: Single, Double and Extended.

Note: The type Real is the same as Double. This can be redefined if wanted simply by writing: Type Real=Single;

The type Extended is an internal format used when calculating or converting real values. It can be used as a normal type in your programs.
If used as Type Real=Extended; it will give:
-faster code because variables does not have to be converted before calculations
- better precision on standard arithmetic
- but not much better precision on the trigonometric functions

Single and double are IEEE compatible but the extended format are not.
This may give problem if extended are used for disk files or other interface purposes.

Single reals are moved around very fast in memory because they only take four bytes But. they still gets converted when used in procedure calls and expressions.

This is what you get from the reals:

Type	Range	Useful digits
Single	3.4e-38..3.4e38	7-8
Real=Double	1.7e-308..1.7e308	15-16
Extended	1.1e-4932..1.1e4932	18-19

Structured Types

Simple types can only hold one value for each variable. Structured types holds more than one value at a time. Moreover, a structured type may be packed. If the structure type is prefixed with "packed", then the compiler will try to save storage. Set types are always packed.

The components of a structure may itself be structured.

```
StructuredTypes =
  StructuredTypeIdentifier |
  [ "packed" ] UnpackedStructuredType.

UnpackedStructuredType =
  ArrayType | RecordType | SetType | FileType.

StructuredTypeIdentifier =
  TypeIdentifier.
```

Array

The array type has only one element type (component type), but can contain, a fixed number of them.

```
ArrayType =
  "array" "[" IndexType { "," IndexType } "]" "of"
ComponentType.

IndexType =
  OrdinalType.
```

```
ComponentType =
  Type.
```

The array can be multidimensional. If it is, the declaration can be written in different ways:

```
Way1=array[boolean,7..9] of integer
Way2=array[boolean] of array[7..9] of integer;
```

Both can be accessed (indexed) in either of these ways:

```
n:=Way1[true][8];  or n:=Way1[true,8];
n:=Way2[true][8];  or n:=Way2[true,8];
```

Packed array[1..N] of char does have special properties, not given to other array types: They can be assigned to strings.

Note: When you write array[7..9] of integer; you do create a new subrange type NoName=7..9;, it just don't have a name.

Record

Array type gave one element type but could contain a fixed number of them. Record types are able to contain a fixed number of components with different types.

```
RecordType =
   "record" [ FieldList  [ ";" ] ] "end".

FieldList =
  FixedPart [ ";" VarientPart ] | VarientPart.

FixedPart =
  RecordSection { ";" RecordSection }.

RecordSection  =
  IdentifierList ":" Type.

VariantPart =
  "case" VariantSelector "of" Variant { ";" Variant }.

Variant =
  Constant { "," Constant } ";" "(" FieldList ")".

VariantSelector =
  [ TagField ":" ] TagType.
```

```
TagType =
  OrdinalTypeIdentifier.

TagField =
  Identifier.
```

The Identifier List names each element, as in a variable declaration.

A normal record without variant fields look like this:

```
DateRecord=
  record
    Year:  Integer;
    Month: 1..12;
    Day:   1..31;
  end;
```

With a variant field it looks like this:

```
Graphix=
  record
    x,y:   Integer;
  case Obj: ObjType of
    Dot:   ();
    Line:  (x2,y2: Integer);
    Circle:(Diameter: Integer)
  end;
```

n example of its use:

```
var G: Graphics;
begin
  with G do begin
    x:=7; y:=8; Obj:=Line; x2:=13; y2:=45;
    DrawIt(G);
  end;
end;
```

One use of variant part is for data conversion:

```
var CV: record
        case integer of    {Does not use any space}
        0: (L: LongInt);
        1: (Hi,Lo: Integer);
        end;
begin
  CV.L:=$00110012;
  writeln(CV.Hi,' ',CV.Lo);
end;
```

Set

A set-type variable can contain zero or more elements of the base type.
It can only hold zero or one element of each value.

```
SetType  = "set" "of" BaseType.
BaseType = OrdinalType.
```

In HighSpeed Pascal the ordinallity of all elements of a set type must be within the range of 0..255. You cannot make a set type as: set of 0..999999.

Examples:

```
set of char
set of 10..20
```

```
Program Test;
Var
  S: Set of 0..99;
begin
  S:=[7,9,13,7];      {Three elements, 7, 9 and 13}
end.
```

File

File type is a linear sequence of components that are all of the same type.
The component type can be any type as long as it does not include any file type elements.

The array type looks almost as a file type, but is of fixed size. The file type can contain from zero to "x" elements. "x" is determined by the amount of avaliable storage space.

The predefined type: Text is a special file type in which the elements are lines of text. A variable of type Text is called a textfile. Readln, eoln and other procedures and functions works on textfiles.

```
FileType =
  "file" "of" ComponentType.
```

String

A string type value is a sequence of characters that has a dynamic length within a fixed maximum length. The length can range from zero for the empty string up to 255 for a string of maximal length.

```
StringType =
      "string" [ "[" SizeAttribute "]" ].
```

```
SizeAttribute =
  UnsignedInteger.
```

The SizeAttribute specifies the maximal length allowed for the string variable, not the actual length of the string.

String type values can be compared using equal, greater-than etc.

Examples of comparison:

```
'Hello' = 'Hello'
'Hello' < 'Hello World'
'Hello' > ''
'Hello' < 'hello'
```

The char elements in a string can be accessed like components of an array type.

Special procedure and functions exist for strings. The operator + can be use for concatenating strings.

Pointer

A pointer type defines a set of values that points to variables (dynamic or not) of a specified type, the base type.

```
PointerType =
  "^" BaseType.

BaseType =
  OrdinalType.
```

The base type must be declared before the end of the current type definition block.

Values can be assigned to pointers with the New procedure, the @ operator or the Ptr function.

Variables

Variable declaration part

The variable declaration lists all the variables used in a program, unit, procedure or function.

```
VariableDeclarationPart =
  "var" VariableDeclaration { VariabelDeclaration }.
```

Example:

```
var
  Counter,
  M,N,O   : Integer;
  C1, C2  : Color;
  BigSet  : Set of Byte;
  F1      : Text;
  F2      : File of Rect;
  FileMix : File of Mixed;
```

Variable declaration

A variable can only take values according to the type used for its declaration. A variable is undefined until it is assigned a value.

```
VariableDeclaration =
  IdentifierList ":" Type ";".
```

Variables in different places

Global variables are those in programs and units, but outside of procedure and functions. Each global variable can only be as large as 32 KByte, but the total size of globals are not limited by the compiler.

Variables in procedures and functions are created when the procedure is activated and removed right after the activation is terminated. Each procedure can only have 32 KBytes of local variables, but the total size of the stack can be as large as memory allows. Using too much stack space might stop the program with a runtime error if the {$S+} is active.

Dynamic variables are created on the heap with New and removed again with Dispose. The heap can be as big as free memory allows it

Use of Variables

Using a variable is called referencing a variable.

```
VariableReference =
  [ VariableIdentifier | VariableTypeCast ] { Qualifier }
```

Qualifiers

```
Qualifier =
  [ Index | FieldDesignator | "^" ]
```

A variable can be used just by writing its name thereby referencing the entire variable:

```
Rect
Person
```

The variable can be followed by zero or more qualifiers, thereby referencing smaller and smaller parts of the variable:

```
Rect.Point1
Rect.Point1.X
Person[3].Name
ZipCode[ Person[7].ZipIndex ]
P^[N][8].R^^
```

Arrays and String indexes

Whenever the referenced part of a variable is an array, it can be qualified with an array index. Strings can be referenced like arrays.

```
Index =
  "[" Expression { "," Expression } "]".
```

The index used must be of an assignment compatible type with the corresponding index type used for the declaration.

Multiple indexes or multiple expressions within an index can be freely used:

```
AnArray[1,2,3]    is the same as writing
AnArray[1,2][3]   or as
AnArray[1][2][3]
```

As mentioned, strings can be indexed as normal arrays. A string is always defined as String[n]=array [0..n] of char (not exactly right).
The first index [0] is the actual length of the string. Obtaining the length of a string can then also be done this way:

```
Length(Str) = Ord(Str[0])
```

Records

Whenever the referenced part of a variable is a record, it can be qualified with a field designator.

```
FieldDesignator =
  "." FieldIdentifier.
Person.Name
Person.Name.First
```

Pointers

Whenever the referenced part of a variable is a pointer, it can be qualified by writing a pointer symbol after the variable.

```
Person.Father^
Person.Next^
P^
```

Variable Typecast

A variable reference can have its type changed by applying another type to it. This is done by writing the new type as a function taking the variable as argument, and returning the variable with the new type. The function name is the name of the new type wanted.

```
VariableTypeCast =
  TypeIdentifier "(" VariableReference ")".
```

```
type
  TRect: record
           x,y: Integer;
         end;
var
  R: TRect;
  L: Longint;
begin
  L:=Longint(R);
  TRect(L).y:=3;
  Inc(Longint(R),$00020002);
end.
```

Expressions

The productive part of a program is made by using expressions and statements. Expressions are made up of operators and operands. Some operators are unary, that only takes one operand as "not true", while others are binary and takes two operands as in "2+3".

The operators have different precedences which tells which operator to use first when more than one operator is applied at a time. Parenthesized expressions are always evaluated first. If an operand is located between two operators of different precedence it is bound to the operator with the higher precedence. Sequences of operators of same precedence are executed from left to right.

Operator precedence:

Operators	Precedence	Operator
@, not	1st	unary
*, /, div, mod, and, shl, shr	2th	multiplying
,+ -, or, xor	3th	adding
=, <>, <, <=, >, >=, in	4th	relational

Expression syntax:

```
Expression =
  SimpleExpression [ RelationalOperator SimpleExpression ]

SimpleExpression =
  [ Sign ] Term { AddingOperator Term}

Term =
  Factor { MultiplyingOperator Factor }

Factor =
  UnsignedConstant |
  Variable |
  SetConstructor |
  FunctionCall |
  ValueTypeCast |
  "not" Factor |
  VariableReference |
  "@" VariableReference |
  "@" ProcedureIdentifier |
  "@" FunctionIdentifier |
  "(" Expression ")".

UnsignedConstant =
  UnsignedNumber |
  CharacterString |
  ConstantIdentifier | "nil".

SetConstructor =
  "[" [ ElementDescription
  { "," ElementDescription } ]
  "]".
ElementDescription =
 Expression [ ".." Expression ].
```

Arithmetic Operators

Arithmetic operators work on integer and real type operands and returns integer and real results.

Unary arithmetic operations

Operator	Operation	Type of Operand	Type of result
+	none	integer/real	integer/real
-	negation	integer/real	integer/real

Binary arithmetic operations

Operator	Operation	Type of Operand	Type of result
+	addition	integer/real	integer/real
-	subtraction	integer/real	integer/real
*	multiplication	integer/real	integer/real
/	division	integer/real	real
div	integer division	integer	integer
mod	remainder	integer	integer

Any type of real gets converted to extended type. If one of the operands are of real type then the type of the result isalways extended; or else if one of the operands are of type longint then the result is also of longint type else the result is of integer type.
The rules used by div and mod are:

```
i mod j = i - ( i div j) * j
```

Examples:

```
+10 div +3 = +3     +10 mod +3 = +1
-10 div +3 = -3     -10 mod +3 = -1
+10 div -3 = -3     +10 mod -3 = +1
-10 div -3 = +3     -10 mod -3 = -1
```

Boolean Operators

Boolean operations

```
The not operator negates a boolean value (monadic).
The and operator returns the logical and.
The or operator returns the logical or.
The xor operator returns the logical xor.
```

Boolean evaluation:

Op1	Op2	and	or	xor
false	false	false	false	false
false	true	false	true	true
true	false	false	true	true
true	true	true	true	false

Logical Operators

Logical operations

The operators not, and, or and xor can also be used as bitwise operators. Bitwise operators works on all bits in the operand(s).
Besides the operators from boolean you will also find the operators shl and shr.

The not operator makes bitwise negation (monadic).
The and operator makes bitwise and.
The or operator makes bitwise or.
The xor operator makes bitwise xor.
The shl operator shifts bitwise to the left.
The xor operator shifts bitwise to the right.

Examples:

```
2 shl 3 = 16
$ffff xor $f = $fff0
not $a5f0 = $5a0f
```

Set Operators

Three set operands exists:

```
+  union
-  difference
*  intersection
```

The operands must be of compatible types, and the result is of type "set of a..b", where a and b are the smallest and largest ordinal value which is a member of the result.

Set union (a+b) returns all members which are EITHER in a OR in b or in both.
Set difference (a-b) returns all members IN a but NOT IN b.
Set intersection (a*b) returns all members that are in BOTH a and b.

Relational Operators

All relational operators return boolean values. They are used when comparing values against each other.

Comparing Ordinals

`Operands applicable: =, <>, <, <=, >, >=.`

Each ordinal operand must be of compatible type.
The result is the mathematical relation of their ordinalities.
Example:

```
Red < Green
```

Comparing Reals

```
Operands applicable: =, <>, <, <=, >, >=.
```

Each operand is first converted to extended type before tested. Take care when comparing reals to be equal and not-equal. An example:

```
var
  ASingle: Single;
begin
  ASingle:=1/3;
  if ASingle=1/3 then NeverExecuted;
```

The first 1/3 is only saved with single precision before converted while the next 1/3 is constant and calculated as an extended right away.

Comparing Strings

```
Operands applicable: =, <>, <, <=, >, >=.
```

All strings are compatible so all strings can be compared.
A char variable can be treated as a string with an actual length of 1.

Examples of comparison:

```
'Hello' = 'Hello'
'Hello' < 'Hello World'
'Hello' > ''
'Hello' < 'hello'
'Hello' <= 'hello'
'Hello' <> 'hello'
```

Strings are compared according to the Atari character set. This can be a problem when sorting strings containing national characters .

Comparing Packed Strings

```
Operands applicable: =, <>, <, <=, >, >=.
```

Both operands must have the same number of elements.
Packed strings are compared like strings (with the same length).

Comparing Pointers

```
Operands applicable: =, <>.
```

The pointers compared must be of compatible types. Two pointers are only equal if they point to the same element.

Comparing Sets

```
Operands applicable: =, <>, <=, >=.
```

Both operands must be of compatible types. If a and b are sets, then:
- a=b is true if all members in a are members of b and the other way around.
- a<>b is true if no member in a are member of b and the other way around
- a<=b is true if all members in a are also a member of b.
- a>=b is true if all members in b are also a member of a.

Testing Set Membership

```
Operands applicable: in.
```

The in operator test to see if the ordinal type operand on the left is a member of the set operand on the right. If so, it returns true else false.

The ordinal type on left side must be compatible with the base type of the right operand.

Example:

```
2 in [1..4] = true
2 in [7,9,13] = false
```

The @ Operator

The @ operator creates a pointer with a type compatible with all pointers. The pointer can be assigned to any pointer variable. The operator works on variables, procedures and functions.

Do not use the @ operator if you don't have to. It is not a standard feature of Pascal. It is known in Turbo Pascal.
Because the type returned is compatible with all pointers, you do not get any type checking error from the compiler. You can easily (by accident) assign the address of a char to an integer-pointer.

Function Calls

```
FunctionCall =
  FunctionIdentifier [ ActualParameterList ].
ActualParameterList =
  "(" ActualParameter { "," ActualParameter } ")".
ActualParameter =
  Expression | VariableReference.
```

A function call activates the function specified by the function identifier in the same way as a procedure call does. The procedure call does not return anything while a function call returns a value.

If the function declaration has a list of formal parameters the function call must also have a corresponding list of actual parameters.

Examples

```
Max(i,j)
Random
```

Set Constructors

Set values are created by writing expressions within brackets.

```
SetConstructor =
  "[" [ MemberGroup ] "]".

MemberGroup =
  Expression { ".." Expression }.
```

All members in the member group must be of compatible types. The result is of type "set of a..b", where a and b are the smallest and largest ordinal value that is a member of the result.

Examples:

```
[]
[May,June]
['0'..'9','a'..'z']
```

Value Typecast

```
ValueTypeCast =
  TypeIdentifier "(" Expression ")".
```

The value type cast can change the type of an expression from one type to another. The type returned is the one given, and the value is the one with the same ordinallity. The expression must be of ordinal type or of pointer type.

Examples:

```
Char(65)    {The same as chr(65)}
Boolean(1)  {Same as True}
Longint(@Buffer)
```

Statements

Statements are said to be executable. It is here your program will do its work.

```
Statement =
  { Label ":" } { SimpleStatement | StructuredStatement }.
Label =
  DigitSequence | Identifier.
```

Simple Statements

The statement is as shown divided into two types of statements. The simple statement is a self contained statement.

```
SimpleStatement =
  EmptyStatement |
  AssignmentStatement |
  ProcedureStatement |
  GotoStatement.

  EmptyStatement = . (Nothing)
```

Structured Statements

The other main type of statements are structured statements. They contain other statements which have to be executed in:
- sequence (compound statements),
- conditionally (if and case statements),
- repeatedly (repeat, while and for statements) or
- within an expanded scope (with statements).

```
StructuredStatement =
  CompoundStatement |
  IfStatement |
  CaseStatement |
  ForStatement |
  RepeatStatement |
  WhileStatement |
  WithStatement.
```

Assignment

```
AssignmentStatement =
  ( Variable | FunctionIdentifier ) ":=" Expression.
```

A function identifier can only appear on the left side of a := sign when assigning a return value to a function (within the function itself).

The expression must be assignment compatible with the type of the result variable or function identifier.

Examples:

```
n:=3;
MyFunc:='Hello';
Letters:=['a'..'z','A'..'Z'];
```

Note: The ; in the last line is NOT part of the statement but a separator before the next (empty) statement.

Procedure Statements

```
ProcedureStatement =
  procedureIdentifier [ ActualParameterList ] |
  "Write" [ WriteParameterList ] |
  "Writeln" [ WriteParameterList ].
```

A procedure call activates the procedure specified by the procedure identifier in the same way as a function call does. The procedure call does not return any value.

If the procedure declaration has a list of formal parameters the procedure call must also have a corresponding list of actual parameters.

Examples

```
SkipUntilEOF;
Randomize;
Writeln('Goodbye');
```

Write and Writeln are special because they can take a variable number of parameters. This cannot be done by using normal procedure definitions.

Goto Statements

Labels can prefix statements which enables you to jump around in the program with goto. But try to avoid "goto" and use the "repeat" , "while" and other high-level statements instead.

GotoStatement =
"goto" Label.

Two rules must be remembered:
- The goto and the corresponding label must be in the same block.
- Do not jump into a structured statement. The for loop statement does some initial calculations which may not be skipped.

Compound Statements

Write begin and end around a sequence of statements, and you have shown the order in which to execute them one by one.

CompoundStatement =
"begin" StatementList "end".

StatementList =
Statement { ";" Statement }.

Note that it is legal to write as many semicolons as you like.

If Statements

The if statement is a two way switch. If the expression is true, the first way is taken, else the second.

```
IfStatement =
  "if" expression "then" statement [ "else" statement ].
```

Note that it is NOT legal to put a semicolon before the else keyword.
Semicolon is not a statement, but a separator.

Examples:

```
if OldEnough then LookTV else GotoBed;
if Bad then else GoodEnough;
if not Bad then GoodEnough;
```

Case Statements

The case statement is a multi way switch.
The expression is compared against the case constants one by one until a match is found. Then the corresponding statement is executed. If no match is made, then either the else clause is executed or nothing at all if no else clause exists.

The expression and all the case constants must be of the same ordinal type.

```
CaseStatement =
  "case" expression "of"
  Case { ";" Case }
  ElseClause
  { ";" }
  "end".

Case =
  ConstantElement { "," ConstantElement } ":" Statement

ConstantElement =
  Constant [ ".." Constant ].
ElseClause =
  "else" StatementList.
```

Examples:

```
Case NextSymbol of
'+': Expression:=n+Factor;
'-': Expression:=n-Factor;
Else
  writeln('Parsing error');
  halt(99);
end;
```

```
case Month of
1,3,5,7,8,10,12: Days:=31;
2: if Leap(Year) then Days:=29 else Days:=28;
else Days:=30;
end;
```

For Statements

When you know how many times a statement (compound statement etc) has to be executed, you will use the "for" repetitive statement.

```
ForStatement =
  "for" ControlVariable ":=" InitialValue
  ( "to" | "downto" ) FinalValue "do" statement.

ControlVariable =
  VariableIdentifier.

InitialValue =
  Expression.

FinalValue =
  Expression.
```

The control variable must be a variable identifier either global or local in the block containing the for loop. The variable must not be qualified, so do not use array index, record fields or pointer variables here.

The control variable takes all values from the initial value to the final value including both. When using the "to" format the control variable is incremend by one each time the loop executes the statement.
When using "downto", the control variable is decremented. The statement will not be executed if the initial value is greater than the final value when using the "to" format opposite when using the "downto" format.

It is not legal to change the control variable inside the loop. This is not checked by the compiler. When the loop has finished the control variable will be undefined. If you jump out of the loop using a goto statement, the control variable is still valid (containing the last value used).

Examples:

```
For n:=1 to 30000 do ;  {Nothing}
For n:=1 to 9 do
  if Odd(n) then writeln(n,' is an odd number')
  else writeln(n,' is an even number');
```

Repeat Statements

The repeat statement executes some statements until an expression returns true. As the sequence of text lines show, the expression is evaluated after the statements are executed, thus the statements are always executed at least once.

```
RepeatStatement =
  "repeat" StatementList "until" Expression.
```

Examples:

```
repeat write('*') until Keypressed;
```

While Statements

The while loop works almost as the repeat loop except when the expression is evaluated. The while loop tests the expression before the statements are executed.

```
WhileStatement =
  "while" Expression "do" Statement.
```

Examples:

```
while not eof(F) do begin
  readln(F,S);
  writeln('Line read: ',S)
end;
```

With Statements

With statements open up a new scope in which all symbols are first looked for. The with statement also establishes a reference to each of these record variables. While the statement is executing the reference does not change, even if you change the variables that made up the reference.

```
WithStatement =
  "with" RecordVariableList "do" Statement.

RecordVariableList =
  RecordVarReference { "," RecordVarReference }.
```

Examples:

```
With SomeOther,Rect do begin
  x:=7; y:=n;
end;
```

This is equivalent to

```
With SomeOther do
  With Rect do begin
    x:=7; y:=n;
  end;
```

This is equivalent to

```
Rect.x:=7; Rect.y:=n;
```

Another example:

```
P:=HeaderField;
while P<>Nil do
  with P^ do begin
    a:=AField;       {Note a is the same as}
    P:=NextField;
    b:=AField;       {b, because of with P^ do}
  end;
```

Note that this is litterally equal to writing

```
P:=HeaderField;
while P<>Nil do begin
  a:=P^.AField;      {Now a is NOT equal}
  P:=NextField;
  b:=P^.AField;      {to b!!!}
end;
```

but the program will behave in quite another way.

Procedures and Functions

This section describes how to declare procedures and functions.

Procedures gets activated by a procedure statement while functions are activated when used in an expression that contains its function name.

Procedures and functions are defined as:

```
ProcedureDeclaration =
  ProcedureHeading ";" FuncProcBody ";".

FunctionDeclaration =
  FunctionHeading ";" FuncProcBody ";".

FuncProcBody =
  Block |
  "Forward" |
  "External" |
  "Inline" Constant { "," Constant }.
```

```
ProcedureHeading =
    "procedure" Identifier [ FormalParameterList ].

FunctionHeading =
  "function" Identifier [ FormalParameterList ]
  ":" ResultType.

ResultType =
  TypeIdentifier | "string".
```

If the procedure body is declared as a block, all at the declaration is done right away. Alternatively, it may be declared as a forward declaration or as an external declaration. Another way is to code a procedure as an inline declaration.

Examples:

```
procedure HexDump(Val,Format: Integer); forward;
Function Power10(N: Integer): Real;
procedure HexDump;
function Max(a,b: Integer): Integer; inline $xxxx,$xxxx;
```

Forward

A procedure that has the directive "forward" instead of the block is a forward declaration. Later, in the same declaration part, the procedure gets defined with its real block with a defining declaration. In a defining declaration, the parameter list (and a function result type) is not repeated. In HighSpeed Pascal it is allowed to repeat the parameter list (and a function result type), but it is NOT checked and NOT used for anything, only as a comment.

Procedures named in the interface part of a unit work like forward declarations, but the keyword forward is not used.

Examples:

```
function Func(N: Integer): Integer; forward;

procedure Proc;
begin
  if Func(7)=3 then {};
end;

function Func;
begin
  Func:=N*2
end;
```

External

A procedure that has the directive "external" instead of the block is a external declaration. The actual code block must be linked into the program by using a {$L MyCode.obj} directive somewhere in the source for this declaration.

HighSpeed Pascal cannot check that the parameters passed are in the same order as the assembly code assumes.

Examples:

```
procedure BlockMove(var Src, Dst; Count: Integer); external;
```

Inline

A normal procedure call is done internally by making a JSR to the procedure. The inline directive permits you to replace this JSR with a sequence of instructions entered as constants after the inline directive.

Note that each procedure "call" using inline inserts all constants listed, therefore keep them short.

HighSpeed Pascal cannot check that the parameters passed are in the same order as the inline code assumes.

A forward or interface procedure may not be declared inline later.

Se "Inside HighSpeed Pascal" for further explanations.

Parameters

The declaration of a procedure (or function) can specify a formal parameter list. Each parameter in the list is local to the procedure in the same way as normal local variabels declared in a procedure. The only difference is that they are initialized when the procedure is activated, and that they might be referencing other variables which are changed whenever the formal parameter is changed (var parameters).

FormalParameterList =
"(" ParameterDeclaration { ";" ParameterDeclaration ")".

ParameterDeclaration =
IdentifierList ":" ParameterType |
"var" IdentifierList ":" ParameterType |
"var" IdentifierList.

ParameterType =
TypeIdentifier | "string".

There are three kinds of parameters: value, variable and untyped variable.

Value Parameters

Value parameters are not preceded with "var". They act like normal local variables. They are just initialized when the procedure is initialized. The formal parameter can be changed without affecting the actual parameter used.

The actual parameter must be an expression that does not contain any file type elements. File types can only be passed as variable parameters.
If the parameter type is string, the formal parameter has a type of String[255].
The actual parameter must be assignment compatible with the formal parameter.

Variable Parameters

A variable parameter is a parameter declaration preceded by "var" and followed by a type.
A formal variable parameter is located right on top of it's actual parameter, so each time the formal parameter gets changed, the actual parameter is also changed.
If the parameter type is string, the formal parameter has a type of String[255], and the actual parameter must also have a length of 255.
The type of the actual parameter and the type of the formal parameter must be identical.

Untyped Variable Parameters

A variable parameter is a parameter declaration preceded by "var" but without any type following it.
The actual parameter passed can be of any type. The formal parameter are typeless. They can only be used as typeless pointers or by variable type castings like in this example:

```
function Equal(var Src1, Src2; Length: Integer): Boolean;
type
  Bytes= array [1..MaxInt] of ShortInt;
var
  n: Integer;
begin
  Equal:=False;
  for n:=1 to Size do
    if Bytes(Src1)[n]<>Bytes(Src2)[n] then Exit;
  Equal:=True
end;

var
  TestVar,TestVar2: array[1..99] of LongInt;
  B:= Boolean;
  B:=Equal(TestVar,TestVar2,SizeOf(TestVar));
  B:=Equal(TestVar[10],TestVar2[20],SizeOf(LongInt));
  B:=Equal(TestVar[10],TestVar2[20],SizeOf(LongInt)*10);
```

Be careful when using untyped variables. It is almost as calling a procedure in the C language.

Procedure
A procedure is activated in a procedure statement like:

```
DumpSpaced(123).
```

Below is shown what a normal procedure might look like:

```
procedure DumpSpaced(N: Longint);
var
  S: String[9];
begin
  Str(N,S);
  for N:=1 to Length(S) do
    write(S[N],' ');
  writeln;
end.
```

It re-uses N, without destroying the actual parameter. It does not return anything.

Function
A function is activated when its name appears in a function call in an expression like:
`R:=100*Factorial(10)+1.`

A normal function may look like this:

```
Function Factorial(N: Integer): Real;
begin
  if N<=1 then
    Factorial:=1
  else
    Factorial:=Factorial(N-1)*N
end;
```

A function must return a value. If it does not, you get a garbage value.
Return values are assigned to the function when the function name appears on the left side of an assignment (:=). If the function name appears any other place, it is treated as another (recursive) call to the function itself.
You cannot read the value back from the function name once it is assigned, you will have to keep a copy yourself.
You may assign a return value as many times as you like.

Programs, Units , Procedures and functions.

Blocks

All code fragments used to form a Pascal program has a name. Every one of these pieces are collected into blocks. These can be procedures, functions, a program or a unit. In each block, new identifiers can be declared, which are local to the block. The only exceptions are units, declaring identifiers for use by others.

```
Block           = DeclarationPart StatementPart.
DeclarationPart = { LabelDeclarationPart |
                    ConstantDeclarationPart |
                    TypeDeclarationPart |
                    VariableDeclarationPart |
                    ProcedureDeclarationPart |
                    FunctionDeclarationPart }.

LabelDeclarationPart     = "label" Label [ "," Label ] ";"
Label                    = Identifier | DigitSequence

ConstantDeclarationPart  = "const" ConstantDeclaration
                                { ConstantDeclaration }.

TypeDeclarationPart      = "type" TypeDeclaration
                                { TypeDeclaration }.

VariableDeclarationPart  = "var" VariableDeclaration
                              { VariabelDeclaration }.

ProcedureDeclarationPart = ProcedureDeclaration.

FunctionDeclarationPart  = FunctionDeclaration.
```

Program Syntax

A program is similar to a procedure, except for the first and last line. The heading is different and the last line reads "end." instead of "end;".

```
Program        = [ ProgramHeading ] [ UsesClause ] Block ".".
ProgramHeading = "program" Identifier [ ProgramParameterList ].
ProgramParameterList = "(" IdentifierList ")".
UsesClause           = [ "uses" IdentifierList ].
```

The identifier after "program" names the program.

Unit Syntax

Units are used when making modular programs. Each unit gets a name which must be the same as the name of the source file (extended with .PAS).

```
Unit                = "unit" Identifier ";"
                      InterfacePart
                      ImplementationPart
                      InitializationPart "."
InterfacePart       = "interface" UsesClause
                      { ConstantDeclarationPart |
                        TypeDeclarationPart |
                        VariableDeclarationPart |
                        ProcedureHeadingPart |
                        FunctionHeadingPart }.
ProcedureHeadingPart = ProcedureHeading ";" [InlineDirective ";" ].
FunctionHeadingPart  = FunctionHeading ";" [InlineDirective ";" ].

ImplementationPart  = "implementation"
                      { LabelDeclarationPart |
                        ConstantDeclarationPart |
                        TypeDeclarationPart |
                        VariableDeclarationPart |
                        ProcedureDeclarationPart |
                        FunctionDeclarationPart }.
InitializationPart = "end" | StatementPart.
```

There is no label declaration in the interface part.
The implementation part must implement all the procedures and functions defined in the interface part.
Anything not defined in the interface part is private to the unit.

The code entered in the initialization part will be executed right before the first begin in the program module. This initialization part will be in the same order as used.

Scope and Activations

The scope rules determine where an identifier can be used.
- An identifier cannot be used, before it has been defined. "Before" means reading the source text from top to bottom.
- An identifier cannot be used after the block in which it is defined ends.
- An identifier cannot be redefined in the same block.
- An identifier can be redefined in an enclosed block.
- An identifier can be reused in a record structure.

This does not hide the first identifier from been used.

Scope for Units

Each unit used in a program or unit opens up a new scope. An identifier can be redefined in each of these units, and only the last seen will be used, except if it is referenced with a qualified name as Unit3.N:=3;.

Scope for Standard Identifiers

All identifiers defined by HighSpeed Pascal are defined in the System units. These units are always the first ones used. As the normal scope rule for a unit says, all identifiers can therefore be redefined.

Activations

When a procedure is called, it is activated. Much can be said about what an activation contains, but the most interesting is where the local variables are.
Each local variable gets new space each time a procedure is called.
Each time a procedure calls itself (recursive or mutually recursive) it gets new space for the local variables. Each of the activations naturally knows where each of it's variables for its own activation are.

Input and Output

All input and output routines in HighSpeed Pascal are described here. They are all predeclared. Some of them takes a variable number of parameters, thus they cannot be created using normal Pascal style programming.

Three types of files exist in a HighSpeed Pascal program; text, typed and untyped files. They are declared as:

```
aTextFile:     Text;
aTypedFile:    File of SomeThing;
anUntypedFile: File;
```

All files used must have a name assigned. The name is the same as the name of the file on the disk. Some text files are treated differently by not using the disk, and has the information stream going somewhere else like to the screen or to a printer.

All files must be opened before use, giving them a name at the same time. After use they must be closed again. Text files can be opened for re-use (only input or appending output) or they can be created each time used (only output) thereby first deleting the one with the same name.
Other files are always both input and output.

Text files are always accessed sequentially, you cannot seek around in text files. In other files you can position your current read/write position by using the seek procedure.

After all calls to the input/output system an I/O-check routine is called to see if anything went wrong in the last routine. This can be disabled using the directive {$I-} or by disabling the $I flag in the OPTIONS/Compiler dialogue box.

Routines for Input and Output

These are the routines that can be used for file I/O:

```
Reset, Rewrite, Append, Close,
Seek, FilePos, FileSize,
Eof, Eoln, SeekEof, SeekEoln,
Read, Readln, Write, WriteLn, Page, BlockRead, BlockWrite,
Rename, Erase,
IOResult.
```

General control routines

These are the routines used for creating, erasing, renaming and opening files:

```
Reset, Rewrite, Append, Close, Rename, Erase, IOResult.
```

Reset (F, Title)
Open the file named Title. The file is read only if it is a text file.

Reset (F, Title, BufSize)
A text file may optionally specify the size of a buffer in which the input is buffered if not used right away.

Reset (F)
Rewind the file to the start of the file. May only be used on an already opened file.

Rewrite(F, Title)
Creates and opens the file named Title. The file is write only if it is a text file.

Rewrite(F, Title, BufSize)
A text file may optionally specify the size of a buffer in which the output is buffered until full.

Rewrite(F)
Rewind the file to the start of the file. May only be used on an already opened file. The contents should have been deleted, but the Atari cannot truncate a file. To do this you will have to erase the file yourself. HighSpeed Pascal does not know the filename, and cannot do this for you.

Append(F, Title)
Only for textfiles. Opens a file like rewrite, but starts writing from the last position used in the file.

Append(F, Title, BufSize)
Specify a buffersize like rewrite.

Append(F)
Start appending on an already opened file.

Close(F)
Flush unwritten data and close the file.

Rename(OldName, NewName)
Rename a file from oldName to newName. Both parameters are string types.

Erase(Title)
Delete the file from the disk.

IOResult: Integer
Return the result of last input or output operation. Zero is returned when no errors has occurred. IOResult is cleared by this call.

Typed and Untyped Files
Untyped files are treated as a typed file with a type of: file of ShortInt.
These are the routines to use on typed and untyped files:
`Seek, FilePos, FileSize, Eof, Read, Write, BlockRead, BlockWrite.`

Seek(F, N: Longint)
Position the current read and write position. The first position is number zero.

FilePos(F): Longint
Return the current file position. FilePos returns zero after reset and rewrite.

FileSize(F): Longint
Return the number of elements in the file F.

Eof(F): Boolean
Return false if there is still data to read after the current file position.

Read(F, v { ,vn })
Read a file component into a variable. The variable(s) v must be of same type as the component type of F. The current file position is advanced one for each variable.

Write(F, v { ,vn })
Write a file component from a variable. The variable(s) v must be of same type as the component type of F. The current file position is advanced one position for each variable.

BlockRead (F, Buffer, Count)
Read Count bytes from file F into Buffer. An IOError is generated if Count bytes could not be read.

BlockRead (F, Buffer, Count, ActualCount)
Read Count bytes from file F into Buffer. The actual count read is returned in ActualCount. No errors are generated.

BlockWrite (F, Buffer, Count)
Write Count bytes to file F from Buffer. An IOError is generated if Count bytes could not be Write.

BlockWrite (F, Buffer, Count, ActualCount)
Write Count bytes to file F from Buffer. The actual count written is returned in ActualCount. No errors are generated.
BlockRead and BlockWrite can be used on both typed and untyped files. Untyped files are treated as a typed file with a type of: file of ShortInt. The ShortInt size tells reset, rewrite and seek that the element size is one byte. BlockRead and BlockWrite takes it's size as byte count. To read a LongInt from a file of LongInt you will have to use BlockRead(F,B,1*SizeOf(LongInt)).

Text Files
These are the routines for text file I/O.
`Write, WriteLn, Page, Readln, Read Eof, Eoln, SeekEof SeekEoln.`
For all text file routines, you do not have to specify a file variable. It is then assumed to be either the standard file Input or Output. Write and WriteLn takes Output as there default file, while all the other takes Input as there default.

Read (F, v { , Vn })
The variable V can be any one of these types: Integer, Real, Char, String.
When reading Integer or Reals, HighSpeed Pascal expects to read a sequence of digits and special symbols like ., +, -, e, or E. Any white spaces (tab, space and end-of-line) are skipped first.
When reading to a Char variable, one byte is read from the file. In standard Pascal the end-of-line symbol is read as a space, chr(32), but in HighSpeed Pascal, you will read it as a carriage return, chr(13).
Reading a string will read until end of line is met. As many characters as the length of the string allows will then be put into the string. Reading a string will never read past a newline. If that is wanted, then use ReadLn.

```
Readln (F, v { , Vn } )
Write (F, p { , Pn } )
WriteLn (F, p { , Pn } )
```

The variable P can be any one of these types: Integer, Real, Char, String, PackedString, Boolean.

```
WriteLn(F,p1,p2,p3);
```

is the same as:

```
Write(F,p1,p2,p3);
WriteLn(F);
```

is the same as:

```
Write(F,p1);
Write(F,p2);
Write(F,p3);
WriteLn(F);
```

The write parameters p, P1, P2.. Pn has the form:

```
Expression [ [ ":" MinWidth ] ":" DecPlaces ].
```

where Expression is the wanted output, and MinWidth and DecPlaces are integer expressions.

MinWidth positions are used when writing the expression. If the expression needs more space, more space is used. Default for MinWidth is 0.
DecPlaces specifies how many decimal digits a real number should have when converted to text. As shown, it can only be specified if MinWidth is also specified.

Integer, Char, String and PackedString type variables are written left justified in a field with width MinWidth. If MinWidth is too small, no spaces are placed in front.
Boolean type variables are written by either writing the string FALSE or TRUE.
Real type variables are a bit more complicated. They are first converted to text format using the MinWidth and DecPlaces value.
If MinWidth is omitted, it is assumed to be 10.

If DecPlaces is omitted, the number is converted to a floating point decimal string, or else it is converted to a fixed point decimal string.

Floating point decimal string:

```
( " " | "-" ) digit "." decimals "e" ( "+" | "-" ) exponent.
```

Fixed point decimal string:

```
[ blanks ] "-" digits "." decimals.
```

Eof (F): Boolean
Return false if there is data still to read after the current file position.

Eoln (F): Boolean
Return false if there is data still to read after the current file position, but before the next newline.

SeekEof (F): Boolean
As Eof, but first it skips all white spaces.

SeekEoln (F): Boolean
As Eoln, but first it skips all white spaces except newline.

Page (F)
Write a form feed to the file. Same as Write(chr(12)).

Error handling

After all calls to the input/output system an I/O-check routine is called to see if anything went wrong in the last routine. This can be disabled using the directive {$I-} or by disabling the $I flag in the dialog box. If IO-checking is disabled, the result of each operation can be checked by using the IOResult function. If it returns zero, no error did occur. Note that IOResult is a function that only returns its value once, the next time it returns zero. Be sure to "empty" the IOResult after eac

Text File Devices

Whenever a text file is about to be opened, a chain of known "files" is searched. If you try to open one of these files, you will get access to an internal handler, that decides what to do about all your input and output calls. The Input and Output files goes this way as default. You can naturally overwrite this, just by doing a new reset and rewrite on them.

Units

The DOS unit.

The DOS unit implements a variety of operating system and disk relatedroutines.

Constants, Types and Variables.

The File Attribute Constants

```
Const
        ReadOnly    = $01;
        Hidden      = $02;
        Sysfile     = $04;
        VolumeID    = $08;
        Directory   = $10;
        Archive     = $20;
        AnyFile     = $3F;
```

The file attribute constants are used in routines that involve Getting, Setting and Testing file attributes. They are also used when searching indirectories for specific file groups using the FindFirst and FindNextprocedures.

The SearchRec Type

```
Type
        SearchRec  = RECORD
                       Reserved : Packed  Array[0..20] OF Byte;
                       Attr     : Byte;
                       Time     : LongInt;
                       Size     : LongInt;
                       Name     : String[12];
                     END;
```

The SearchRec type is used when searching for files with the FindFirst and FindNext procedures.

The DateTime Type

```
Type
        DateTime    = RECORD
                        Year    : Integer;
                        Month   : Integer;
                        Day     : Integer;
                        Hour    : Integer;
                        Min     : Integer;
                        Sec     : Integer;
                      END;
```

The DateTime type is used in connection with several of the routines in the DOS unit. It finds primary use in the routines that Gets and Sets the time and date in the operating system, but it is also used in connection with file related operations such as FindFirst and FindNext.

File-Handling String Types

```
Type
        ComStr      = String[127];
        PathStr     = String[79];
        DirStr      = String[67];
        NameStr     = String[8];
        ExtStr      = String[4];
```

The file-handling string types are used in the routines that deals with paths and file names. Examples of such routines are ChDir, RmDir, MkDir and FExpand.

The DosError Variable

```
Var
        DosError : Integer;
```

The values stored in the DosError variable are DOS errors. The following is the DosError codes as they are declared in the DOS unit:

```
Const
        EINVFN    = -32;    { Invalid Function Number          }
        EFILNF    = -33;    { File Not Found                   }
        EPTHNF    = -34;    { Path Not Found                   }
        ENHNDL    = -35;    { File Handle Pool Exhausted       }
        EACCDN    = -36;    { Access Denied                    }
        EIHNDL    = -37;    { Invalid Handle                   }
        ENSMEM    = -39;    { Insufficient Memory              }
        EIMBA     = -40;    { Invalid Memory Block Addr        }
        EDRIVE    = -46;    { Invalid Drive Specification      }
        ENSAME    = -48;
        ENMFIL    = -49;    { No More Files                    }
        ERANGE    = -64;    { Range Error                      }
        EINTRN    = -65;    { GEMDOS Internal Error            }
        EPLFMT    = -66;    { Invalid Executable File format   }
        EGSBF     = -67;    { Memory Block Growth Failure      }
```

A value of zero indicates that no error occoured.

Environment functions

EnvCount	Returns the number of strings in the environment block.
EnvStr	Returns the environment string number N.
GetEnv	Returns the value of a specific environment string.

Parameter functions

ParamCount	Returns the number of parameters passed on the command line.
ParamStr	Returns command line parameter number N.

Date and Time procedures

GetDate	Gets the current date in the operating system.
SetDate	Sets the current date in the operating system.
GetTime	Gets the current time in the operating system.
SetTime	Sets the current time in the operating system.
SetFtime	Sets the time and date a file was last updated.
GetFtime	Gets the time and date a file was last updated.
Packtime	Packs a DateTime variable into a 32-bit long integer variable.
UnpackTime	Unpacks a 32-bit long integer to a DateTime variable.

Disk status procedures and functions

DiskSize	Returns the size of a disk in a specified disk drive.
DiskFree	Returns the number of free bytes on a diskin a specified disk drive.
SetDrive	Sets the current disk drive.
GetDrive	Returns the number of the current disk drive.
GetVerify	Returns the value of the disk verify flag.
SetVerify	Sets the value of the disk verify flag.

Directory-handling procedures

GetDir	Gets the current directory on a specified drive.
ChDir	Changes the current directory.
MkDir	Makes a new directory.
RmDir	Removes a directory.
FSplit	Splits a path up into the Directory name,File name and File extension.
FExpand	Takes a path and returns a fully qualified path consisting of Drive,Directory name and File name.

File-handling procedures and functions

FindFirst	Searches a directory in order to find the first file name matching a specified group of files.
FindNext	Searches for the next file name matching the specifications given in the call to FindFirst.
SetFAttr	Sets the attributes of a specified file.
GetFattr	Gets the attributes of a specified file.

Miscellaneous functions

TosVersion	Returns the version of the operating system.
Super	Switches the CPU between Supervisor and User mode.

The BIOS unit.

The BIOS unit implements routines both from the standard BIOS and the extended BIOS.
Since this manual is not meant to be a guide to the Atari BIOS, we strongly recommend that you read a specific BIOS manual, if you are uncertain about what a specific BIOS routine does.

Disk related procedures and functions

FlopRd Reads one or more sectors from a floppy drive.

FlopWr Writes one or more sectors to a floppy drive.

FlopVer Verifies one or more sectors on a floppy drive.

FlopFmt Formats a track on a floppy drive.

Rwabs Reads one or more logical sectors from a disk drive.

ProTobt Creates a prototype of a boot sector.

DrvMap Returns information about the number of drives attatched to the system.

MediaCh Checks to see if a media (disk) has been changed.

GetBpb Returns the BIOS parameter block for a specified drive.

Screen related procedures and functions

PhysBase Returns the address of the physical screen memory.

LogBase Returns the address of the logical screen memory.

GetRez Returns information about the current screen resolution.

SetScreen Sets the Screen resolution and the physical and logical screen memory address.

ScrDmp Dumps the screen to the printer.

Vsync Waits for the next vertical-blank interrupt.

PrtBlk

SetPalette Sets the contents of a hardware palette register.

SetColor Sets a palette entry to a given color.

Parallel and Serial port procedures and functions

RsConf Configures the serial port.

SetPrt Gets or sets the printer configuration byte.

IoRec Returns a pointer to a serial devices input buffer record.

Keyboard and Cursor procedures and functions

CursConf Sets the state of the cursor.

KbShift Returns the status of the Control, Caps and Shift keys.

KeyTbl Sets pointers to the keyboard translation tables.

BiosKeys Restores the default BIOS translation tables.

IKbdWs Writes a sequence of characters to the keyboard.

KbdvBase

KbRate Sets the keyboard repeat rate.

Sound procedures and functions

Giaccess Reads or writes a register on the sound chip.

OffGiBit Sets a bit in the PORT A register to zero.

OnGiBit Sets a bit in the PORT A register to one.

MidiWs Writes a sequence of characters to the midi port.

DoSound Executes a block of sound chip commands.

Interrupt procedures and functions

jDisInt Disables an interrupt.

jEnabInt Enables an interrupt.

MfpInt Sets an interrupt vector.

XbTimer

IO procedures and functions

BConStat Returns the status of a character device.

BConIn Reads a character from a device.

BConOut Writes a character to a device.

BCoStat Returns the character output status for a device.

Miscellaneous procedures and functions

X_Random Returns a random number.

XSetTime Sets the time and date in the keyboard.

XGetTime Gets the time and date in the keyboard.

TickCal Returns a system-timer calibration value.

SupExec Executes a fragment of code in Supervisor mode.

PuntAES Throws the AES away.

SsBrk Reserves an ammount of memory.

GetMpb

SetExc Sets an interrupt vector.

InitMous Initializes the mouse packet handler.

The System unit.

Types and Variables

The Byte Type

```
Type
        Byte        =  0..255;
```

The byte type is a subrange specifying the legal range of variables of type byte.

The Word Type

```
Type
        Word     = 0..65535;
```

The word type is a subrange specifying the legal range of variables of type word.

The RBasePage Type

```
Type
        PtrLen      = RECORD
                        Base  : Pointer;
                        Size  : Longint;
                      END;

        PBasePage   = ^RBasePage;
        RBasePage   = RECORD
                        P_lowtpa    : Pointer;
                        P_hitpa     : Pointer;
                        Text        : PtrLen;
                        Data        : PtrLen;
                        BSS         : PtrLen;
                        P_dta       : Pointer;
                        P_parent    : PBasePage;
                        P_resrvd0   : Longint;
                        P_env       : Pointer;
                        P_stdfh     : Array[1..6] of ShortInt;
                        P_resrvd1   : ShortInt;
                        P_curdrv    : ShortInt;
                        P_resrvd2   : Array[1..18] of LongInt;
                        P_cmdlin    : String[127];
                      END;
```

The RBasePage type is used as a type for the BasePage variable.

The BasePage Variable

```
Var
        BasePage    : PBasePage;
```

The BasePage is used as a control block for your compiled programs. One of its functions is to provide the command line that started the program.
The command line can be accesssed through the P_cmdlin variable in the BasePage record.

The Input and OutPut Variables

```
Var
        Input,
        Output      : Text;
```

The Input and OutPut variables are used as the standard input and output devices.

The Stack Variables

```
Var
        HighStak    : Pointer;
        LowStack    : Pointer;
```

The HighStack variable points to the top of the stack. The LowStack variable points to the bottom of the stack.

The HeapOrg Variable

```
Var
        HeapOrg     : Pointer;
```

The HeapOrg points to the bottom of the heap.

The RandSeed Variable

```
Var
    RandSeed  : LongInt;
```

The RandSeed variable acts as a seed for the random-number generator.

The ExitProc Variable

```
Var
        ExitProc    : Pointer;
```

The ExitProc variable points to a chain of routines which is executed when a program terminates.

The ErrorAdr Variable

```
Var
        ErrorAdr    : Pointer;
```

The ErrorAdr contains the offset from the BasePage of the instruction that caused a program to terminate with a runtime error.

The ExitCode Variable

```
Var
        ExitCode    : Integer;
```

The ExitCode contains the number of the runtime error that caused a program to terminate.

The Trap Variables

```
Var
    STrap5    : Pointer;
    STrap102  : Pointer;
```

The two STrapXX pointers holds the addresses of the interrupt vectors XX as they were when the program was loaded.

The IOResVar Variable

```
Var
        IOResVar    : Integer;
```

The IOResVar is used by the IOresult function.

Screen procedures

ClrScr Clears the screen.

ClrEos Clears the screen, starting at the current cursor location.

ClrEol Clears the line, starting at the current cursor location.

InsLine Inserts a new line at the current cursor position.

DelLine Deletes the current line.

GotoXY Positions the cursor at specifiedcoordinates.

Miscellaneous functions

RunFromMemory

Returns true if a program is runnig from within the editor.

The AppFlag Variable

```
Var
        AppFlag     : Boolean;
```

The AppFlag will return true if a program is a standard application (x.PRG), otherwise it will return false.

The DevChain Variable

```
Var
        DevChain    : Pointer;
```

The DevChain variable points to a list of known devices.

```
Type
        Char32K    = Packed Array[0..32000] of Char;
        Char32KPtr = ^Char32K;
        FileRec    = RECORD
                       fInpFlag   : Boolean;
                       fOutFlag   : Boolean;
                       fHandle    : Integer;
                       fBufSize   : Integer;
                       fBufPos    : Integer;
                       fBufEnd    : Integer;
                       fBuffer    : Char32KPtr;
                       fInOutProc : Pointer;
                     END;
```

The FileRec type is used as a file control block for all types of files (Untyped, Typed and Text).

The LastPC Variable

```
Var
        LastPC    : LongInt;
```

The LastPC variable holds the address of the last known location in memory. Set by range and stack checking routines.

The ShftShft Variable

```
Var
        ShftShft  : Pointer;
```

The ShftShft (ShiftShift) pointer points to the current state of the shift keys. Is used when checking for the Shift-Shift user break combination.

The system 2 unit

Types

The FileRec Type

```
Type
        Char32K    = Packed Array[0..32000] of Char;
        Char32KPtr = ^Char32K;
        FileRec    = RECORD
                       fInpFlag   : Boolean;
                       fOutFlag   : Boolean;
                       fHandle    : Integer;
                       fBufSize   : Integer;
                       fBufPos    : Integer;
                       fBufEnd    : Integer;
                       fBuffer    : Char32KPtr;
                       fInOutProc : Pointer;
                     END;
```

The FileRec type is used as a file control block for all types of files (Untyped, Typed and Text).

Keyboard functions

KeyPressed The KeyPressed function returns true if a key has been pressed, but not yet read.

ReadKey Reads and returns a character from the keyboard.

Screen procedures

ClrScr Clears the screen.

ClrEos Clears the screen, starting at the current cursor location.

ClrEol Clears the line, starting at the current cursor location.

InsLine Inserts a new line at the current cursor position.

DelLine Deletes the current line.

GotoXY Positions the cursor at specified coordinates.

The Printer unit.

The printer unit implements an easy way of writing data to the printer connected to the PRN: port.

Outputting data to the printer is done by writing to the text fileby the name of Lst.

An example:

```
Program Print;

Uses Printer;

Begin
  Writeln(Lst,'Hello printer.');
End.
```

Variables

The Lst Variable

```
Var
        Lst      : Text;
```

The Lst file variable is used when writing data to the printer.

GEM

Purpose of this section
This section is ment to give the reader a short introduction to GEM and the terms connected. It is, however, not the purpose to teach the reader how to make a GEM program, as it lies outside the frame of this manual.

The GEM section is separeted into five sections:
GEM: An overview
THE GEM DEMO
GEMDecl (Reference)
GEMVdi (Reference)
GEMAes (Reference)

GEM : An overview

GEM is an abbreviation for „Graphics Environment Manager" i.e. it is supposed to create and maintain a graphic environment consisting of the following basic items :

- the mouse,
- the keyboard,
- the screen, and to a limited extent,
- file handling (resource files only)

As it is assumed that you're already familiar with the basic operations of GEM (e.g. the desktop), we shall be concentrating on how to use the multitude of procedures, functions and data structures that GEM presents to the programmer via the HighSpeed Pascal libraries GemDecl, GemAES and GemVDI.

The internal operating system construction of importance to us on the ST is as follows

```
  GEM AES          highest level
  GEM VDI
( GEMDOS/TOS )
( XBIOS/BIOS )
```

But before delving into the depths of the AES and VDI, let's get to terms with some abbreviations :

```
AES   = Applications Environment Services
VDI   = Virtual Device Interface
OS    = Operating System
DOS   = Disk OS; TOS = Tramiel (?) OS...
GDOS  = Graphic Device OS
BIOS  = Basic Input Output System
XBIOS = eXtended BIOS
```

The basic idea behind this OS construction is as follows : Let the BIOSes handle all machine-dependent operations concerning screen input/output, keyboard I/O, mouse I/O, disk I/O etc. and let the GEMDOS/TOS handle disk operations on a more general level, that is, using the GEMDOS instead of the BIOS, makes it much more simple to port a program from, say, an IBM PC to the ST. And that leads us to the simple consequence : The construction of a general graphics-based operating system : The GEM.

The purpose of the VDI

The VDI contains a number of lower-level procedures and functions which can be found in the GemVDI library. They present „simple" graphics operations such as drawing a line, setting a point, defining colors etc. together with more complex operations like opening „workstations" and handling mouse movements.

The purpose of the AES

The AES presents more complex operations based on simple VDI operations. Among these are : Event handling, menu control, dialog boxes etc. It enables the programmer to disregard almost all hardware considerations and is so the ideal environment for creating easily portable programs.

Basic terms

Before a GEM program can use the ST's hardware, it must create a workstation." A workstation is the logical implementation of the physical environment on the ST - that is, the programmer can - in principle - make GEM ignore e.g. that on a monochrome monitor, the raster has a size of 640 * 400, and instead treat it as if it were 32678 * 32768 pixels wide. The „workstation" concept also implies that the computer does not necessarily have to use the screen for output - it could just as well write the output to a disk, plotter etc.

THE GEM DEMO

In this section we shall present possible solutions to common problems concerning GEM programming. We do not claim out methods to be the best available, and they should only be used for inspiration - after all, one only gets the feel of things when he's forced to think it over. So the moral is : Take a look at it, use it if you're learning GEM or otherwise are having problems - but don't imitate uncritically!

As discussing hypothetical problems tends to get rather confusing, we have chosen to discuss a "real-life" problem : The creation of a GEM demo program... Ah, well, not so "real-life" perhaps, but it allows us to demonstrate a wide number of problems and their possible solutions. Therefore we shall imitate the steps of conceptual design, and discuss the HighSpeed Pascal implementation.

NOTE : Before reading on, it's recommended you "play" a bit with the demo program (GEMDEMO.PRG) which is supplied with the HighSpeed Pascal system - just to get an overview of what the program does.

Purpose

The purpose of the GEM demo program is to demonstrate the following subjects :

- application initialization
- resource handling
- menu handling
- simple dialog box handling
- simple window handling (no scrolling)
- simple graphics and text output
- simple event handling

What the demo program shall do :

- handle two windows : A text window and a graphics window.
- draw text in the text window and one of four graphic demos in the graphics window

As constructing a resource data structure via the demo program is a rather extensive task, we selected to create it with a Resource Construction Set package which generated the two files GEMDE-MO.RSC and GEMDEMO.H. The contents of these files will be examined later. GEMDEMO.H contains a list of constants, describing the object trees and elements, and is included into the DemoInterface (see below) unit's INTERFACE section, thereby making them available to all client units.

Unit division

The above list of subjects to be covered inspired the following unit division :

```
DemoGraphs     - takes care of drawing the desired graphics
DemoMenu       - menu handler/dispatcher
DemoWindows    - text/graphics window handler
DemoInterface  - resource load, setup, GEM init., exit, object
                 manipulation, global variables, constants etc.
GEMDemo        - main program
```

Resource contents

In the below, we shall examine the contents of the resource file, i.e. the object's internal structure and interaction. To aid the understanding, the same names are used for describing the objects as the ones in the GEMDEMO.H file.

The file GEMDEMO.RSC contains three object trees :

```
MAINMENU - the main menu
ABOUTBOX - the information box opened by the About... menu item
SLCTDEMO - dialog box for selecting one of four graphic demos
```

The MAINMENU tree contains three title entries with the subordinates shown below

Title entry :	**DESKMENU**	**WINMENU**	**QUITMENU**
Subordinates :	ABOUTBAR	WINCLOSE	QUITBAR
		TXTOPEN	
		TXTCLOSE	
		GRAOPEN	
		GRACLOSE	
		GRASELEC	

All elements of the WINMENU except GRASELEC have their enable/disable state changed by the program in order to prevent the user from making illegal operations. The pattern is this :

Chosen	**Enabled**	**Disabled**
WINCLOSE	TXTOPEN (*)	WINCLOSE (*)
	TXTCLOSE (*)	TXTCLOSE (*)
		GRACLOSE

TXTOPEN	WINCLOSE TXTCLOSE	TXTOPEN
TXTCLOSE	TXTOPEN	TXTCLOSE WINCLOSE (*)
GRAOPEN	WINCLOSE GRACLOSE	GRAOPEN
GRACLOSE	GRAOPEN	GRACLOSE WINCLOSE (*)

The (*) mark indicates, that WINCLOSE is only enabled if either or both of TXTCLOSE and GRACLOSE are enabled, and that it is the currently active of the text and graphics window which is closed.

The ABOUTBOX contains 1 element only, ABOUTOK, which designates the[Ok] key with the following flag layout : Selectable, Default, Exit.

Finally, the SLCTDEMO dialog box contains four icons, each representing a specific demo type. They are laid out as radio buttons, each one's flag state is : RadioButton, Selectable, Exit. The icons are :

```
BOXES    - draws a number of filled boxes
LINES    - draws a line pattern
ELLIPSES - draws a number of filled ellipses
PIES     - draws a number of filled pie slices
```

Initializing the application

In order to make GEM familiar with our demo program, we have to pass some information to the AES and VDI. In return, we get an AES handle, which we won't use for anything, and a VDI handle, which we will be using when drawing the demos etc. The steps involved in initializing GEM are :

1. Initialize the application and get the AES handle in return
2. Get the VDI's graphics handle
3. Choose how we want the graphics to behave
4. Open a virtual workstation and pass the VDI handle to it
5. From the workstation return parameters, determine actual workstation appearance.
6. If running on a color monitor then abort!

In case GEM returns a negative AES handle, we have no other option than to abort the program with a "Fatal error" message.

The above steps are implemented in DemoInterface.Init_GEM :

```
Function Init_Gem;
VAR
  workin  : IntIn_Array;   { workstation input parameters  }
  workout : workout_Array; { workstation output parameters }
  dummy   : Integer;
BEGIN
  AES_handle := appl_init;          { get AES handle }
  IF AES_handle >= 0 THEN BEGIN     { if legal then go on : }

    { get VDI handle and height of a character box : }
    VDI_handle := graf_handle(dummy, dummy,
                              CharBoxHeight, dummy);

    { initialize workstation input parameters : }
    FOR dummy := 0 TO 9 DO workin[dummy] := 1;
    workin[10] := 2;    { use Raster Coordinates }

    v_opnvwk(workin, VDI_handle, workout); { open workstation }
    IF workout[39] > 2 THEN           { Oh no! A color monitor! }
      FatalError('GEMDEMO only runs on a | monochrome monitor');

    { get size of free desktop area : }
    wind_get(0, WF_FULLXYWH, MinX, MinY, MaxW, MaxH);
    graf_growbox(0, 0, 0, 0, MinX, MinY, MaxW, MaxH)    { nice! }
  END;
  Init_Gem := AES_handle >= 0       { return TRUE if init ok }
END;
```

Notes : The v_opnvwk procedure passes the size of the screen in pixels, but as this doesn't take the menu bar into account, wind_get is used instead with a window handle (first parameter) of zero, which designates the desktop. A WF_FULLXYWH will then return the size of the available desktop space. The graf_handle returns four parameters describing graphic text character and box widths and heights, but only one of these, the height of a character box, is stored here, as it's used in the text window redraw

Handling the resources

After successful initialization of the GEM environment, we turn to the resource file. The steps involved are :

1. Load the resource file
2. If it couldn't be found then tell the user so and abort
3. Determine the addresses of the object trees
4. Set object states to their defaults (just to be sure, in case something was forgotten when designing the resource file)
5. Turn the main menu on

These steps are implemented in DemoInterface.Init_Resource :

```
Procedure Init_Resource;
VAR ResourceName : String;
BEGIN
  ResourceName := 'GEMDEMO.RSC' + #0;    { Note #0  ! }
  rsrc_load(addr(ResourceName[1]));      { Note [1] ! }
  IF GemError = 0 THEN FatalError('GEMDEMO.RSC is missing!');
  rsrc_gaddr(R_TREE, MAINMENU, menu);
                                  { : set menu tree ptr      }
  rsrc_gaddr(R_TREE, ABOUTBOX, about);
                                  { : set aboutbox tree ptr  }
  rsrc_gaddr(R_TREE, SLCTDEMO, selection);
                                  { : set selection tree ptr }

  { set default states : }
  SetMenuState(s_disable, s_enable, s_disable, s_enable,
               s_disable);
  SetObjectStatus(about,     ABOUTOK,  NORMAL);
  SetObjectStatus(selection, BOXES,    SELECTED);
  SetObjectStatus(selection, LINES,    NORMAL);
  SetObjectStatus(selection, ELLIPSES, NORMAL);
  SetObjectStatus(selection, PIES,     NORMAL);
  menu_bar(menu, 1)     { draw menu }
END;
```

Note how the "Pascal-to-C-string" conversion is made : Use a normal Pascal string, append a "C"-style #0, and ignore the length-of-string element [0] when passing the address to rsrc_load.Also note how the WINMENU is changed, so that only the TXTOPEN, GRAOPEN and GRASELEC items are enabled. DemoInterface.SetObjectStatus simply changes an object's state field :

```
Procedure SetObjectStatus(t                    : TreePtr;
                          index, newstatus : Integer);
BEGIN
  t^[index].o_status := newstatus    { quite simple... }
END;
```

Waiting for events

In the demo program a rather simple event scheme is used : Wait for something to be present in the message-pipe, react upon the messages and do nothing else!
More formally, this can be expressed so :

```
1. As long as the user doesn't want to quit do :
2.   Wait for an event
3.   If it's a menu event then call the menu handler else
4.   If it's a window event then call the window handler
5. Loop
6. Close still open windows if any exist
```

This was implemented in GemDemo.Dispatch :

```
Procedure Dispatch;
VAR
  pipe : ARRAY [0..15] OF Integer;   { message pipe }
BEGIN
  REPEAT
    evnt_mesag(@pipe);          { wait for an event... }
    CASE pipe[0] OF             { take action          }
      MN_SELECTED  : HandleMenu   (pipe[3], pipe[4]);
      WM_REDRAW    : RedrawWindow (pipe[3], pipe[4], pipe[5],
                                   pipe[6], pipe[7]);
      WM_TOPPED    : TopWindow    (pipe[3]);
      WM_CLOSED    : CloseWindow  (pipe[3]);
      WM_FULLED    : FullWindow   (pipe[3]);
      WM_SIZED     : SizeWindow   (pipe[3], pipe[6], pipe[7]);
      WM_MOVED     : MoveWindow   (pipe[3], pipe[4], pipe[5])
    END
  UNTIL quit;
  CloseWindow(textwindow); { make sure, the windows are closed }
  CloseWindow(grafwindow)
END;
```

Dealing with menu events

When the user selected a menu item, the menu handler is called in order to do the requested operation and reset the menu bar :

1. If the item was ABOUTBAR then show the ABOUTBOX
2. If the item was a window operation then call the window handler
3. Reset the menu title to normal video (non-selected)

DemoMenu.HandleMenu implements this :

```
Procedure HandleMenu;
BEGIN
  CASE title OF
    DESKMENU : CASE index OF
                 ABOUTBAR : DoAboutBox
               END;
    WINMENU  : CASE index OF
                 WINCLOSE : CloseTopWindow;
                 TXTOPEN  : OpenTextWindow;
                 TXTCLOSE : CloseTextWindow;
                 GRAOPEN  : OpenGraphicsWindow;
                 GRACLOSE : CloseGraphicsWindow;
                 GRASELEC : SelectGraphicsDemo
               END;
    QUITMENU : CASE index OF
                 QUITBAR  : quit := TRUE
               END
  END;
  menu_tnormal(menu, title, 1) { set menu title back to normal }
END;
```

Handling the ABOUTBOX

The ABOUTBOX shows some information about the demo and waits for the user to click the [Ok] button. The general steps involved in showing a dialog box are :

1. Center the dialog box on the screen
2. Reserve some memory space for the dialog box border area
3. Draw a growing box outline
4. Draw the dialog box
5. Let the AES supervise all events until an object with an Exit flag is chosen
6. Draw a shrinking box outline
7. Restore the border area memory
8. Reset selected objects if desired

The AES automatically makes sure to place a Redraw message in the pipe.

```
Procedure DoAboutBox;
VAR
  x, y, w, h : Integer; { Dialog box outline size    }
  i          : Integer; { Item index ending dialog box (ABOUTOK) }
BEGIN
                { centre on screen   }
  form_center(about, x, y, w, h);
                { reserve RAM        }
  form_dial(FMD_START, 0, 0, 0, 0, x, y, w, h);
                { draw growing box   }
  form_dial(FMD_GROW, 0, 0, 0, 0, x, y, w, h);
                { draw dialog box    }
  objc_draw(about, 0, $7FFF, x, y, w, h);
                { do the dialog      }
  i := form_do(about, 0);
                { shrinking box      }
  form_dial(FMD_SHRINK, 0, 0, 0, 0, x, y, w, h);
                { release RAM and redraw borders }
  form_dial(FMD_FINISH, 0, 0, 0, 0, x, y, w, h);
                { reset button state }
  SetObjectStatus(about, ABOUTOK, NORMAL)
END;
```

Notes : The form_dial calls with the FMD_START and FMD_FINISH parameters stores and restores, respectively, the screen area where the dialog box will be shown. The rest of the redraw, however, must be conducted by the application (GEMDEMO) itself.objc_draw is set to start drawing at the root (0) and all levels ($7FFF = 32767). A such enormous number is chosen to be 117% sure that all elements are drawn, no matter how obscure the dialog tree is!

Handling the SLCTDEMO dialog

Dealing with the SLCTDEMO dialog really isn't much different from handling the ABOUTBOX, except that step 8 (reset objects) isn't performed, as this dialog's selectable items are all radio buttons, i.e. the AES automatically deselects the currently selected when another item is chosen by the user. The final step 8 is replaced with :

8 : Determine which icon was selected

DemoWindows.SelectGraphicsDemo implements this :

```
Procedure SelectGraphicsDemo;
VAR
  x, y, w, h : Integer;    { dialog border size       }
  selected   : Integer;    { selected demo icon index }
BEGIN
  form_center(selection, x, y, w, h);
  form_dial(FMD_START,  0, 0, 0, 0, x, y, w, h);
  form_dial(FMD_GROW,   0, 0, 0, 0, x, y, w, h);
  objc_draw(selection,  0, $7FFF,   x, y, w, h);
  selected := form_do(selection, 0);
  form_dial(FMD_SHRINK, 0, 0, 0, 0, x, y, w, h);
  form_dial(FMD_FINISH, 0, 0, 0, 0, x, y, w, h);
  CASE selected OF                  { set correct demo state : }
    BOXES     : demo := BoxesDemo;
    LINES     : demo := LinesDemo;
    ELLIPSES  : demo := EllipsesDemo;
    PIES      : demo := PiesDemo
  END;
  ForceGraphicsRedraw { make sure graphics window is redrawn }
END;
```

Opening the windows

Window management can be divided into the following actions :

```
- creation           : reserving memory
- set-up             : setting title, info-line etc.
- opening            : putting the window on the screen
- changing/redrawing : changing size, title etc. and updating
                       work area
- closing            : removing the window from the screen
- deleting           : release memory
```

We shall in this section concentrate on creation, set-up and opening. The steps are :

1. Prevent the AES from responding to mouse actions
2. Decide window elements to be active (name, close box etc.)
3. Decide maximum window size (when fulled)
4. Create the window and store the window handle
5. Set the window title. The title must be a static (global) variable in "C" format!
6. Draw expanding rectangle
7. Open the window on a specified position
8. Setting the WINMENU items correctly
9. Allow the AES to respond to mouse actions again

The AES will all by itself put a redraw message in the pipe.

Two almost identical procedures exist, one for the text window and one for the graphics window, both of which can be found in DemoWindows : OpenTextWindow and OpenGraphicsWindow, where only the last is shown here :

```
Procedure OpenGraphicsWindow;
BEGIN
  { create window : }
  grafwindow := wind_create(GrafElements, MinX,MinY, MaxW,MaxH);
  IF grafwindow >= 0 THEN BEGIN   { created ok... }
    wind_update(BEG_UPDATE);      { AES : leave us alone! }

    { set window name : }
    wind_set(grafwindow, WF_NAME,
             HiPtr(GrafName[1]), LoPtr(GrafName[1]), 0, 0);

    { draw a nice expanding box and open window }
    graf_growbox(GrafX + GrafW DIV 2 - 5,
                 GrafY + GrafH DIV 2 - 5, 10, 10,
                 GrafX, GrafY, GrafW, GrafH);
    wind_open(grafwindow, GrafX, GrafY, GrafW, GrafH);

    { set WINMENU's state }
    SetMenuState(s_enable, s_leave, s_leave, s_disable,
                 s_enable);
    wind_update(END_UPDATE)         { let the AES rule again }
  END
END;
```

Notes : The wind_update calls tells the AES that we're changing the desktop, and don't want to be interfered doing that. Also note the use of the HiPtr and LoPtr functions to return the high-word and low-word of a variable's address respectively.

Redrawing windows

This is one of the more exiting aspects, frightening at first but actually quite simple when you get to know it. The basic principle of redrawing a window is that of redrawing a number of small rectangles, each representing a visible area of the window. These rectangles are kept in a "rectangle list" which the AES manages automatically. Each window has its own rectangle list, which it empties when a redraw is requested.In order to redraw a window, the following steps are to be performed :

```
 1. Erase mouse and begin window update
 2. Get the first element in the rectangle list
 3. While rectangle from rectangle list is not empty do :
 4.   If the rectangle is to be redrawn (intersects) :
 5.     Set clipping rectangle
 6.     Clear it
 7.     Call the appropriate redraw procedure (text or graphics)
 8.   Else ignore
 9.   Get next rectangle from the rectangle list
10. Loop
11. Turn mouse back on and end window update
```

As strange as it may seem the actual redraw procedure will do a complete drawing, but as the clipping rectangle is set, only a part of it is actually drawn on the screen. This can be done without an enormous waste of time as the process of ignoring graphics is much faster than the actual setting of points.

The above scheme is implemented in DemoWindows.RedrawWindow :

```
Procedure RedrawWindow(handle, x0, y0, w0, h0 : Integer);
VAR
  R1, R2      : Grect;      { used for conversion purposes }
  x, y, w, h : Integer;     { rectangle to redraw          }
  a           : Array_4;    { used for conversion purposes }

  { Redraw text window }
  Procedure DoTextRedraw;
  VAR
    x, y, w, h : Integer;     { work area size }
    i          : Integer;
  BEGIN
    wind_get(textwindow, WF_WORKXYWH, x, y, w, h);
    i := CharBoxHeight;          { y-offset to start writing at }
    WHILE i <= (y + h - 1 + CharBoxHeight) DO BEGIN
      v_gtext(VDI_handle, x, y + i, TextLine);
      INC(i, CharBoxHeight)      { one line written }
    END
  END;
```

```
  { Redraw graphics window }
  Procedure DoGraphicsRedraw;
  BEGIN
    CASE demo OF
      BoxesDemo    : DoBoxes;
      LinesDemo    : DoLines;
      EllipsesDemo : DoEllipses;
      PiesDemo     : DoPies
    END
  END;

BEGIN { RedrawWindow }
  R1.x := x0;  R1.y := y0;
  R1.w := w0;  R1.h := h0; { set up R1 }
  wind_update(BEG_UPDATE);          { we're updating!           }
  graf_mouse(M_OFF, NIL);           { and want no mice around! }
  vsf_color(VDI_handle, White);     { clear work area styles : }
  vsf_style(VDI_handle, SOLID);     {   solid, white fill      }

  { get first rectangle from the window rectangle list : }
  wind_get(handle, WF_FIRSTXYWH, x, y, w, h);

  WHILE (w <> 0) AND (h <> 0) DO BEGIN { there IS a rect. : }
    R2.x := x;  R2.y := y;
    R2.w := w;  R2.h := h; { set up R2   }
    IF intersect(R1, R2) THEN BEGIN    { is area destroyed?  }
      x    := R2.x;        y    := R2.y;
      w    := R2.w;        h    := R2.h;
      a[0] := x;           a[1] := y;
      a[2] := x + w - 1;   a[3] := y + h - 1;
      vs_clip(VDI_handle, 1, a);    { set clipping           }
      v_bar(VDI_handle, a);         { and clear messy rect.  }
      IF handle = textwindow THEN DoTextRedraw
        ELSE DoGraphicsRedraw
    END;

    { one rectangle done, the rest to go! }
    wind_get(handle, WF_NEXTXYWH, x, y, w, h)
  END;
  graf_mouse(M_ON, NIL); { let's see the mouse again  }
  wind_update(END_UPDATE) { now we're through updating }
END;
```

After some operations, it's required to redraw all of a window. This can be done in many ways, one of which is simply calling an appropriate redraw procedure with the entire work area as input parameters. Another method, the one used here, is that of sending messages from one part of an application to another via the AES pipe system! It's a terrible waste of time and resources, but informative, so let's take a look at the procedure DemoGraphs.ForceGraphicsRedraw :

```
Procedure ForceGraphicsRedraw;
VAR
  mypipe : ARRAY [0..7] OF Integer;
BEGIN
  GetWindowSize;                  { Redraw all of window  }
  mypipe[0] := WM_REDRAW;         { The message           }
  mypipe[3] := grafwindow;        { The window            }
  mypipe[4] := wX;                { x                     }
  mypipe[5] := wY;                { y                     }
  mypipe[6] := wW;                { width                 }
  mypipe[7] := wH;                { height                }
  appl_write(AES_handle, SizeOf(mypipe), addr(mypipe))
                                  { : write message       }
END;
```

The values in the pipe buffer are set up as described in the GEM documentation.

Topping, moving and sizing windows

Topping a window is very simple : Call the wind_set procedure with the new window handle and the correct function id, and the AES takes care of the rest, including placing a redraw message in the pipe if required. DemoWindows.TopWindow implements this :

```
Procedure TopWindow(handle : Integer);
BEGIN
  wind_set(handle, WF_TOP, 0,0,0,0)  { set handle to top window }
END;
```

Moving a window is not much more difficult : Call wind_set with window handle, function id and its new x,y coordinates and old width and height. DemoWindows.TopWindow shows a not too efficient implementation. Its lack of efficiency is due to the fact that allthough GEM puts both x, y, w and h information in the pipe, the width and height is ignored and have to be obtained again with help from wind_get. This is due to a wish of demonstrating one of the wind_get facilities : How to obtain information about the current window size and position. Well, here it is :

```
Procedure MoveWindow(handle, toX, toY : Integer);
VAR w, h, dummy : Integer;   { border area size }
BEGIN
  wind_get(handle, WF_CURRXYWH, dummy, dummy, w, h); { get w,h }
  wind_set(handle, WF_CURRXYWH, toX, toY, w, h);     { new x,y }
  IF handle = grafwindow THEN ForceGraphicsRedraw
END;
```

Sizing a window is much like moving it; the only difference is in the parameters :

```
Procedure SizeWindow(handle, newW, newH : Integer);
VAR x, y, dummy : Integer;
BEGIN
  wind_get(handle, WF_CURRXYWH, x, y, dummy, dummy); { get x,y }
  wind_set(handle, WF_CURRXYWH, x, y, newW, newH);   { set w,h }
  IF handle = grafwindow THEN ForceGraphicsRedraw
END;
```

Closing the windows

The term "close" is used rather loosely here : It covers two aspects of handling :
- removing the window from the screen (closing it)
- removing the window information from the memory (deleting it)

In the GEMDEMO we both close and delete a window when the user closes it, as shown in both DemoWindows.CloseGraphicsWindow and CloseTextWindow of which the former is shown here :

```
Procedure CloseGraphicsWindow;
VAR x, y, w, h : Integer;          { window border area }
BEGIN
  IF grafwindow >= 0 THEN BEGIN
    wind_update(BEG_UPDATE);       { leave us alone }

    { get border size and draw a shrinking box : }
    wind_get(grafwindow, WF_CURRXYWH, x, y, w, h);
    graf_shrinkbox(x + w DIV 2 - 5, y + h DIV 2 - 5, 10, 10,
                   x, y, w, h);
    wind_close(grafwindow);        { remove window from screen }
    wind_delete(grafwindow);       { and memory too            }
    grafwindow := -1;              { -1 to prevent mistakes }
    IF textwindow >= 0 THEN        { update WINMENU            }
      SetMenuState(s_leave, s_leave, s_leave, s_enable,
                   s_disable)
    ELSE
      SetMenuState(s_disable, s_leave, s_leave, s_enable,
                   s_disable);
    wind_update(END_UPDATE)        { let AES rule }
  END
END;
```

Note the order in which the operations are performed : A window MUST be closed before it's deleted or your ST just might get very strange ideas!

Finishing up with the resources

When the program is through with the resources, it can release the memory occupied by these. Before doing so, it removes the menu bar from the screen, as shown in DemoInterface.End_Resource:

```
Procedure End_Resource;
BEGIN
  menu_bar(menu, 0);     { remove menu from screen }
  rsrc_free              { release memory          }
END;
```

Saying good-bye

All good things must have an end, and so must our demo :

1. Shrink a box to indicate to the user that we're saying good-bye.
2. Close the virtual workstation
3. Exit application

The above steps are contained in DemoInterface.End_Gem :

```
Procedure End_Gem;
BEGIN
  graf_shrinkbox(0, 0, 0, 0, MinX, MinY, MaxW, MaxH);
  v_clsvwk(VDI_handle);     { close virtual workstation  }
  appl_exit                 { exit application (GEMDEMO) }
END;
```

The main program

Finally, we show the main program, which performs the following steps :

```
1. Initialize GEM
2. If GEM initialization was successful then :
3.   Initialize the resources
4.   Create default graphics data
5.   Call the event dispatcher
6.   End resource usage when the user selected to quit
7.   End GEM usage
8. Else abort with error message
```

The implementation is :

```
BEGIN  { main program }
  IF Init_Gem THEN BEGIN        { initialize                    }
    graf_mouse(ARROW, NIL);     { arrow mouse form              }
    Init_Resource;              { load and set up resources     }
    MakeGraphData;              { set up data for graphics demo }
    Dispatch;                   { handle the events until quit  }
    End_Resource;               { remove resources              }
    End_Gem                     { end GEM usage                 }
  END
  ELSE FatalError('Could not initialize | GEM properly!')
END.
```

This concludes the GEM demo section.

The HighSpeed libraries

Purpose of this section
This section concentrates on the definitions in the HighSpeed libraries GemDecl, GemAES and GemVDI. It is mainly intended as a reference and therefore kept short. For full understanding, it is recommended that it's read in conjunction with a more fulfilling document on GEM programming.

All procedures/functions are shown with their heading and a very short description of their purpose, their parameters/return values and mostly a short example of their use. The constants are mainly listed together with the procedures that use them.

Although the style is rather formal, it is the author's hope that you, dear reader, don't find it triple-B : Boring Beyond Belief!

Basic terms
Before discussing the procedures, functions etc. it would be of help to you to know the following terms :

Term	Interpretation
„Button“	A button on the mouse
„Key“	A key on the keyboard

It may be quite natural to you, but unfortunately not all literature is consequent when it comes to these terms.

Introduction to the HighSpeed libraries
The HighSpeed libraries implement a full set of GEM routines on the ST. We have chosen to adopt the more-or-less standard calling names used by the „C“ language libraries and adopt the parameter layout and names as well. This was done in order to simplify reading additional GEM documentation such as the Digital Research GEM manuals, as most of the documents available concentrate on the „C“ way of doing things.As most of GEM was written in „C“, it is necessary to obey the requirements of parameters, especially the extensive use of pointers instead of typed parameters, and the „C“ way of using strings, which is a bit different from the HighSpeed Pascal way. In HighSpeed Pascal, the string layout is as follows :

```
String element        holds

[0]                   the length of the string
[1]                   first character of the string
...
[n]                   n'th character of the string
```

„C" uses this layout :

```
[0]                 the first character of the string
[1]                 second character of the string
...
[n]                 (n-1)'th character of the string
[n+1]               CHR(0) = #0 = string terminator character
```

This implies that when passing a pointer to a string to a GEM function such as rsrc_load you will have to append a #0 and ignore the first element [0] :

```
VAR filename : String;
...
  filename := 'MYFILE.RSC' + #00;
  rsrc_load( addr( filename[1]) );
```

In this way, the Pascal length-of-string element [0] is ignored.

Note that the procedures which take a string as parameter don't have to worry about the string length element, but they do have to take care of appending a #0

GemDecl

This library contains constants, types and variables used by the other two libraries together with a few procedures.

The constants are :

```
control_max = 11;  { Max size of control array                    }
intin_max   = 131; { Max size of intin array                      }
intout_max  = 139; { Max size of intout array                     }
workout_max = 56;  { Max size of workout array = 44+control_max   }
addrin_max  = 15;  { Max size of addrin array                     }
addrout_max = 15;  { Max size of addrout array                    }
global_max  = 14;  { Max size of global array                     }
pts_max     = 144; { Max size of ptsin/ptsout arrays              }
```

The declared types are :

```
{ Parameter blocks }
AES_param     = ARRAY [0..5] OF ^global_ARRAY;          { AES }
VDI_param     = ARRAY [0..4] OF ^global_ARRAY;          { VDI }

{ Other strictly GEM-related arrays }
global_ARRAY  = ARRAY [0..global_max]  OF Integer;
control_ARRAY = ARRAY [0..control_max] OF Integer;
intin_ARRAY   = ARRAY [0..intin_max]   OF Integer;
intout_ARRAY  = ARRAY [0..intout_max]  OF Integer;
ptsin_ARRAY   = ARRAY [0..pts_max]     OF Integer;
ptsout_ARRAY  = ARRAY [0..pts_max]     OF Integer;
addrin_ARRAY  = ARRAY [0..addrin_max]  OF Pointer;
addrout_ARRAY = ARRAY [0..addrout_max] OF Pointer;
workout_ARRAY = ARRAY [0..workout_max] OF Integer;

{ Multi-purpose }
ARRAY_2  = ARRAY [0..1]  OF Integer;
ARRAY_3  = ARRAY [0..2]  OF Integer;
ARRAY_4  = ARRAY [0..3]  OF Integer;
ARRAY_5  = ARRAY [0..4]  OF Integer;
ARRAY_6  = ARRAY [0..5]  OF Integer;
ARRAY_8  = ARRAY [0..7]  OF Integer;
ARRAY_10 = ARRAY [0..9]  OF Integer;
ARRAY_16 = ARRAY [0..15] OF Integer;
ARRAY_37 = ARRAY [0..36] OF Integer;

{ A graphic rectangle }
GRect = RECORD
   x,y,w,h: Integer
END;
```

Declared variables are :

```
AES_pb  : AES_param;            { AES parameter block }
VDI_pb  : VDI_param;            { VDI parameter block }
control : control_ARRAY;        { General arrays :    }
intin   : intin_ARRAY;          { Input parameters    }
intout  : intout_ARRAY;         { Output parameters   }
ptsin   : ptsin_ARRAY;          { VDI arrays :        }
ptsout  : ptsout_ARRAY;
addrin  : addrin_ARRAY;         { AES arrays          }
addrout : addrout_ARRAY;
global  : global_ARRAY;         { Global information  }
```

Function **Min(a, b : Integer) : Integer;**

Return the smallest number of a and b.

Ex. : a := min(5,2) returns a = 2.

See also : Max

Function **Max(a, b : Integer) : Integer;**

Return the largest number of a and b.

Ex. : a := max(20,40) returns a = 40.

See also : Min

Function **HiPtr(VAR v) : Integer;**

Return the high-word of v's address.

Ex. : hi_word := HiPtr(a)

See also : LoPtr

Function **LoPtr(VAR v) : Integer;**

Return the low-word of v's address.

Ex : lo_word := LoPtr(a)

HiPtr and LoPtr especially finds their use with GemAES.wind_set, as it - depending on how it is called - requires a four-byte address in the form of two two-byte integers.

See also : HiPtr

Function **BitTest(No : Integer; Value : LongInt) : Boolean;**

Test if bit No in Value is set.

```
Ex. : IF BitTest(3, 200) THEN WriteLn('Bit 3 set') ELSE
                               WriteLn('Bit 3 clear')
```

Function **Intersect(r1 : GRect; VAR r2 : GRect) : Boolean;**

Test if two rectangles intersect (have an area in common). Used for window redraws. r2's contents are changed to hold the resultingrectangle, and hence contains garbage if no intersection was found.

Ex. :

```
VAR
  r1, r2 : GRect;
BEGIN
  r1.x :=  10;  r1.y := 10;  r1.w := 200;  r1.h := 100;
  r2.x := 125;  r2.y := 55;  r2.w := 100;  r2.h := 100;
  IF Intersect(r1, r2) THEN
    WITH r2 DO
      WriteLn('Intersection : x=', x, ' y=', y, ' w=', w,
              ' h=', h)
  ELSE
    WriteLn('No intersection!')
END
```

produces as output :
Intersection : x=125 y=55 w=85 h=55

See also : EmptyRect

Function **EmptyRect(r : GRect) : Boolean;**

Test if the rectangle r is empty, i.e. w <= 0 or h <= 0.

Ex. : IF NOT EmptyRect(r) THEN DoMyOwnDrawing

See also : Intersect

Procedure **MakeXYXY(xywh : GRect; VAR xyxy : Array_4);**

Convert between x1, y1, x2, y2 and x, y, w, h format. MakeXYXY and MakeXYWH are complementary, as the following example shows :

Ex. :

```
VAR
  xyxy : Array_4;
  xywh : GRect;
...
  MakeXYXY(xywh, xyxy);
  MakeXYWH(xyxy, xywh)
```

won't bring any change to xywh.

See also : MakeXYWH

Procedure **MakeXYWH(xyxy : Array_4; VAR xywh : GRect);**

Convert from XYXY system to XYWH system.

See also : MakeXYXY

GemVDI

All GEM VDI procedures and functions can be found in this section.
The VDI takes care of basic graphics operations, which the AES then takes advantage of in order to do more advanced things like dialog box management and menu handling. But of course, the VDI procedures are available - and indispensable - for the GEM programmer.

The constants declared in GemVDI are :

```
                                { Fill interior styles :          }
HOLLOW       = 0;               { no fill                         }
SOLID        = 1;               { solid fill in current color     }
PATTERN      = 2;               { bit pattern                     }
HATCH        = 3;               { hatch                           }
UDFILLSTYLE= 4;                 { user defined                    }

                                { Some fill patterns :            }
DOTS    = 3;
GRID    = 6;
BRICKS = 9;
WEAVE   = 16;

                                { PolyLine line styles :          }
SOLID        = 1;                 same as above                  }
LDASHED      = 2;               { ************.... bit layout    }
DOTTED       = 3;               { ***.....***.....                }
DASHDOT      = 4;               { *******...***...                }
DASHED       = 5;               { ********........                }
DASHDOTDOT = 6;                 { ****...**..**...                }

                                { PolyLine end styles :           }
SQUARED = 0;
ARROWED = 1;
ROUNDED = 2;

                                { Text effects :                  }
NORMAL       = 0;               { normal appearance               }
BOLD         = 1;               { boldface                        }
SHADED       = 2;               { shaded                          }
SKEWED       = 4;               { italic                          }
UNDERLINED = 8;                 { underlined                      }
OUTLINE      = 16;              { outlined                        }
SHADOW       = 32;              { shadowed                        }
```

```
                                { Writing modes :                }
MD_REPLACE = 1;                 { replace                        }
MD_TRANS   = 2;                 { transparent                    }
MD_XOR     = 3;                 { XOR                            }
MD_ERASE   = 4;                 { erase                          }

                                { Bit-Blit flags :               }
ALL_WHITE  = 0;
S_AND_D    = 1;
S_AND_NOTD = 2;
S_ONLY     = 3;
NOTS_AND_D = 4;
D_ONLY     = 5;
S_XOR_D    = 6;
S_OR_D     = 7;
NOT_SORD   = 8;
NOT_SXORD  = 9;
D_INVERT   = 10;
S_OR_NOTD  = 11;
NOT_D      = 12;
NOTS_OR_D  = 13;
NOT_SANDD  = 14;
ALL_BLACK  = 15;

                                { Input modes :                  }
REQUEST    = 1;                 { request mode                   }
SAMPLE     = 2;                 { sample mode                    }
```

GemVDI also declares a couple of types :

```
MFDB = record                   { MFDB layout (MemoryForm)       }
  mptr       : POINTER;         { raster address                 }
  formwidth  : Integer;         { width of raster in pixels      }
  formheight : Integer;         { height of raster in pixels     }
  widthword  : Integer;         { width in terms of 16-bit words }
  formatflag : Integer;         { device specific/standard : 0/1 }
  memplanes  : Integer;         { raster planes                  }
  res1       : Integer;         { reserved...                    }
  res2       : Integer;
  res3       : Integer;
END;

String80  = String[80];
String125 = String[125];
```

```
Procedure v_opnwk(WorkIn     : intin_Array;
                  VAR handle : Integer;
                  VAR WorkOut : workout_Array);
```

Open workstation (load device drivers). WorkIn specifies what the workstation should look like, handle is the workstation handle, whereas WorkOut returns what the workstation does look like. Note : This procedure is not properly implemented on all ST-GEM versions and tend to crash system if used. This is due to the ST's missing device drivers for plotters, graphics tablets etc.

WorkIn : input parameters
handle : workstation/device handle
WorkOut : return parameters

See also : v_clswk, v_opnvwk, v_clsvwk, v_clrwk, v_updwk

```
Procedure v_clswk(handle : Integer);
```

Close the workstation opened by v_opnvk.

handle : workstation/device

See also : v_opnwk, v_opnvwk, v_clsvwk, v_clrwk, v_updwk

```
Procedure v_opnvwk(WorkIn     : intin_Array;
                   VAR handle : Integer;
                   VAR WorkOut : workout_Array);
```

Open a virtual workstation. On the ST, this means opening the screen. The WorkIn parameters are the same as the ones used by v_opnvk, but they are all ignored by the ST, so all attributes must be set by using the proper attribute procedures. This call must be preceded by a call to GemAES.appl_init. handle is the value obtained by an earlier call to graf_handle.

WorkIn : input parameters
handle : workstation/device handle (normally from graf_handle)
WorkOut : output parameters

See also : v_opnwk, v_clswk, v_clsvwk, v_clrwk, v_updwk

Procedure **v_clsvwk(handle : Integer);**

Close the virtual workstation (screen). This is usually one of the last calls, a program makes before terminating.

handle : workstation/device

See also : v_opnwk, v_clswk, v_opnvwk, v_clrwk, v_updwk

Procedure **v_clrwk(handle : Integer);**

Clear workstation, i.e. clear the screen and set it to the background color. On a printer or a plotter this would indicate paging, whereas the opcode would be written to a Metafile.

handle : workstation/device

See also : v_opnwk, v_clswk, v_opnvwk, v_clsvwk, v_updwk

Procedure **v_updwk(handle : Integer);**

Update workstation. This procedure is only of interest when using devices such as printers, plotters etc. but not the screen. It flushes the internal output-buffer and thereby makes sure that what the application wrote to the device really is printed.

handle : workstation/device

See also : v_opnwk, v_clswk, v_opnvwk, v_clsvwk, v_clrwk

Function **vst_load_fonts(handle, select : Integer) : Integer;**

Load fonts. It loads a character set (select) from a specified device. It returns 0 if no (other) character set(s) exist. This implies that 0 is returned if it is attempted to load a font for the ST screen, as it has one font only for each resolution. Is mainly used together with GDOS.

```
handle       : device
select       : the character set to load
Return value : loaded font, or 0 = no font loaded

Ex. : IF vst_load_fonts(VDI_handle, 2) = 0
        THEN WriteLn('No font 2');
```

See also : vst_unload_fonts, vst_font, vqt_name, vqt_fontin

Procedure **vst_unload_fonts(handle, select : Integer);**

Free up memory used by a font. GDOS related.

handle : device
select : font to remove

Ex. : vst_unload_font(VDI_handle, 2);

See also : vst_load_fonts, vst_font, vqt_name, vqt_fontinfo

Procedure **vs_clip(handle, clipflag : Integer;**
pxyarray : Array_4);

Turn clipping on/off. Graphical operations are only performed if their coordinates lay inside the clipping rectangle.

handle : device
clipflag : 0 = no clipping; 1 = clipping
pxyarray : corner coordinates

Ex. :

```
VAR
  p          : Array_4;
BEGIN
  { GEM initializations goes here... }
  p[0] := 100;  p[1] := 110;  { upper left  (x,y) = (100, 110) }
  p[2] := 400;  p[3] := 310   { lower right (x,y) = (400, 310) }
  vs_clip(VDI_handle, 1, p);  { turn clipping on }
...
  vs_clip(VDI_handle, 0, p);  { turn clipping off again }
```

Output Functions

These procedures handle simple graphics outputs such as line drawing, point setting etc. together with more complex ones like filling polygons etc.

```
Procedure v_pline(handle, count : Integer;
                  pxyarray : ptsin_Array);
```

Draw PolyLine. pxyarray contains coordinates, so that pxyarray[0]= starting x, pxyarray[1] = starting y, pxyarray[2] = to x, pxyarray[3] = to y etc. count contains the number of coordinates used = (highest array element used + 1) DIV 2.

```
handle   : device
count    : number of coordinate sets (x,y) (min. 2)
pxyarray : coordinate sets
```

Ex. :

```
VAR p : ptsin_Array;
...
BEGIN
  { GEM initializations go here... }
  p[0] := 10;  p[1] := 20;  p[2] := 30;  p[4] := 40;
  p[5] := 50;  p[6] := 60;  p[7] := 70;  p[8] := 80;
  v_pline(VDI_handle, 8 DIV 2, p);
  { try changing it to v_pmarker }
```

draws a line from (10,20) to (30,40) to (50,60) to (70,80).

See also : v_pmarker, vsl_type, vsl_udsty, vsl_width, vsl_color, vsl_ends

```
Procedure v_pmarker(handle, count : Integer;
                    pxyarray      : ptsin_Array);
```

Set a series of points. Coordinate setup is as for v_pline.

```
handle   : device
count    : number of coordinate sets (x,y) (min. 2)
pxyarray : coordinate sets
```

See also : v_pline, vsm_type, vsm_height, vsm_color

Procedure **v_gtext(handle, x, y : Integer; chstring : String80);**

Draw graphic text starting at coordinates (x,y) = lower left corner of the text.

handle : device
x, y : string coordinate
chstring : the string („C" format : #00 suffix)

Ex. : v_gtext(VDI_handle, 10, 20, 'HighSpeed Pascal' + #00);

See also : v_justified, vst_height, vst_point, vst_rotation, vst_color, vst_effects, vst_alignment

Procedure **v_fillarea(handle, count : Integer; pxyarray : ptsin_Array);**

Fill a polygon with the color, style, pattern etc. set by the attribute Functions. pxyarray holds the polygon's corner coordinates, whereas count holds the number of coordinates.

handle : device
count : number of coordinate sets
pxyarray : corner coordinates

See also : v_contourfill, vr_recfl, vsf_style, vsf_color, vsf_perimeter, vsf_updat, vsf_interior

Procedure **v_cellarray(handle : Integer; pxyarray : Array_4; rowlength, elused, numrows, wrtmode : Integer; colarray : intin_Array);**

Create a cell array. A cell array is a rectangle divided into a number of rows and columns, where each of these sub-areas can be assigned their own point color. The parameters are :
handle : Device handle
pxyarray : Corner coordinates
[0] = lower left x; [1] = lower left y
[2] = upper right x; [3] = upper right y
rowlength : Line length in colarray
elused : Number of zones in colarray lines
numrows : Number of lines in colarray
wrtmode : Character mode
colarray : Color specifications
See also : vq_cellarray

```
Procedure v_contourfill(handle, x, y, index : Integer);
```

Fill an area with the current fill colorstyle color, until the color given by index is met.

```
handle : device
x, y   : point within area to be filled
index  : contour color
```

```
See also : v_fillarea, vr_recfl, vsf_style, vsf_color,
           vsf_perimeter, vsf_updat, vsf_interior
```

```
Procedure vr_recfl(handle : Integer; pxyarray : Array_4);
```

Fill a rectangle with the currently set color and attributes. Pxyarray [0] and [1] defines one corner of the rectangle, and pxyarray [2] and [3] defines the diagonally opposite corner of the rectangle.

```
handle   : device
pxyarray : corner coordinates
```

Ex. :

```
VAR p : Array_4;
...
  p[0] := 10;  p[1] := 40;  p[2] := 200;  p[3] := 100;
  vr_recfl(VDI_handle, p)
```

```
See also : v_fillarea, v_contourfill, vsf_style, vsf_color,
           vsf_perimeter, vsf_updat, vsf_interior
```

Graphic Functions

These are the more advanced GEM procedures such as drawing circles, pies etc.

Note : In GEM, all normal degree-measurements are multiplied with 10 to give a one-digit precision when converting degrees from real to integer, e.g. 1.23 degrees becomes 12.3 in GEM notation.
A formula for converting to GEM format is this :

```
  GEMdegree := TRUNC(NormalDegree * 10);
```

Procedure **v_bar(handle : Integer; pxyarray : Array_4);**

Filled bar. Attributes must be set beforehand.

handle : device
pxyarray : diagonally opposite corner coordinates

See also : v_arc, v_pieslice, v_circle, v_ellarc,
v_ellpie, v_ellipse, v_rbox, v_rfbox

Procedure **v_arc(handle, x, y, radius, begang, endang : Integer);**

Draw an arc.

handle : device
x, y : center point
radius : arc's radius
begang : starting angle (degrees * 10)
endang : ending angle (degrees * 10)

Ex. : v_arc(VDI_handle, 320, 200, 100, 0, 450);

See also : v_bar, v_pieslice, v_circle, v_ellarc, v_ellpie,
v_ellipse, v_rbox, v_rfbox

Procedure **v_pieslice(handle, x, y, radius,**
begang, endang : Integer);

Draw a filled arc in the same way as v_arc, except that the end points of the arc are connected to the centre point, and the thereby created area is filled.

handle : device
x, y : center point
radius : arc's radius
begang : starting angle (degrees * 10)
endang : ending angle (degrees * 10)

Ex. : v_pieslice(VDI_handle, 320, 200, 100, 0, 450);

See also : v_bar, v_arc, v_circle, v_ellarc, v_ellpie, v_ellipse,
v_rbox, v_rfbox

Procedure **v_circle(handle, x, y, radius : Integer);**

Draw a filled circle. All attributes must be set beforehand.
handle : device
x, y : center point
radius : circle's radius

Ex. : v_circle(VDI_handle, 320, 200, 100);

See also : v_bar, v_arc, v_pieslice, v_ellarc, v_ellpie,
v_ellipse, v_rbox, v_rfbox

Procedure **v_ellarc(handle,**
x, y,
xradius, yradius,
begang, endang : Integer);

Draw an elliptical arc.

handle : device
x, y : center point
xradius : radius in the x-direction
yradius : radius in the y-direction
begang : starting angle (degrees * 10)
endang : ending angle (degrees * 10)

Ex. : v_ellarc(VDI_handle, 320, 200, 50, 70, 0, 450);

See also : v_bar, v_arc, v_pieslice, v_circle, v_ellpie,
v_ellipse, v_rbox, v_rfbox

```
Procedure v_ellpie(handle,
                   x, y,
                   xradius, yradius,
                   begang,  endang   : Integer);
```

Draw a filled elliptical arc in the same way as v_ellarc.

```
handle  : device
x, y    : center point
xradius : radius in the x-direction
yradius : radius in the y-direction
begang  : starting angle (degrees * 10)
endang  : ending angle   (degrees * 10)
```

```
Ex. : v_ellpie(VDI_handle, 320, 200, 50, 70, 5, 455);
```

```
See also : v_bar, v_arc, v_pieslice, v_circle, v_ellarc,
           v_ellipse, v_rbox, v_rfbox
```

```
Procedure v_ellipse(handle, x, y,
                    xradius,
                    yradius      : Integer);
```

Draw a filled ellipse. Attributes must be set beforehand.

```
handle  : device
x, y    : center point
xradius : radius in the x-direction
yradius : radius in the y-direction
```

```
Ex. : v_ellipse(VDI_handle, 320, 200, 50, 70);
```

```
See also : v_bar, v_arc, v_pieslice, v_circle, v_ellarc,
           v_ellpie, v_rbox, v_rfbox
```

```
Procedure v_rbox(handle : Integer; xyarray : Array_4);

Draw a box with rounded corners.
handle  : device
xyarray : diagonally opposite corner coordinates

Ex. :
VAR
  a : Array_4;
...
  a[0] := 10;  a[1] := 30;  a[2] := 150;  a[3] := 200;
  v_rbox(VDI_handle, a);

See also : v_bar, v_arc, v_pieslice, v_circle, v_ellarc,
           v_ellpie, v_ellipse, v_rfbox


Procedure v_rfbox(handle : Integer; xyarray : Array_4);

Draw a filled box with rounded corners using the current attributes.

handle  : device
xyarray : diagonally opposite corner coordinates

See also : v_bar, v_arc, v_pieslice, v_circle, v_ellarc,
           v_ellpie, v_ellipse, v_rbox


Procedure v_justified(handle, x, y : Integer;
                      gstring      : String80;
                      jlength,
                      wordspace,
                      charspace    : Integer);
Write justified text.

handle    : device
x, y      : where to start writing
gstring   : string to write (remember #00)
jlength   : width of output area
wordspace : insert additional spaces between words?
                0 = no
              <> 0 = yes, if required
charspace : insert additional spaces between letters?
                0 = no
              <> 0 = yes, if required

See also : v_gtext, vst_height, vst_point, vst_rotation,
           vst_color, vst_effects, vst_alignme
```

Attribute Functions

The following procedures are used for setting colors, styles etc. for the different graphic operations. HighSpeed Pascal defines constants for a number of these colors/ styles/etc :

Note : These constants are defined in GemAES.

```
White      =    0;  Black      =    1;
Red        =    2;  Green      =    3;
Blue       =    4;  Cyan       =    5;
Yellow     =    6;  Magenta    =    7;
LWhite     =    8;  LBlack     =    9;
LRed       =   10;  LGreen     =   11;
LBlue      =   12;  LCyan      =   13;
LYellow    =   14;  LMagenta   =   15;
```

The constants prefixed with 'L' are light appearances.

For line styles, fill styles etc. it is recommended to read the sections concerning these attribute Functions.

```
Procedure vswr_mode(handle, mode : Integer);

Set the writing mode of all following graphics operations.

handle : device
mode   : HighSpeed CONST

  1      MD_REPLACE     replace
  2      MD_TRANS       transparent
  3      MD_XOR         XOR
  4      MD_ERASE       reverse transparent

Ex. : vswr_mode(VDI_handle, MD_REPLACE);
```

Procedure **vs_color(handle, index : Integer; rgbin : Array_3);**

Set color representation.

handle : device
index : color index in palette
rgbin : normalized red, green and blue intensity values :

[0] = red
[1] = green
[2] = blue

The normalized values are in the range [0..1000].

Ex. :

```
VAR r : Array_3;
...
  r[0] := 750;  r[1] := 400;  r[2] := 200;
  vs_color(VDI_handle, 3, r);
  { sets color index 3 to a new color }
```

Procedure **vsl_type(handle, style : Integer);**

Set PolyLine line style. The line styles are :

style	HighSpeed CONST	pattern MSB LSB
1	SOLID	**********....
2	LDASHED	***.....***.....
3	DOTTED	*******...***...
4	DASHDOT	********.........
5	DASHED	****...**..**...
6	DASHDOTDOT	
7		user defined (see vsl_udsty below)

A style is 16 bits wide; „*" = set (1); „." = clear (0).

Ex. : vsl_type(VDI_handle, DASHDOTDOT);

See also : v_pline, v_pmarker, vsl_udsty, vsl_width, vsl_color, vsl_ends

Procedure **vsl_udsty(handle, pattern : Integer);**

Set user defined line style with a 16-bit pattern laid out in the manner described in vsl_type.

handle : device
pattern : 16-bit pattern

Ex. :

```
vsl_udsty(VDI_handle, $CCCC);    { **..**..**..**.. layout }
vsl_type(VDI_handle, 7);         { user line style        }
```

See also : v_pline, v_pmarker, vsl_type, vsl_width, vsl_color, vsl_ends

Procedure **vsl_width(handle, width : Integer);**

Set PolyLine's line width. Note that width must be an odd number.

handle : device
width : new line width

Ex. : vsl_width(VDI_handle, 7); { pretty thick line! }

See also : v_pline, v_pmarker, vsl_type, vsl_udsty, vsl_color, vsl_ends

Procedure **vsl_color(handle, colindex : Integer);**

Set PolyLine color.

handle : device
colindex : a color index from the palette

Ex. : vsl_color(VDI_handle, Blue);

See also : v_pline, v_pmarker, vsl_type, vsl_udsty, vsl_width, vsl_ends

Procedure **vsl_ends(handle, begstyle, endstyle : Integer);**

Set PolyLine start and end styles. The style layout is as follows:

numeric value	HighSpeed CONST
1	SQUARED
2	ARROWED
3	ROUNDED

handle : device
begstyle : line begin style
endstyle : line end style

Ex. : vsl_ends(VDI_handle, ARROWED, ROUNDED);

See also : v_pline, v_pmarker, vsl_type, vsl_udsty, vsl_width, vsl_color

Procedure **vsm_type(handle, symbol : Integer);**

Set PolyMarker type. The different types are :

symbol : value	shape
1	point
2	plus sign
3	star
4	square
5	diagonal cross
6	diamond
n	device dependent

handle : device

There are six shapes available on the ST screen.

Ex. : vsm_type(VDI_handle, 4);

See also : v_pmarker, v_pline, vsm_height, vsm_color

```
Procedure vsm_height(handle, height : Integer);

Set PolyMarker height.

handle : device
height : desired height

Ex. : vsm_height(VDI_handle, 5);

See also : v_pmarker, v_pline, vsm_type, vsm_color


Function vsm_color(handle, colindex : Integer);

Set PolyMarker color to colindex in the current palette.

handle   : device
colindex : color index

Ex. : vsm_color(VDI_handle, Yellow);

See also : v_pmarker, v_pline, vsm_type, vsm_height


Procedure vst_height(handle, height : Integer;
                     VAR charwidth,
                         charheight,
                         cellwidth,
                         cellheight : Integer);
Set absolute graphic character height. As a result of the set height,
new character cell sizes are calculated and returned.

handle     : device
height     : new height in pixels
charwidth  : width of a char with the new height
charheight : height of character
cellwidth  : width of a character cell with the new height
cellheight : height of cell

Ex. :
CONST newheight = 13;
VAR
  chW, chH, celW, celH : Integer;
...
  vst_height(VDI_handle, newheight, chW, chH, celW, celH);

See also : v_gtext, v_justified, vst_point, vst_color,
           vst_rotation, vst_effects, vst_alignment
```

```
Procedure vst_point(handle, point : Integer;
                    VAR charwidth,
                        charheight,
                        cellwidth,
                        cellheight : Integer);
```

Basically the same Function as vst_height, except for character height being given in „points" = 1/72 of an inch.

```
handle     : device
height     : new height in points
charwidth  : width of a char with the new height
charheight : height of character
cellwidth  : width of a character cell with the new height
cellheight : height of cell
```

See also : v_gtext, v_justified, vst_height, vst_rotation, vst_color, vst_effects, vst_alignment

```
Procedure vst_rotation(handle, angle : Integer);
```

Set character baseline vector to a given (GEM style) degree.

```
handle : device
angle  : baseline angle
```

```
Ex. : vst_rotation(VDI_handle, 2700);
```

See also : v_gtext, v_justified, vst_height, vst_point, vst_color, vst_effects, vst_alignment

```
Procedure vst_font(handle, font : Integer);
```

Select either the system font or a previously loaded font.

```
handle : device
font   : desired font : system = 1 (6 * 6 font)
```

```
Ex. : vst_font(VDI_handle, 1);
```

vst_font is of most use together with GDOS.

See also : vst_load_fonts, vst_unload_fonts, vqt_name, vqt_fontinfo

Procedure **vst_color(handle, colindex : Integer);**

Set text color.

handle : device
colindex : new text color

Ex. : vst_color(VDI_handle, 4);

See also : v_gtext, v_justified, vst_height, vst_point,
vst_rotation, vst_effects, vst_alignment

Procedure **vst_effects(handle, effect : Integer);**

Set special text effects. The available effects are :

effect	HighSpeed CONST	resulting style
0	NORMAL	normal
1	BOLD	boldface
2	SHADED	shaded
4	SKEWED	italic
8	UNDERLINED	underlined
16	OUTLINE	outlined
32	SHADOW	shadowed

The effects may be combined by adding them.

handle : device

Ex. : vst_effects(VDI_handle, BOLD + OUTLINE);
vst_effects(VDI_handle, NORMAL);

See also : v_gtext, v_justified, vst_height, vst_point,
vst_rotation, vst_color, vst_alignment

```
Procedure vst_alignment(handle, horin,  vertin  : Integer;
                        VAR     horout, vertout : Integer);

Set horizontal and vertical text alignment.

handle  : device
horin   : horizontal alignment :

              0 : left aligned
              1 : centred
              2 : right aligned

vertin  : vertical alignment

              0 : base line
              1 : half line
              2 : ascent line
              3 : bottom line
              4 : descent line
              5 : top line

horout  : the value used by the VDI
vertout : the value used by the VDI

See also : v_gtext, v_justified, vst_height, vst_point,
           vst_rotation, vst_color, vst_effects



Procedure vsf_interior(handle, style : Integer);

Set fill interior style. The available styles are:

style  HighSpeed CONST  surface

  0    HOLLOW           not filled
  1    SOLID            solid fill with fill color
  2    PATTERN          dotted fill
  3    HATCH            cross-hatch fill
  4    UDFILLSTYLE      user defined fill style

handle : device

Ex. : vsf_interior(VDI_handle, PATTERN);

See also : v_fillarea, v_contourfill, vr_recfl, vsf_style,
           vsf_color, vsf_perimeter, vsf_updat
```

Procedure **vsf_style(handle, styleindex : Integer);**

Set fill style index. This allows the selection of one of either 24 bit patterns or 12 cross-hatch patterns. Before calling this procedure, it must be selected whether the styleindex concerns a bit pattern or a cross-hatch pattern by calling vsf_interior.

handle : device
styleindex : new style

Ex. :

```
vsf_interior(VDI_handle, HATCH);     { cross-hatch          }
vsf_style   (VDI_handle, 3);         { cross-hatch style 3  }
vsf_interior(VDI_handle, PATTERN);   { bit pattern          }
{ a call to a fill procedure at this point would result in }
{ a bit pattern code 3 to be drawn                         }
```

See also : v_fillarea, v_contourfill, vr_recfl, vsf_color, vsf_perimeter, vsf_updat, vsf_interior

Procedure **vsf_color(handle, colorindex : Integer);**

Set fill color.

handle : device
colorindex : new fill color

Ex. : vsf_color(VDI_handle, Magenta);

See also : v_fillarea, v_contourfill, vr_recfl, vsf_style, vsf_perimeter, vsf_updat, vsf_interior

Procedure **vsf_perimeter(handle, pervis : Integer);**

Set fill frame on/off.

handle : device
pervis : fill frame flag : 0 = off; 1 = on (default)

Ex. : vsf_perimeter(VDI_handle, 0); { turn frame off }

See also : v_fillarea, v_contourfill, vr_recfl, vsf_style, vsf_color, vsf_updat, vsf_interior

```
Procedure vsf_udpat(handle : Integer; VAR pfillpat;
                    planes : Integer);

Set user defined fill pattern.

handle   : device
pfillpat : memory block containing the fill pattern
planes   : number of colors involved

See also : v_fillarea, v_contourfill, vr_recfl, vsf_style,
           vsf_color, vsf_perimeter, vsf_interior
```

Raster operations

All but one of the raster procedures require MFDBs (Memory Form Definition Blocks) as parameters, so before looking at the procedures, let's recall the MFDB layout :

```
TYPE
  MFDB = RECORD
    mptr       : Pointer; { upper left corner of raster          }
    formwidth  : Integer; { raster width in pixels               }
    formheight : Integer; { raster height in pixels              }
    widthword  : Integer; { raster width DIV 16 + 1 = word size  }
    formatflag : Integer; { 0 = device specific; 1 = standard    }
    memplanes  : Integer; { raster planes                        }
    res1       : Integer; { reserved                             }
    res2       : Integer;
    res3       : Integer
  END;
```

Note that rasters must begin on a word boundary and be a multiple of 16 bits.

```
Procedure vro_cpyfm(handle, wrmode : Integer;
                    pxyarray       : Array_8;
                    psrcMFDB,
                    pdesMFDB       : MFDB);
```

Copy raster, opaque. wrmode defines the logical operations that are performed on the raster when stored. HighSpeed Pascal defines some constants concerning the raster operations :

value	HighSpeed CONST	operation performed s=source; d=destination
0	ALL_WHITE	0
1	S_AND_D	s AND d
2	S_AND_NOTD	s AND NOT d
3	S_ONLY	s
4	NOTS_AND_D	(NOT s) AND d
5	D_ONLY	d
6	S_XOR_D	s XOR d
7	S_OR_D	s OR d
8	NOT_SORD	NOT (s OR d)
9	NOT_SXORD	NOT (s XOR d)
10	D_INVERT	NOT d
11	NOT_D	NOT d
12	S_OR_NOTD	s OR (NOT d)
13	NOTS_OR_D	(NOT s) OR d
14	NOT_SANDD	NOT (s AND d)
15	ALL_BLACK	1

```
handle   : device
wrmode   : writing mode (see above)
pxyarray : [0]..[3] : source corner coordinates
           [4]..[7] : destination corner coordinates
           The coordinates are given in terms of two diagonally
           opposite corners.
psrcMFDB : source
pdesMFDB : destination

See also : vrt_cpyfm, vr_trnfm, v_get_pixel
```

```
Procedure vrt_cpyfm(handle, wrmode : Integer;
                    pxyarray       : Array_8;
                    psrcMFDB,
                    pdesMFDB       : MFDB;
                    color_index    : Array_2);
```

Copy raster, transparent.

```
handle      : device
wrmode      : writing mode : MD_TRANS, MD_REPLACE, MD_XOR,
              MD_ERASE
pxyarray    : [0]..[3] : source corner coordinates
              [4]..[7] : destination corner coordinates
              The coordinates are given in terms of two diagonal-
              ly opposite corners.
psrcMFDB    : source
pdesMFDB    : destination
color_index : [0] = color of set pixels
              [1] = color of clear pixels
```

See also : vro_cpyfm, vr_trnfm, v_get_pixel

```
Procedure vr_trnfm(handle    : Integer;
                   psrcMFDB,
                   pdesMFDB  : MFDB);
```

Transform form from standard format to device specific.

```
handle   : device
psrcMFDB : source Memory Form Definition Block
pdesMFDB : destination Memory Form Definition Block
```

See also : vro_cpyfm, vrt_cpyfm, v_get_pixel

```
Procedure v_get_pixel(handle, x, y : Integer;
                      VAR pel, index : Integer);
```

Determine if a pixel is set or clear. v_get_pixel can only be used in device-specific format.

```
handle : device
x, y   : pixel to get
pel    : pixel set = 1 or clear = 0
index  : pixel's color index
```

See also : vro_cpyfm, vrt_cpyfm, vr_trnfm

Input Functions

Most of the input procedures work in one of two ways :

mode	HighSpeed CONST	operation
1	REQUEST	waits for an event to occur
2	SAMPLE	determines current state and returns immediately

```
Procedure vsin_mode(handle, devtype, mode : Integer);

Set input mode.

handle  : device
devtype : input device type :

            0 : position input
            1 : value input
            2 : selection input
            3 : string input

mode    : Input mode

            0 : REQUEST
            1 : SAMPLE

See also : vqin_mode
```

```
Procedure vrq_locator(handle, x, y    : Integer;
                      VAR xout, yout  : Integer;
                      VAR term        : Integer);

Input locator, request mode.

handle     : device
x, y       : graphic cursor start position
xout, yout : graphic cursor end position
term       : the keyboard key/mouse button that ended the input
             #32 = left mouse button; #33 = right button

See also : vsm_locator
```

```
Function  vsm_locator (handle, x, y          : Integer;
                        VAR xout, yout, term : Integer) : Integer;
Input locator, sample mode.

handle       : device
x, y         : graphic cursor start position
xout, yout   : graphic cursor end position
term         : the keyboard key/mouse button that ended the input
                 #32 = left mouse button; #33 = right button
return value : status value :
               value   key press?  position change?

                  0      no           no     (note the
                  1      no           yes     binary appea-
                  2      yes          no      rance)
                  3      yes          yes
```

See also : vrq_locato

```
Procedure vrq_valuator(handle, valin : Integer;
                       VAR valout : Integer;
                       VAR term   : Integer);
Input valuator, request mode.
handle : device
valin  : start value
valout : end value
term   : key terminating operation
```

See also : vsm_valuator

```
Procedure vsm_valuator(handle, valin : Integer;
                       VAR valout    : Integer;
                       VAR term      : Integer;
                       VAR status    : Integer);
Input valuator, sample mode.
handle : device
valin  : start value
valout : end value
term   : key terminating operation
status : value  interpretation

         0      no actions
         1      value changed
         2      key pressed
```

See also : vrq_valuator

```
Procedure vrq_choice(handle, ch_in : Integer;
                     VAR ch_out    : Integer);
```

Input choice, request mode.

handle : device
ch_in : initial value
ch_out : pressed key

See also : vsm_choice

```
Function vsm_choice(handle : Integer; VAR choice : Integer)
                                                  : Integer;
```

Input choice, sample mode.

handle : device
choice : pressed key
Return value : status : 0 = no key; 1 = key pressed

See also : vrq_choice

```
Procedure vrq_string(handle, maxlen, echomode : Integer;
                     echo_xy                  : Array_2;
                     VAR instring             : String80);
```

Input string, request mode. Waits for <Return> key before terminating.

handle : device
maxlen : maximum string length
echomode : 0 = no echo; 1 = echo
echo_xy : echo area (x,y)
instring : the string input

See also : vsm_string

```
Function vsm_string(handle, maxlen,
                    echomode       : Integer;
                    echo_xy        : Array_2;
                    VAR instring   : String80) : Integer;

Input string, sample mode.

handle       : device
maxlen       : maximum string length
echomode     : 0 = no echo; 1 = echo
echo_xy      : echo area (x,y)
instring     : the string input
Return value : 0 = illegal key pressed; otherwise length
               of string.

See also : vrq_string

Procedure vsc_form(handle : Integer; pcurform : Array_37);

Set mouse form.

handle   : device
pcurform : cursor form :

           [ 0], [ 1] : hot spot (x,y)
           [ 2]       : must be 1 (!)
           [ 3]       : mask color index
           [ 4]       : form color index
           [ 5]..[20] : mask raster lines (16)
           [21]..[36] : form raster lines (16)

See also : graf_mouse


Procedure vex_timv(handle       : Integer;
                   timaddr      : Pointer;
                   VAR otimaddr : Pointer;
                   VAR timconv  : Integer);

Exchange timer interrupt vector. Directs the system timer inter-
rupt to user routine.

handle   : device
timaddr  : address of new timer routine
otimaddr : address of old timer routine
timconv  : interrupt interval (milliseconds)

See also : vex_butv, vex_motv, vex_curv
```

Procedure **v_show_c(handle, reset : Integer);**

Show graphic cursor (mouse). Each time v_show_c is called, an internal counter is updated, so it is necessary to call v_hide_c exactly as many times as v_show_c to hide the cursor. The internal counter can, however, be cleared if reset is set to 0. In normal operation, reset should be 1.

handle : device
reset : reset flag : 0 = clear counter; 1 = maintain counter

Ex. :

```
VAR i : Integer;
...
  FOR i := 1 TO 10 DO v_show_c(VDI_handle, 1);
  FOR i := 1 TO  9 DO v_hide_c(VDI_handle);
  { at this point, the cursor is still on }
  v_hide_c(VDI_handle);
  { now it's turned off }
```

See also : v_hide_c, graf_mouse

Procedure **v_hide_c(handle : Integer);**

Hide graphic cursor (mouse).

handle : device

See also : v_show_c, graf_mouse

Procedure **vq_mouse(handle : Integer;**
VAR pstatus, x, y : Integer);

Sample mouse button state and return graphic cursor's coordinates.

handle : device
pstatus : mouse button state : 0 = not pressed; 1 = pressed
x, y : cursor coordinates

```
Procedure vex_butv(handle      : Integer;
                   pusrcode    : Pointer;
                   VAR psavcode : Pointer);
```

Exchange button change vector. This is one of the more „dirty" procedures, as it allows the programmer to specify, what his program should do, if the mouse button is pressed.

handle : device
pusrcode : address of the new routine to be installed
psavcode : address of the old routine

See also : vex_timv, vex_motv, vex_curv

```
Procedure vex_motv(handle      : Integer;
                   pusrcode    : Pointer;
                   VAR psavcode : Pointer);
```

Exchange mouse movement vector. As the above, this is also one of the „dirty" procedures. This time, the user can specify what shall happen, each time the mouse is moved, and the mouse coordinates passed on to the application can be changed as well. „Speed mouse" utilities are often made this way.

handle : device
pusrcode : address of new routine
psavcode : address of old routine

See also : vex_timv, vex_butv, vex_curv

```
Procedure vex_curv(handle      : Integer;
                   pusrcode    : Pointer;
                   VAR psavcode : Pointer);
```

Exchange cursor change vector. In this case, the routine is called, when it is required to change the graphic cursor's position on the screen.

handle : device
pusrcode : address of new routine
psavcode : address of old routine

See also : vex_timv, vex_butv, vex_m

Procedure **vq_key_s(handle : Integer; VAR pstatus : Integer);**

Sample keyboard state information. Returns the status of four special keys on the keyboard, each represented by a single bit :

bit	value	HighSpeed CONST	key
0	1	K_RSHIFT	right SHIFT key
1	2	K_LSHIFT	left SHIFT key
2	4	K_CTRL	CONTROL key
3	8	K_ALT	ALTERNATE key

Set bits indicate a pressed key. A returned value of, say, 3 would indicate that both shift keys are pressed.

Ex. : Determine, if ALTERNATE key is pressed :

```
VAR i : Integer;
...
  vq_key_s(VDI_handle, i);
  IF BitTest(3, i) THEN WriteLn('ALTERNATE pressed')
```

Inquire Functions

These Functions are used for obtaining information on current device settings.

```
Procedure vq_extended(handle, owflag : Integer;
                      VAR workout    : workout_Array);
```

Extended inquire. It's used for determining either the Open Workstation parameters, or the extended parameters. The information about the device (handle) is returned in workout.

```
handle  : device
owflag  : inquiry type : 0 = Open Workstation parameters
                         1 = extended parameters
workout : returned values
```

```
Function vq_color(handle,
                  colorindex,
                  setflag     : Integer;
                  VAR rgb     : Array_3) : Integer;
```

Inquire color representation. Returns the red, green and blue composition of colorindex in rgb in scaled values (0-1000).

```
handle     : device
colorindex : color index
setflag    : 0 = passed color index; 1 = current color index.
rgb        : color composition :

                [0] = red
                [1] = green
                [2] = blue
```

See also : vs_color

```
Procedure vql_attributes(handle     : Integer;
                         VAR attrib : Array_6);
```

Inquire PolyLine attributes.

```
handle : device
attrib : polyline attributes
```

See also : v_pline

```
Procedure vqm_attributes(handle      : Integer;
                         VAR attrib  : Array_5);
```

Inquire PolyMarker attributes.

handle : device
attrib : polymarker attributes

See also : v_pmarker

```
Procedure vqf_attributes(handle      : Integer;
                         VAR attrib : Array_5);
```

Inquire fill attributes.

handle : device
attrib : fill attributes

See also : vr_recfl

```
Procedure vqt_attributes(handle      : Integer;
                         VAR attrib : Array_10);
```

Inquire graphic text attributes.

handle : device
attrib : text attributes

See also : v_gtext, v_justified

```
Procedure vqt_extent(handle      : Integer;
                     chstring    : String80;
                     VAR extent  : Array_8);
```

Inquire graphic text extent. It returns four coordinates, describing an imaginary box surrounding chstring as it would appear with the current text attributes.

handle : device
chstring : input string
extent : extent coordinates

```
Function vqt_width(handle          : Integer;
                   character       : Char;
                   VAR cellwidth,
                       leftdelta,
                       rightdelta : Integer) : Integer;
```
Inquire character cell width.

```
handle     : device
character  : the character to measure
cellwidth  : width of character cell
leftdelta  : left offset
rightdelta : right offset
```

```
Function vqt_name(handle, elementnum : Integer;
                  VAR name           : String80) : Integer;
```
Inquire font name and index.

```
handle       : device
elementnum   : font index
name         : name and style of font
Return value : current font index
```

vqt_name is only useable with GDOS fonts.

See also : vst_load_fonts, vst_unload_fonts, vst_font, vqt_fontinfo

```
Procedure vq_cellarray(handle       : Integer;
                       pxyarray     : Array_4;
                       rowlen,
                       numrows      : Integer;
                       VAR elused,
                           rowsused : Integer;
                       status       : Integer;
                       VAR colarray : intout_Array);
```

Inquire cell array settings.

```
handle   : device
pxyarray : lower left and upper right corners of rectangle
rowlen   : length of each row in color index array
numrows  : number of rows
elused   : number of used elements
rowsused : number of used rows
status   : 0 = error; <> 0 = ok
colarray : returned colors
```

See also : v_cellarray

```
Procedure vqin_mode(handle, dev_type : Integer;
                    VAR inputmode    : Integer);
```

Inquire input mode.

```
handle    : device
dev_type  : device type : Locator, valuator, choice or string
inputmode : request or sample
```

See also : vsin_mode

```
Procedure vqt_fontinfo(handle        : Integer;
                       VAR minADE,
                           maxADE    : Integer;
                       VAR distances : Array_4;
                       VAR maxwidth  : Integer;
                       effects       : Array_3);
```

Inquire information about the current font.

```
handle    : device
minADE    : ASCII value of first writeable character in font
maxADE    : ASCII value of last writeable character in font
distances : bottom, descent, half line, ascent, top values
maxwidth  : char width (left and right delta values are not
            included)
effects   : enlargement by italic etc., left and right delta
            values
```

See also : vst_load_fonts, vst_unload_fonts, vst_font, vqt_name

Escapes

Most of these procedures concern the „alpha" mode, i.e. a non-graphical environment much like the VT-52 terminal. It's a leftover from the IBM PC environment, which doesn't share text and graphics as easy as it's done on the ST. A PC has two screen modes : A mode for graphics and a mode for text.Other procedures include printer handling facilities.

```
Procedure vq_chcells(handle      : Integer;
                     VAR rows,
                         columns : Integer);
```

Inquire addressable alpha character cells. Returns the number of addressable character cells in terms of rows and columns.

```
handle  : device
rows    : number of character rows available
columns : number of character columns available
```

```
Procedure v_exit_cur(handle : Integer);
```

Exit alpha mode and go into graphics mode (if there's a difference).

handle : device

See also : v_enter_cur

```
Procedure v_enter_cur(handle : Integer);
```

Enter alpha mode (leave graphics mode).

handle : device

See also : v_exit_cur

```
Procedure v_curup(handle : Integer);
```

Move alpha cursor one line up.

handle : device

See also : v_curdown, v_curleft, v_curright, v_curhome

Procedure **v_curdown(handle : Integer);**

Move alpha cursor one line down.

handle : device

See also : v_curup, v_curleft, v_curright, v_curhome

Procedure **v_curright(handle : Integer);**

Move alpha cursor one position to the right.

handle : device

See also : v_curup, v_curdown, v_curleft, v_curhome

Procedure **v_curleft(handle : Integer);**

Move alpha cursor one position to the left.

handle : device

See also : v_curup, v_curdown, v_curright, v_curhome

Procedure **v_curhome(handle : Integer);**

Home alpha cursor.

handle : device

See also : v_curup, v_curdown, v_curleft, v_curright

Procedure **v_eeos(handle : Integer);**

Erase from alpha cursor to End Of Screen.

handle : device

See also : v_eeol

Procedure **v_eeol(handle : Integer);**

Erase from alpha cursor to End Of Line.

handle : device

See also : v_eeos

Procedure **vs_curaddress(handle, row, column : Integer);**

Set alpha cursor address to row and column. If values are illegal the cursor is set as close to the desired position as possible.

handle : device
row : row for cursor to be placed on
column : column for cursor to be placed on

See also : vq_curaddress

Procedure **v_curtext(handle : Integer; chstring : String80);**

Write cursor addressable text.

handle : device
chstring : string to output („C" format)

Procedure **v_rvon(handle : Integer);**

Set reverse video on.

handle : device

See also : v_rvoff

Procedure **v_rvoff(handle : Integer);**

Turn reverse video off.

handle : device

See also : v_rvon

Procedure **vq_curaddress(handle : Integer;**
VAR row,
column : Integer);

Return current cursor position.

handle : device
row : cursor's row position
column : cursor's column position

See also : vs_curaddress

Function **vq_tabstatus(handle : Integer) : Integer;**

Return tablet status. A tablet could be a mouse, joystick etc.

handle : device
Return value : tablet status : 1 = available; 0 = not available

Procedure **v_hardcopy(handle : Integer);**

Make a hardcopy.

handle : source device

Procedure **v_dspcur(handle, x, y : Integer);**

Set the graphic cursor at position (x,y) and show it.
Note the difference between a graphic cursor and an alpha cursor.

handle : device
x,y : cursor coordinates

See also : v_rmcur

Procedure **v_rmcur(handle : Integer);**

Remove the last set graphic cursor.

handle : device

See also : v_dspcur

Procedure **v_form_adv(handle : Integer);**

Write a form feed to a printer.

handle : device

See also : v_output_window, v_clear_disp_list, v_bit_image

Procedure **v_output_window(handle : Integer; xyarray : Array_4);**

Write a part of the current output window to a printer.

handle : device
xyarray : two diagonally opposite corners representing the source window

See also : v_form_adv, v_clear_disp_list, v_bit_image

Procedure **v_clear_disp_list(handle : Integer);**

Clear a printer's display list.

handle : device

See also : v_form_adv, v_output_window, v_bit_image

```
Procedure v_bit_image(handle   : Integer;
                      filename : String80;
                      aspect,
                      scaling,
                      num_pts  : Integer;
                      xyarray  : Array_4);

Output bit image file to a printer.

handle   : device
filename : input file to output (remember : append #0)
aspcet   : aspect ratio flag :
             0 = ignore aspect ratio
             1 = pixel aspect ratio
             2 = page aspect ratio
scaling  : scaling flag :
             0 = uniform scaling
             1 = seperate scaling
xyarray  : upper left corner and lower right

See also : v_form_adv, v_output_window, v_clear_disp_list


Function vs_palette(handle, palette : Integer) : Integer;

Select IBM color palette.

handle  : device
palette : selected palette : colors :

          0                  red, green, yellow
          1                  cyan, blue, magenta

See also : vqp_films, vqp_state, vsp_state, vsp_save,
           vsp_message, vqp_error


Procedure vqp_films(handle : Integer; VAR filmnames : String125);

Inquire palette film types (filmnames).

handle    : device
filmnames : film types : Five sets of 25 characters

See also : vs_palette, vqp_state, vsp_state, vsp_save,
           vsp_message, vqp_error
```

```
Procedure vqp_state(handle     : Integer;
                    VAR port,
                        filmname,
                        lightness,
                        interlace,
                        planes     : Integer;
                    VAR indexes    : Array_16);
```

Inquire palette state.

```
handle    : device
port      : communications port : 0 = first
filmname  : film index : 0..4
lightness : lightness control : -3..3
interlace : interlace flag : 0 = noninterlaced; 1 = interlaced
planes    : number of bitplanes
indexes   : coded information about color indexes
```

See also : vs_palette, vqp_films, vsp_state, vsp_save, vsp_message, vqp_error

```
Procedure vsp_state(handle,
                    port,
                    filmnum,
                    lightness,
                    interlace,
                    planes     : Integer;
                    indexes    : Array_16);
```

Set palette state.

```
handle    : device
port      : communications port : 0 = first
filmname  : film index : 0..4
lightness : lightness control : -3..3
interlace : interlace flag : 0 = noninterlaced; 1 = interlaced
planes    : number of bitplanes
indexes   : coded information about color indexes
```

See also : vs_palette, vqp_films, vqp_state, vsp_save, vsp_message, vqp_error

Procedure **vsp_save(handle : Integer);**

Save palette driver state.

handle : device

See also : vs_palette, vqp_films, vqp_state, vsp_state,
vsp_message, vqp_error

Procedure **vsp_message(handle : Integer);**

Suppress palette driver messages.

handle : device

See also : vs_palette, vqp_films, vqp_state, vsp_state, vsp_save,
vqp_error

Function **vqp_error(handle : Integer) : Integer;**

Inquire palette error.

handle : device
Return value : Error code

See also : vs_palette, vqp_films, vqp_state, vsp_state, vsp_save,
vsp_message

Procedure **v_meta_extents(handle,**
minx,
miny,
maxx,
maxy : Integer);

Update current metafile's extent header.

handle : device
minx,
miny,
maxx,
maxy : minimum and maximum coordinates for bounding rectangle

See also : v_write_meta, vm_filename

```
Procedure v_write_meta(handle,
                       numintin : Integer;
                       int_in   : intin_Array;
                       numptsin : Integer;
                       pts_in   : ptsin_Array);
```

Write a user-defined metafile item.

```
handle   : device
numintin : number of elements in int_in
int_in   : user defined information
numptsin : number of elements in pts_in
pts_in   : user defined information
```

See also : v_meta_extents, vm_filename

```
Procedure vm_filename(handle : Integer; filename : String80);
```

Change MetaFile's name.

```
handle   : device
filename : new file name („C" format)
```

See also : v_meta_extents, v_write_meta

GemAES

All GEM AES procedures and functions can be found in this section.

GemAES, like GemDecl and GemVDI, contains a lot of constants, useable for improving program readability :

```
{ EVENT manager definitions : }

                        { Flags for evnt_multi :                          }
MU_KEYBD  = 1;          { Wait for keyboard message                       }
MU_BUTTON = 2;          { Wait for mouse button message                   }
MU_M1     = 4;          { Wait for mouse to enter/leave rectangle 1       }
MU_M2     = 8;          { Wait for mouse to enter/leave rectangle 2       }
MU_MESAG  = 16;         { Wait for a message to be ready in pipe          }
MU_TIMER  = 32;         { Wait for timer message                          }

                        { Keyboard states :                               }
K_RSHIFT = 1;           { Right <Shift> key                               }
K_LSHIFT = 2;           { Left  <Shift> key                               }
K_CTRL   = 4;           { <Control> key                                   }
K_ALT    = 8;           { <Alternate> key                                 }

                        { Message types :                                 }
MN_SELECTED = 10;       { Menu item selected                              }
WM_REDRAW   = 20;       { Window redraw requested                         }
WM_TOPPED   = 21;       { Window topped                                   }
WM_CLOSED   = 22;       { Window close „button" was hit                   }
WM_FULLED   = 23;       { Window full „button" was hit                    }
WM_ARROWED  = 24;       { One of the slider bar arrows was hit            }
WM_HSLID    = 25;       { Horizontal slider moved                         }
WM_VSLID    = 26;       { Vertical slider moved                           }
WM_SIZED    = 27;       { Window re-sized                                 }
WM_MOVED    = 28;       { Window moved                                    }
WM_NEWTOP   = 29;       { New window on top (activated)                   }
AC_OPEN     = 40;       { Accessory opened                                }
AC_CLOSE    = 41;       { Accessory closed                                }

{ FORM Manager definitions }

                        { Form flags (for form_dial) :                    }
FMD_START  = 0;         { Start form (reserve memory)                     }
FMD_GROW   = 1;         { Draw „growing box" image                        }
FMD_SHRINK = 2;         { Draw „shrinking box" image                      }
FMD_FINISH = 3;         { Finish form (release memory)                    }
```

```
{ RESOURCE Manager Definitions }

                    { Resource data structure types :                }
R_TREE       = 0;     { An entire tree                               }
R_OBJECT     = 1;     { An object                                    }
R_TEDINFO    = 2;     { Text info block (TEDINFO)                    }
R_ICONBLK    = 3;     { Icon data block (ICONBLK)                    }
R_BITBLK     = 4;     { Raster block    (BITBLK)                     }
R_STRING     = 5;     { String                                       }
R_IMAGEDATA = 6;      { Image                                        }
R_OBSPEC     = 7;     { Object specification                         }
R_TEPTEXT    = 8;     { Text                                         }
R_TEPTMPLT   = 9;     { Text template                                }
R_TEPVALID   = 10;    { Text validation                              }
R_IBPMASK    = 11;    { Icon display mask                            }
R_IBPDATA    = 12;    { Icon bit map                                 }
R_IBPTEXT    = 13;    { Icon text string                             }
R_BIPDATA    = 14;    { Pointer to bit mapped graphics               }
R_FRSTR      = 15;    { Address of pointer to free string            }
R_FRIMG      = 16;    { Address of pointer to free image             }

{ WINDOW Manager Definitions }

                      { Window elements :                            }
NAME          =    1; { Window name field                            }
CLOSER        =    2; { Close fields                                 }
FULLER        =    4; { Full field                                   }
MOVER         =    8; { Moveable                                     }
INFO          =   16; { Info line                                    }
SIZER         =   32; { Size field                                   }
UPARROW       =   64; { UpArrow at vertical slider field             }
DNARROW       =  128; { DownArrow at vertical slider field           }
VSLIDE        =  256; { Vertical slider field                        }
LFARROW       =  512; { LeftArrow at horizontal slider field         }
RTARROW       = 1024; { RightArrow at horizontal slider field        }
HSLIDE        = 2048; { Horizontal slider                            }

                     {wind_calc flags :                              }
WC_BORDER     =    0; {Calculate border area from known work area}
WC_WORK       =    1; {Calculate work area from known border area}

                      { wind_get flags                               }
WF_KIND       =    1;  { Window kind                                 }
WF_NAME       =    2;  { Window name                                 }
WF_INFO       =    3;  { Window info                                 }
WF_WORKXYWH   =    4;  { Window work area x, y, w, h                 }
WF_CURRXYWH   =    5;  { Current border area x, y, w, h              }
WF_PREVXYWH   =    6;  { Previous border area x, y, w, h             }
```

```
WF_FULLXYWH =    7;  { Max border area size x, y, w, h        }
WF_HSLIDE   =    8;  { Horizontal slider position [0..1000]   }
WF_VSLIDE   =    9;  { Vertical slider position [0..1000]     }
WF_TOP      =   10;  { Active window's handle                 }
WF_FIRSTXYWH =   11;  { First rectangle in rectangle list      }
WF_NEXTXYWH =   12;  { Next rectangle in rectangle list       }
WF_RESVD    =   13;  { Reserved!                              }
WF_NEWDESK  =   14;  { New desktop background object tree    }
WF_HSLSIZE  =   15;  { Relative size of horizontal slider     }
WF_VSLSIZE  =   16;  { Relative size of vertical slider       }
WF_SCREEN   =   17;

                     { wind_update flags                      }
END_UPDATE  =    0;  { End update                            }
BEG_UPDATE  =    1;  { Begin update                          }
END_MCTRL   =    2;  { End mouse control                     }
BEG_MCTRL   =    3;  { Begin mouse control                   }

{ GRAPHICS Manager Definitions }

                     { Mouse Forms :                          }
ARROW       =    0;  { The normal arrow                      }
TEXT_CRSR   =    1;  { Text cursor                           }
BUSYBEE     =    2;  { The busy bee                          }
HOURGLASS   =    2;  { Same as BUSYBEE                       }
POINT_HAND  =    3;  { Pointing hand                         }
FLAT_HAND   =    4;  { Flat hand                             }
THIN_CROSS  =    5;  { Thin cross                            }
THICK_CROSS =    6;  { Thick cross                           }
OUTLN_CROSS =    7;  { Outlined cross                        }
USER_DEF    =  255;  { User defined mouse pattern            }
M_OFF       =  256;  { Turn mouse off                        }
M_ON        =  257;  { Turn mouse on                         }

                     { Graphic objects :                      }
G_BOX       =   20;  { A box                                 }
G_TEXT      =   21;  { Text                                  }
G_BOXTEXT   =   22;  { Text within a box                     }
G_IMAGE     =   23;  { Raster image                          }
G_USERDEF   =   24;  { User defined object (could be code!)  }
G_IBOX      =   25;  { Invisible (transparent) box           }
G_BUTTON    =   26;  { Centred text in a box                 }
G_BOXCHAR   =   27;  { Char within a box                     }
G_STRING    =   28;  { Text (menu item)                      }
G_FTEXT     =   29;  { Formatted text                        }
G_FBOXTEXT  =   30;  { Text within a box                     }
G_ICON      =   31;  { Icon                                  }
G_TITLE     =   32;  { Text (menu title)                     }
```

```
                        { Object flags :                                }
NONE            =    0; { No flags                                       }
SELECTABLE      =    1; { Object is selectable                           }
DEFAULT         =    2; { Object is default                              }
F_EXIT          =    4; { Object causes exit                             }
EDITABLE        =    8; { Object is editable                             }
RBUTTON         =   16; { Object is radiobutton                          }
LASTOB          =   32; { Object is last in tree                         }
TOUCHEXIT       =   64; { Object touch causes exit                       }
HIDETREE        =  128; { Hide subordinate objects                       }
INDIRECT        =  256; { Object specification is a pointer              }

                        { Object states :                               }
NORMAL          =    0; { Normal state                                   }
SELECTED        =    1; { Object is selected                             }
CROSSED         =    2; { Object is crossed                              }
CHECKED         =    4; { Object is checked                              }
DISABLED        =    8; { Object is disabled                             }
OUTLINED        =   16; { Object is outlined                             }
SHADOWED        =   32; { Object is shadowed                             }

                        { Object colour numbers :                       }
White           =    0; Black        =    1;
Red             =    2; Green        =    3;
Blue            =    4; Cyan         =    5;
Yellow          =    6; Magenta      =    7;
LWhite          =    8; LBlack       =    9;
LRed            =   10; LGreen       =   11;
LBlue           =   12; LCyan        =   13;
LYellow         =   14; LMagenta     =   15;

                        { Editable text field definitions :             }
EDSTART = 0;            { Reserved
EDINIT  = 1;            { Mix text and template and turn cursor on}
EDCHAR  = 2;            { Test and insert char if char allowed          }
EDEND   = 3;            { Remove edit cursor                            }

                        { Editable text justification :                 }
TE_LEFT  = 0;           { Left justified                                }
TE_RIGHT = 1;           { Right justified                               }
TE_CNTR  = 2;           { Centred                                       }

                      { Scancodes for evnt_multi and evnt_keybd : }
BackSpace  = $0E08;  Tab        = $0F09;
S_Delete   = $537F;  S_Insert   = $5200; { S_ doesn't kill      }
Shift_Ins  = $5230;  Return     = $1C0D; { string procedures    }
Enter      = $720D;  Undo       = $6100;
```

```
Help        = $6200;  Home        = $4700;
Cur_Up      = $4800;  Cur_Down    = $5000;
Cur_Left    = $4B00;  Cur_Right   = $4D00;
Shift_Home  = $4737;  Shift_CU    = $4838;
Shift_CD    = $5032;  Shift_CL    = $4B34;
Shift_CR    = $4D36;  Esc         = $011B;
Ctrl_A      = $1E01;  Ctrl_B      = $3002;
Ctrl_C      = $2E03;  Ctrl_D      = $2004;
Ctrl_E      = $1205;  Ctrl_F      = $2106;
Ctrl_G      = $2207;  Ctrl_H      = $2308;
Ctrl_I      = $1709;  Ctrl_J      = $240A;
Ctrl_K      = $250B;  Ctrl_L      = $260C;
Ctrl_M      = $320D;  Ctrl_N      = $310E;
Ctrl_O      = $180F;  Ctrl_P      = $1910;
Ctrl_Q      = $1011;  Ctrl_R      = $1312;
Ctrl_S      = $1F13;  Ctrl_T      = $1414;
Ctrl_U      = $1615;  Ctrl_V      = $2F16;
Ctrl_W      = $1117;  Ctrl_X      = $2D18;
Ctrl_Y      = $2C19;  Ctrl_Z      = $151A;
Ctrl_1      = $0211;  Ctrl_2      = $0300;
Ctrl_3      = $0413;  Ctrl_4      = $0514;
Ctrl_5      = $0615;  Ctrl_6      = $071E;
Ctrl_7      = $0817;  Ctrl_8      = $0918;
Ctrl_9      = $0A19;  Ctrl_0      = $0B10;
Alt_A       = $1E00;  Alt_B       = $3000;
Alt_C       = $2E00;  Alt_D       = $2000;
Alt_E       = $1200;  Alt_F       = $2100;
Alt_G       = $2200;  Alt_H       = $2300;
Alt_I       = $1700;  Alt_J       = $2400;
Alt_K       = $2500;  Alt_L       = $2600;
Alt_M       = $3200;  Alt_N       = $3100;
Alt_O       = $1800;  Alt_P       = $1900;
Alt_Q       = $1000;  Alt_R       = $1300;
Alt_S       = $1F00;  Alt_T       = $1400;
Alt_U       = $1600;  Alt_V       = $2F00;
Alt_W       = $1100;  Alt_X       = $2D00;
Alt_Y       = $2C00;  Alt_Z       = $1500;
Alt_1       = $7800;  Alt_2       = $7900;
Alt_3       = $7A00;  Alt_4       = $7B00;
Alt_5       = $7C00;  Alt_6       = $7D00;
Alt_7       = $7E00;  Alt_8       = $7F00;
Alt_9       = $8000;  Alt_0       = $8100;
F1          = $3B00;  F2          = $3C00;
F3          = $3D00;  F4          = $3E00;
F5          = $3F00;  F6          = $4000;
F7          = $4100;  F8          = $4200;
F9          = $4300;  F10         = $4400;
Shift_F1    = $5400;  Shift_F2    = $5500;
```

```
Shift_F3    = $5600;  Shift_F4    = $5700;
Shift_F5    = $5800;  Shift_F6    = $5900;
Shift_F7    = $5A00;  Shift_F8    = $5B00;
Shift_F9    = $5C00;  Shift_F10   = $5D00;

                      { Language dependent scan codes :          }
Ctrl_AE     = $2804;  Ctrl_OE     = $2714;
Ctrl_UE     = $1A01;  Alt_AE      = $285D;
Alt_OE      = $275B;  Alt_UE      = $1A40;
SH_Alt_AE   = $287D;  SH_Alt_OE   = $277B;
SH_Alt_UE   = $1A5C;
```

General Procedures

```
Function GemError : Integer;

GemError returns the value of intout[0] which GEM uses to report
error messages most of the time.
See also : GemDecl unit
```

Application library

The routines contained in the application library are capable of performing the following tasks :

- initialization and de-init. of application programs
- simple pipe handling
- recording of user actions

```
Function appl_init : Integer;

Initialize application and return application's identification.

See also : appl_read, appl_write, appl_find, appl_tplay,
           appl_trecord, appl_exit


Procedure appl_read(ap_rid, ap_rlength : Integer;
                    ap_rpbuff          : Pointer);

Read a number of bytes from the event buffer.

ap_rpbuff  : event buffer to read from
ap_rid     : receiving application's id
ap_rlength : number of bytes to read

See also : appl_init, appl_write, appl_find, appl_tplay,
           appl_trecord, appl_exit
```

```
Procedure appl_write(ap_wid, ap_wlength : Integer;
                     ap_wpbuff          : Pointer);
```

Write a number of bytes to the event buffer.

```
ap_wpbuff  : event buffer to store in
ap_wid     : the receiving application's id
ap_wlength : number of bytes to read
```

See also : appl_init, appl_read, appl_find, appl_tplay, appl_trecord, appl_exit

```
Function appl_find(ap_fpname : Pointer) : Integer;
```

Find the application id of another application in the system.

```
ap_fpname    : points to a string („C" format, null terminated)
               which have to be eight characters long, i.e. it
               might be necessary to fill up with spaces. The
               string is the application's file name
Return value : application's id or -1 if not found
```

See also : appl_init, appl_read, appl_write, appl_tplay, appl_trecord, appl_exit

```
Procedure appl_tplay(ap_tpmem              : Pointer;
                     ap_tpnum, ap_tpscale : Integer);
```

The AES is capable of recording user actions and „replaying" them. The latter is exactly what appl_tplay does : Replays a piece of a recording.

```
ap_tpmem   : address of recorded data
ap_tpnum   : number of actions to play
ap_tpscale : play back speed : Range 0..10000 :

                 0 = slowest speed
             10000 = highest speed
               100 = normal speed
```

See also : appl_init, appl_read, appl_write, appl_find, appl_trecord, appl_exit

```
Function appl_trecord(ap_trmem   : Pointer;
                      ap_trcount : Integer) : Integer;

Records user actions.

ap_trmem     : address of recording buffer
ap_trcount   : max number of recordings, the buffer can hold =
               buffer size DIV 6 (size of a recording)
Return value : the number of actions recorded

See also : appl_init, appl_read, appl_write, appl_find,
           appl_tplay, appl_exit
```

```
Procedure appl_exit;

Exit application. Should be done before returning to the GEM
Desktop.

See also : appl_init, appl_read, appl_write, appl_find,
           appl_tplay, appl_trecord
```

Event library

The AES views upon all AES-related actions made by the user and two internal action as events. The event library presents procedures designed to wait for one or more events.

Before reading on, note the difference between a (keyboard) key and a (mouse) button!

```
Function evnt_keybd : Integer;

Wait for keypress and return key's code.

See also : evnt_button, evnt_mouse, evnt_mesag, evnt_timer,
           evnt_multi, evnt_dclick
```

```
Function evnt_button(ev_bclicks, ev_bmask,
                     ev_bstate              : Integer;
                     VAR ev_bmx, ev_bmy,
                         ev_bmb, ev_bks     : Integer) : Integer;
```

Wait for one or more mouse buttons to be pressed.
GEM allows a mouse to have as many as 16 buttons, where each bit in ev_bmb represents a button. A set bit represents a pressed button.

```
ev_bclicks : the number of mouse clicks GEM should look for, most
             often 1 or 2 = double click.
ev_bmask   : the keys allowed to generate the event. A set bit
             represents a „legal" key.
ev_bstate  : whether the in ev_bmask given keys should be :
                0 = released or
                1 = pressed
             in order to generate the event.
ev_bmb     : returned mouse button state
ev_bks     : keyboard state (CTRL, ALT, etc.)
ev_bmx,
ev_bmy     : mouse position at time of the „click"

See also : evnt_keybd, evnt_mouse, evnt_mesag, evnt_timer,
           evnt_multi, evnt_dclick
```

```
Procedure evnt_mouse(ev_moflags, ev_mox, ev_moy,
                     ev_mowidth, ev_moheight     : Integer;
                     VAR ev_momx, ev_momy,
                         ev_momb, ev_momks       : Integer);
```

Wait for the mouse to enter or leave a rectangle.

```
ev_moflags  : entry/exit flag : 0 = leave; 1 = enter rect.
ev_mox,
ev_moy      : rectangle specification : upper left corner
ev_mowidth,
ev_moheight : rectangle specification : height and width
ev_momx,
ev_momy     : mouse position at event-time
ev_momb     : mouse button state at time of event
ev_momks    : keyboard state at time of event

See also : evnt_keybd, evnt_button, evnt_mesag, evnt_timer,
           evnt_multi, evnt_dclick
```

Procedure **evnt_mesag(ev_mgpbuff : Pointer);**

Wait for a report to be present in the message pipe.

ev_mgpbuff : message pipe

See also : evnt_keybd, evnt_button, evnt_mouse, evnt_timer,
evnt_multi, evnt_dclick

Procedure **evnt_timer(ev_tlocount, ev_thicount : Integer);**

Wait for a number of milliseconds given by the four-byte value formed by ev_thicount and ev_tlocount.

ev_tlocount : low word
ev_thicount : high word

Total period time = ev_thicount * 2^16 + ev_tlocount milliseconds

Ex. :

```
VAR time : LongInt;
...
  Write('How many milliseconds? ');  ReadLn(time);
  evnt_timer(loword(time), hiword(time));
  WriteLn('Here we are again...');
```

See also : evnt_keybd, evnt_button, evnt_mouse, evnt_mesag,
evnt_multi, evnt_dclick

```
Function evnt_multi
  (ev_mflags, ev_mbclicks,
   ev_mbmask, ev_mbstate,
   ev_mm1flags, ev_mm1x,
   ev_mm1y, ev_mm1width,
   ev_mm1height,
   ev_mm2flags, ev_mm2x,
   ev_mm2y, ev_mm2width,
   ev_mm2height                    : Integer;
   ev_mmgpbuff                     : Pointer;
   ev_mtlocount, ev_mthicount      : Integer;
   VAR ev_mmox, ev_mmoy,
       ev_mmobutton, ev_mmokstate,
       ev_mkreturn, ev_mbreturn    : Integer ) : Integer;
```

This is one of the more tough ones! Unlike the other event procedures, evnt_multi allows for any kind of events to occur.
The allowed events are given via ev_mflags :

ev_mflags	HighSpeed CONST	event type
1	MU_KEYBD	wait for keyboard event
2	MU_BUTTON	wait for mouse button event
4	MU_M1	wait for mouse to enter/leave one of two rectangles
8	MU_M2	wait for mouse to enter/leave the second of the two rectangles
16	MU_MESAG	wait for event message to be present
32	MU_TIMER	wait for timer event

It is suggested for the reader to scan through the other event-procedures, as they give additional information about the parameters to evnt_multi.

```
ev_mbclicks    : mouse button clicks required
ev_mbmask      : mouse button mask
ev_mbstate     : mouse button state
ev_mm1flags    : rectangle 1 entry/exit flag
ev_mm1x,
ev_mm1y,
ev_mm1width,
ev_mm1height   : rectangle 1

The same layout applies to rectangle 2.

ev_mmgpbuff    : message pipe address
ev_mtlocount,
ev_mthicount   : time in milliseconds
ev_mmox,
ev_mmoy        : mouse (x,y) at event-time
ev_mmobutton   : mouse button pressed
ev_mmokstate   : keyboard state (SHIFT, CTRL etc.)
ev_mkreturn    : keyboard key pressed
ev_mbreturn    : number of mouse clicks
Return value   : „occurred-events" flag
```

evnt_multi is one of the vital functions in a „normal" GEM application using menus, windows, dialog boxes etc. as it allows the user to generate a multitude of events, which the application then can dispatch easily.

See also : evnt_keybd, evnt_button, evnt_mouse, evnt_mesag, evnt_timer, evnt_dclick

```
Function evnt_dclick(ev_dnew, ev_dgtest : Integer) : Integer;

Set double click speed.

ev_dnew       : new value or dummy
ev_dgtest     : set new speed or return current speed value?
                  0 = return current (ev_dnew is dummy)
                  1 = set new
Return value : current speed/set speed

Ex. :

VAR
  currentspeed : Integer;
...
  currentspeed := evnt_dclick(0, 0);  { return current speed }

See also : evnt_keybd, evnt_button, evnt_mouse, evnt_mesag,
           evnt_timer, evnt_multi
```

Menu library

This library presents procedures for menu handling, i.e. drawing, selecting, checking etc.

```
Procedure menu_bar(me_btree : Pointer; me_bshow : Integer);

Display or erase a menu bar. Should always be called before
appl_exit.

me_btree : menu tree
me_bshow : show/erase flag : 0 = erase; 1 = show

See also : menu_icheck, menu_ienable, menu_tnormal, menu_text,
           menu_register
```

Compiler directives

HighSpeed Pascal can be controlled by using compiler directives, which are commands not defined in standard Pascal. Therefore they are placed in specially formed comments.

A directive must start with a $ dollar sign right after the starting bracket, as in '{$' or '(*$', followed by the command, which can be one or more letter(s). Multi-letter directives are used for conditional compilation.

The different directives have default values, either set by the command line version by the given commands , or for the integrated version by filling the different dialogues in the options menu.

Switch directives

Switch directives are written by using a single letter followed by a '+' or a '-'. The dialogues show this as a check-mark next to the '$X' text.

A '+' (or a check-mark in the dialogue) enables the feature.

Several options can be given in one comment as in (*$R+,S-*).

Debug Information {$D+}

{$D+} enables generation of debug information in the generated program file. The names saved are for all procedures and functions used.

Far Calls {$F+}

{$F+} enables the program to be greater than 32 KByte. $F makes all procedure and function calls use 32 bit if needed, instead of 16.bit. Each unit may have to be compiled with this flag. In Turbo-Pascal for the PC, it is fatal to forget using $F, because the information saved on the stack depends on the $F setting. On the 68000, the return address is always saved as 32 bit. The $F flag in HighSpeed Pascal only helps the compiler to make the best decision when making calls.

IOResult Checking {$I+}

{$I+} enables automatic input/output checking. An IO error will terminate the program. By using the $I- option, you will have to test the IOResult function yourself after each IO operation.

Range Checking {$R+}

{$R+} enables range checking on simple types. Every assignment to simple types is then checked, and every use of indexing in an array or in a string is also checked to be within legal range.

Stack Checking {$S+}

{$S+} enables stack checking. Stack checking is done as the very first thing in each procedure and function that is compiled using the $S+ flag.

VAR String checking {$V+}

{$V+} enables the (normal) var-string checking. By using the $V- flag, you can pass strings of any length to a procedure, that has a string variable as a VAR parameter. It is up to you to make sure the string passed is not overwritten with more than the parameter can hold.

Other single letter directives

Memory Size Layout {$M stack,hmin,hmax,dosfree}

{$M stack,hmin,hmax,dosfree} sets the default memory allocation requests. $M does not work in a unit.

Each of the 4 parameters specifies the number of Kilobytes wanted.

Stack is the request for stack space, minimum 1
hmin is the minimum request for heap space, minimum 1.
hmax is the maximum request for heap space.
dosfree is the amount of memory, that must be left free when the program is started.

Include Files {$I FileName} {$I FileName}

{$I FileName} makes the compiler include the named file into the source . The file is searched for using the path(s) specified in the Compiler-Option dialogue or on the command line in the -I command.

Object Files {$L FileName}

{$L FileName} makes the compiler include the named object file when all declaration has been done. The EXTERNAL procedures and functions areassumed to be in a .O file. You can specify several $L directives, one for each object file. The object file is searched for by using the path(s) specified in the Compiler-Option dialogue or on the command line in the -Ornand.

Source code control

The compiler maintains some variables, that are not part of the program being compiled. They can be used to control the way the compiler works, by enabling or disabling the compilation of certain parts of the source.

These 3 variables are defined in this version: Atari, 68000 and Ver10.
The last one is valid only in release number 1.0.

The variables does not contain any value, just test if they are there or not.

New variables are defined with DEFINE, and removed by using the UNDEF directive.

The test for their presence is done with either IFDEF (for IF Defined THEN) or IFNDEF (for IF Not Defined THEN).

The compiling state can be changed using the ELSE directive.
An ENDIF directive ends the $IFxx sequence.

Finally you can test if a switch option is set by using the IFOPT directive.

Examples of the use of conditional compilation.

```
Unit OutStuff;
{$IFOPT R+}
  {$Define RangeCheck}
{$Else}
  {$UnDef RangeCheck}
{$EndIF}
begin
{$IfDef CPU86}
 N:=Swap(N);  {Swap bytes if using the 80x86 family}
{$EndIF}

{$IfDef DebugMode}
  writeln('Debug=> N is now: ',N);
{$EndIF}
end.
```

Examples:

```
{$R-,S+,M 5,10,10,20}
{$S- Disable stack checking}
{$R+ Make sure range checking is enabled}
```

Reference

Abs function

Function Abs(N) : (Same type as parameter) — Declaration

Returns the absolute value of the parameter N. — Function

N is an integer-type or a real-type expression. — Remark

Int,Trunc,Round — See also

Example

```
Program Abs_Demo;

Const
        Negative = -1000;
        Positive =  1000;
Begin
  Writeln(Negative,' becomes ',ABS(Negative));
  Writeln(Positive,' is still ',ABS(Positive));
End.
```

Addr function

Function Addr(var Object) : Pointer; — Declaration

Returns the address of the specified object. — Function

The Object parameter can be a variable or a procedure or function identifier. — Remark

SPtr,Ptr — See also

Example

```
Program Addr_Demo;

Var
        A : Pointer;
        L : LongInt;
Begin
  A := Addr(L);
  { Another way of doing it is: }
  A := @L;
End.
```

Append procedure

Declaration Procedure Append(var F : text [;Name : String [;BufSize : LongInt]]);

Function Prepares a text file for appending.

Remark The Append procedure can be used in 3 different ways:

Append(MyFile, 'NAME.EXT');
Does the same as ReWrite, but positions the file pointer at the end of the file.
Append(MyFile, 'NAME.EXT', 10000);
Does the same as the above, but also creates a 10000 byte buffer.
Append(MyFile);
Positions the file pointer at the end of an already opened file.

See also ReWrite,Reset

Example

```
Program Append_Demo;
Var
        DataFile : Text;
        X        : Byte;
Begin
  { Create a temporary file. }
  ReWrite(DataFile,'TEMP.DTA');
  For X := 1 to 100 DO
    Writeln(DataFile,'HELLO WORLD');
  Close(DataFile);
  { Now let's append some data. }
  Append(DataFile,'TEMP.DTA');
  Writeln(DataFile,'Last line in file');
  Close(DataFile);
End.
```

ArcTan function

Declaration Function ArcTan(N) : Real;

Function Returns the arctangent of the parameter N.

Remark N is an integer-type or real-type expression. The result is the principal value, in radians, of the argument of N.

See also ValidReal

Example

```
Program ArcTan_Demo;
Var
        Res : Real;
Begin
  Res := ArcTan(123);
End.
```

BlockRead procedure

Procedure BlockRead(var F,Buf; Cnt : LongInt [; var Res : Integer]); — Declaration

Reads Cnt bytes from the file F into the variable Buf. — Function

F is a file variable (Typed or Untyped) that has been opened. Cnt specifies the number of bytes to read from the file. The actual number of bytes read will be returned in the optional Res parameter. — Remark

If Res is not specified, and Cnt bytes could not be read, an IO-error will occur. Otherwise the number of bytes actually read is returned in Res.

BlockWrite,Seek — See also

Example

```
Program BlockRead_Demo;

Uses DOS;

Var
        F_In,
        F_Out           : FILE; { Untyped files }
        Buf             : Array[1..4096] OF Byte; {
Disk buffer }
        ActualRead,
        ActualWritten : Integer;
Begin
  If (ParamCount <> 2) then HALT;
  Reset(F_In,ParamStr(1));
  {$I-} Erase(ParamStr(2)); {$I+}
  ReWrite(F_Out,ParamStr(2));
  Repeat { Copy file In to file Out. }
    BlockRead(F_In,Buf,SizeOf(Buf),ActualRead);
    BlockWrite(F_Out,Buf,ActualRead,ActualWritten);
  Until (ActualRead = 0) OR (ActualWritten <>
ActualRead);
  Close(F_In);
  Close(F_Out);
End.
```

BlockWrite procedure

Declaration Procedure BlockWrite(var F,Buf; Cnt : LongInt [; var Res : Integer]);

Function Writes Cnt bytes from the variable Buf to the file F.

Remark F is a file variable (Typed or Untyped) that has been opened. Cnt specifies the number of bytes to write to the file. The actual number of bytes written will be returned in the optional Res parameter.

If Res is not specified, and Cnt bytes could not be written, an IO-error will occur. Otherwise the number of bytes actually written is returned in Res.

See also BlockRead,Seek

ChDir procedure

Declaration Procedure ChDir(Dir : DirStr);

Function Changes the current directory.

Remark It is not possible to change to another drive using this procedure. To do that use the SetDrive procedure.

See also RmDir,MkDir,GetDir,SetDrive,DosError,Dos

Example

```
Program ChDir_Demo;
Uses DOS;

Begin
  ChDir('C:\MYDIR');
End.
```

Chr function

Declaration Function Chr(N) : Char;

Function Returns ASCII character number N.

Remark N is an integer type value in the range of 0..255.

See also Ord,Ord4

Example

```
Program Chr_Demo;
Var
        B : Byte;
Begin
  For B := 32 to 255 DO
    Write(CHR(B):4);
End.
```

Close procedure

Procedure Close(var F);

Declaration

Closes an open file.

Function

F is a file variable of any type. Close flushes the files buffer and releases its handle.

Remark

Reset,ReWrite,Append

See also

ClrEol procedure

Procedure ClrEol;

Declaration

Deletes all characters on the current line, starting at the current cursor position.

Function

The position of the cursor is not changed by the ClrEol procedure.

Remark

ClrScr,ClrEos

See also

Example

```
Program ClrEol_Demo;

Var
        Y : Byte;
Begin
  ClrScr;
  For Y := 1 to 10 DO
    Writeln('0123456789');
  For Y := 1 to 10 DO
    Begin
      GoToXY(5,Y); ClrEol;
    End;
End.
```

ClrEos procedure

Declaration Procedure ClrEos;

Function Clears the screen starting at the current cursor location.

Remark The position of the cursor is not changed by the ClrEos procedure.

See also ClrScr,ClrEol

Example

```
Program ClrEos_Demo;

Var
        X : Integer;
Begin
  For X := 1 to 1999 DO
    Begin
      GoToXY(Random(80),Random(25)+1);
      Write('*');
    End;
  GoToXY(1,10);
  ClrEos;
End.
```

ClrScr procedure

Declaration Procedure ClrScr;

Function Clears the screen.

Remark After the call to ClrScr the cursor is placed in the upper left corner of the screen (Coordinates 1,1).

See also ClrEol,ClrEos

Concat function

Function Concat(S1 [, S2, S3 ...]) : String; — Declaration

Concatenates 1 or more strings into 1 string. — Function

Using standard string addition, as for example:

```
S1 := S1 + S2;
```

will give the same result as:

```
S1 := Concat(S1,S2);
```

Copy,Insert,Delete — See also

Example

```
Program Concat_Demo;

Var
        A1,A2,Res : String;
Begin
  A1 := 'HELLO '; A2 := 'WORLD';
  Res := Concat(A1,A2);
  Writeln(Res);
End.
```

Copy function

Function Copy(FromStr : String; Index, Count : Byte) : String; — Declaration

Returns Count characters from FromStr, starting at position Index. — Function

Copy returns an empty string if index is 0 or greater than the length of FromStr. If Index+Count is greater than the length of FromStr, the Copy function stops at the end of FromStr and returns. — Remark

Concat,Insert,Delete,Length,Pos — See also

Example

```
Program Copy_Demo;

Const
        Main = 'EXTRACT THE WORD HELLO FROM THIS
STRING';
Begin
  Writeln(Copy(Main,18,5));
End.
```

Cos function

Declaration Function Cos(N) : Real;

Function Returns the cosine of the parameter.

Remark N is an integer-type or real-type expression. N is assumed to represent an angle in radians.

See also Sin,Trunc,ValidReal

Example

```
Program Cos_Demo;

Var
        R : Real;
Begin
  R := COS(20);
End.
```

Dec procedure

Declaration Procedure Dec(N [, Count]);

Function Decrements an ordinal variable by 1 or by Count.

Remark N and the optional parameter Count are both simple types. If Count is not specified N will be decremented by 1. Otherwise N will be decremented by Count.

See also Inc,Succ,Pred

Example

```
Program Dec_Demo;

Var
        L : LongInt;
Begin
  L := 100000;
  Repeat
    Dec(L);     { Decrease L by 1 }
    Dec(L,2);   { Decrease L by 2 }
  Until (L < 0);
End.
```

Delay procedure

Procedure Delay(Ms : LongInt); — Declaration

Halts machine execution for Ms milliseconds. — Function

Example

```
Program Delay_Demo;

Begin
  Writeln('Going to sleep for 5 seconds.');
  Delay( 5*1000 );
End.
```

Delete procedure

Procedure Delete(var S : String; Index, Count : Byte); — Declaration

Deletes Count characters from the string S, starting at position Index. — Function

If Index is 0 or greater then the length of S, no characters are deleted from S. — Remark

Omit,Copy,Insert,Length,Pos — See also

Example

```
Program Delete_Demo;

Var
        S : String;
Begin
  S := 'DELETE THE WORD HELLO FROM THIS STRING';
  Writeln('Before calling delete: ',s);
  Delete(S,17,6);
  Writeln('After calling delete : ',S);
End.
```

DelLine procedure

Declaration Procedure DelLine;

Function Deletes the current line from the screen.

Remark All lines following the current line will be scrolled up, and an empty line inserted at the bottom of the screen.

See also InsLine,GotoXY

Example

```
Program DelLine_Demo;

Var
      X,Y : Integer;
Begin
  For X := 1 to 80 DO
    For Y := 1 to 24 DO
      Begin
        GoToXY(X,Y); Write('*');
      End;
  GoToXY(30,10);
  Writeln('Press ENTER to go on');
  Readln;
  GoToXY(1,1);
  For Y := 1 to 24 DO
    DelLine;
End.
```

DiskFree function

Function DiskFree(Drive : Byte) : LongInt; — Declaration

Returns the number of available bytes on the disk in the specified drive. — Function

Drive numbers are ordered as follows: — Remark

drive 0 = Current disk drive.
- 1 = Drive A:
- 2 = Drive B:
....

DiskSize,GetDrive,SetDrive,Dos — See also

Example

```
Program DiskFree_Demo;

Uses DOS;

Begin
  Writeln( DiskFree(0) DIV 1024,' Kb available on
current disk.');
End.
```

DiskSize function

Function DiskSize(Drive : Byte) : LongInt; — Declaration

Returns the size of the disk in the specified drive. — Function

Drive numbers are ordered as follows: — Remark

drive 0 = Current disk drive.
- 1 = Drive A:
- 2 = Drive B:
....

DiskFree,GetDrive,SetDrive,Dos — See also

Example

```
Program DiskSize_Demo;

Uses DOS;

Begin
  Writeln( DiskSize(0) DIV 1024,' Kb available on
current disk.');
End.
```

Dispose procedure

Declaration

Procedure Dispose(var P : Pointer);

Function

Disposes a dynamic variable.

Remark

P is a pointer variable of any type that was previously assigned by the New procedure or was assigned a value by an assignment statement.
After a call to Dispose the variable referenced by P is destroyid and its space in the heap is released.

See also

New,FreeMem,GetMem

Example

```
Program Dispose_Demo;

Type
        PersonType = Record
                       Name    : String[20];
                       Age     : Byte;
                     End;
Var
        PersonData : ^PersonType;
Begin
  New(PersonData); { Allocate space in heap }
  With PersonData^ DO
    Begin
      Write('Enter name: '); ReadLn(Name);
      Write('Enter age : '); ReadLn(Age);
    End;
  Dispose(PersonData); { Dispose the space }
End.
```

DosError variable

Var DosError : Integer;

Declaration

Returns the error status for some of the routines in the DOS unit.

Function

The DosError values are declared in the DOS unit:

Remark

```
Const
        EINVFN   = -32;   { Invalid Function Number          }
        EFILNF   = -33;   { File Not Found                   }
        EPTHNF   = -34;   { Path Not Found                   }
        ENHNDL   = -35;   { File Handle Pool Exhausted       }
        EACCDN   = -36;   { Access Denied                    }
        EIHNDL   = -37;   { Invalid Handle                   }
        ENSMEM   = -39;   { Insufficient Memory              }
        EIMBA    = -40;   { Invalid Memory Block Address     }
        EDRIVE   = -46;   { Invalid Drive Specification      }
        ENSAME   = -48;
        ENMFIL   = -49;   { No More Files                    }
        ERANGE   = -64;   { Range Error                      }
        EINTRN   = -65;   { GEMDOS Internal Error            }
        EPLFMT   = -66;   { Invalid Executable File format   }
        EGSBF    = -67;   { Memory Block Growth Failure      }
```

Dos,BiosErrors,RuntimeErrors

See also

EnvCount function

Declaration

Function EnvCount : Integer;

Function

Returns the number of environment strings in the systems environment block.

Remark

Use the functions EnvStr and GetEnv to examine the contents of the environment block.

See also

EnvStr,GetEnv,Dos

Example

```
Program EnvCount_Demo;

Uses DOS;

Begin
  Writeln(EnvCount,' strings in the environment
block.');
End.
```

EnvStr function

Declaration Function EnvStr(Index : Integer) : String;

Function Returns the environment string number Index.

Remark If Index is 0 or greater than EnvCount, EnvStr will return an empty string.

See also EnvCount,GetEnv,Dos

Example

```
Program EnvStr_Demo;

Uses DOS;

Var
        Count : Integer;
Begin
  Writeln('Program to display the environment
block.');
  Writeln;
  For Count := 1 to EnvCount DO
    Writeln( EnvStr(Count) );
  Writeln;
  Writeln('Done.');
End.
```

Eof function

Function Eof [(var F)] : Boolean; — Declaration

Returns false if there is still data to read after the current file position. — Function

F is a file variable of any type, that has been opened. If F is omitted the standard file Input is assumed. — Remark

SeekEof,Eoln — See also

Example

```
Program Eof_Demo;

Uses DOS;

Var
        T : Text;
        D : String;
Begin
  {$I-} Reset(T, 'TEMP.TXT'); {$I+}
  If (IOresult = 0) then
    Begin
      While NOT(Eof(T)) DO
        Begin
          ReadLn(T,D); Writeln(D);
        End;
      Close(T);
    End
  ELSE
   Writeln('Error opening file.');
End.
```

Eoln function

Declaration Function Eoln [(var F : text)] : Boolean;

Function Returns false if there is still data to read after the current file position, but before the next new line.

Remark F is a file variable of type text, that has been opened. If F is omitted the standard file Input is assumed.

See also SeekEoln,Eof

Example

```
Program Eoln_Demo;
Uses DOS;

Var
        F : Text;
        C : Char;
Begin
  {$I-} Reset(F,'TEMP.TXT'); {$I+}
  While NOT(Eoln(F)) DO
    Begin
      Read(F,C);
      Write(C);
    End;
  Close(F);
End.
```

Erase procedure

Declaration Procedure Erase(F : String);

Function Erases a file from disk.

Remark Do not erase an open file.

See also Rename

Example

```
Program Erase_Demo;
Var
        DummyFile : FILE;
Begin
  { Create a file. }
  ReWrite(DummyFile,'DUMMY.DTA');
  Close(DummyFile);
  Write('Press RETURN to erase the dummy file.');
  Readln;
  Erase('DUMMY.DTA');
End.
```

Exit procedure

Procedure Exit; — Declaration

Exits from the current block of code. — Function

If the current block is the main program, the call to Exit will cause the program to terminate. If the current block is a procedure or a function an exit is made, and control is passed to the caller. — Remark

Halt,ExitProc — See also

Example

```
Program Exit_Demo;

Procedure WasteTime;
Begin
  Repeat
    If (Random(1000) = 999) then EXIT;
  Until False;
End;

Begin
  WasteTime;
End.
```

ExitProc variable

Declaration Var ExitProc : Pointer;

Function Points to the start of the exit chain.

Remark The idea of the ExitProc variable is to allow user written procedures or functions to be executed just before a program terminates.
This feature can be very useful if for example some files need to be closed, before the program stops.
As can be seen from the following example, it is very important that the original value of the ExitProc variable is first saved and later restored in the exit routine. If this is not done the systems own exit procedure will not be executed, causing unpredictable results.

See also Halt,Exit

Example
```
Program ExitProc_Demo;
Var
        OldExit : Pointer;
Procedure MyExit;
Begin
 ExitProc := OldExit;{Restore the original value!}
 Writeln('End of program.');
End;
Begin
 OldExit := ExitProc;{ Save the original value! }
 ExitProc := @MyExit;{ Set the new value. }
  { Do work here. }
End.
```

Exp function

Declaration Function Exp(N) : Real;

Function Returns the exponential of N.

Remark N is an integer-type or real-type expression.

See also Ln,ValidReal

Example
```
Program Exp_Demo;
Var
        R : Real;
Begin
  Write('Enter a number: '); Readln(r);
  Writeln;
  Writeln('The enponential is: ',EXP(r));
End.
```

FExpand function

Function FExpand(Path : PathStr) : PathStr;

Declaration

Expands a file path into a fully qualified file path.

Function

Assuming that the current directory is C:\MAXPAS, the following results would be returned by FExpand:

Remark

```
FExpand('');                    = C:\MAXPAS\
FExpand('*.PAS');               = C:\MAXPAS\*.PAS
FExpand('..\*.INF');            = C:\*.INF
FExpand('..\MAXPAS\..\*.PAS');  = C:\*.PAS
```

As can be seen from the above, FExpand converts its output to uppercase.

See also

FSplit,FindFirst,Dos

Example

```
Program FExpand_Demo;

Uses DOS;

Var
       Path : PathStr;
Begin
  Write('Enter path to expand: '); Readln(Path);
  Path := FExpand(Path);
  Writeln('Fully expanded: ',Path);
End.
```

FilePos function

Declaration Function FilePos(var F) : LongInt;

Function Returns the number of the current record in file F.

Remark F is a file variable (Typed or Untyped) that has been opened. After a call to Reset or ReWrite, FilePos will return 0.

See also Seek,FileSize

Example

```
Program FilePos_Demo;

Var
        DataFile : FILE OF Byte;
        B        : Byte;
Begin
  ReWrite(DataFile,'TEMP.DTA');  { Create a temporary file. }
  For B := 1 to 100 DO
    Write(Datafile,B);

  Repeat { Seek around in the file. }
    Seek(DataFile,Random(FileSize(DataFile)));
    WriteLn('Current file position is: ',FilePos(DataFile));
  Until KeyPressed;

  Close(DataFile);
  Erase('TEMP.DTA');
End.
```

FileSize function

Declaration Function FileSize(var F) : LongInt;

Function Returns the number of records in file F.

Remark F is a file variable (Typed or Untyped) that has been opened.

See also FilePos,Seek

FillChar procedure

Procedure FillChar(var Object; Count : LongInt; FillVal : Byte);

Declaration

Fills memory starting at the address specified by Object.

Function

Object is any variable. Count specifies how many bytes of memory that are to be filled. FillVal specifies the value that is to be used for the fill.

Remark

If Count is greater than the size of Object, other data or code may be overwritten, causing unpredictable results. Therefore the SizeOf function should be used to specify the number of bytes to fill.

SizeOf,Move

See also

Example

```
Program FillChar_Demo;

Var
        Big : Packed Array[1..30000] OF Byte;
        C   : Integer;
Begin
  Writeln('Doing it the slow way.');
  For C := 1 to 30000 DO
    Big[c] := 0;

  Writeln('Doing it the fast way.');
  FillChar(Big,SizeOf(Big),0);
End.
```

FindFirst procedure

Declaration Procedure FindFirst(Path : String; Attr : Byte; var S : SearchRec);

Function Searches a directory for the first entry matching a specified file name and set of attributes.

Remark Path is a path to a directory on a specified drive. The path consists of a directory name, which is optional, and a file name. The file name can contain wild card characters (* and ?). The directory name must be implicit.
The Attr parameter specifies the special files to search for (in addition to all normal files).
The file attributes are declared in the DOS unit as follows:

```
Const
        ReadOnly        = $01;
        Hidden          = $02;
        SysFile         = $04;
        VolumeID        = $08;
        Directory       = $10;
        Archive         = $20;
        AnyFile         = $3F;
```

Upon return FindFirst places the result of the search in the S variable. S is of the type SearchRec, which is declared in the DOS unit:

```
Type
    SearchRec  = Record
                   Reserved     : packed Array[0..20] OF Byte;
                   Attr         : Byte;
                   Time         : LongInt;
                   Size         : LongInt;
                   Name         : String[12];
                 End;
```

After a call to FindFirst the variable DosError must be checked. If it is not 0, the contents of the S parameter will be garbage.

See also FindNext,UnPackTime,DosError,Dos

Example

```
Program FindFirst_FindNext_Demo;
Uses DOS;
Var
        DosData : SearchRec;
Begin
  FindFirst( '*.*' , AnyFile, DosData );
  While (DosError = 0) DO
    Begin
      With DosData DO
        Write(Name :20 );
      FindNext(DosData);
    End;
End.
```

FindNext procedure

Procedure FindNext(var S : SearchRec) — Declaration

Finds the next entry matching the specifications given in a previous call to FindFirst. — Function

The S parameter must be the same one that was used in the call to FindFirst. — Remark

After a call to FindNext the variable DosError must be checked. If it is not 0, the contents of the S parameter will be garbage.

FindFirst,Dos — See also

FreeMem procedure

Procedure FreeMem(var P : Pointer; Size : LongInt); — Declaration

Disposes a dynamic variable of a given size. — Function

P is a pointer variable of any type that was previously assigned by the GetMem procedure. Size specifies the size of the dynamic variable to dispose. Size must be equal to the size specified in the call to GetMem. — Remark

GetMem,New,Dispose — See also

Example

```
Program FreeMem_Demo;

Var
        Data : Pointer;
        F    : FILE;
        R    : Integer;
Begin
  GetMem(Data,1024); { Allocate 1K in the heap }
  Reset(F,'TEMP.DTA');
  BlockRead(F,Data^,1024,R);
  Close(F);
  FreeMem(Data,1024);
End.
```

FSplit procedure

Declaration Procedure FSplit(P : PathStr; var D:DirStr;var N:NameStr;var E:ExtStr);

Function Splits a file path into its 3 components:

The directory specification.
The file name.
The file extension.

Remark Any of the 3 components can be empty on return, if the file path did not contain that component on entry.

See also The PathStr, DirStr, NameStr and ExtStr are declared in the DOS unit.

Example

```
Type
        PathStr         = String[79];
        DirStr          = String[67];
        NameStr         = String[8];
        ExtStr          = String[4];
```

FExpand,GetDir,Dos

```
Program FSplit_Demo;

Uses DOS;

Var
        FullPath : PathStr;
        Dir      : DirStr;
        Name     : NameStr;
        Ext      : ExtStr;
Begin
  Repeat
    Write('Enter path to split: ');
Readln(FullPath);
    FSplit(FullPath,Dir,Name,Ext);
    Writeln;
    Writeln(FullPath,' consists of:');
    Writeln;
    Writeln('The directory     : ',Dir);
    Writeln('The file name     : ',Name);
    Writeln('With the extension: ',Ext);
  Until (fullPath = '');
End.
```

GetDate procedure

Procedure GetDate(var Year, Month, Day, DayOfWeek : Integer); — Declaration

Returns the current date set in the operating system. — Function

The ranges of the returned values are: — Remark

```
Year       1980..2099
  Month      1..12
  Day        1..31
  DayOfWeek  0..6 (Where 0 = Sunday, 1 = Monday ... )
```

SetDate,GetTime,Dos — See also

Example

```
Program GetDate_Demo;

Uses DOS;

Var
        Year,Month,Day,DayOfWeek        : Integer;
Begin
  GetDate(Year,Month,Day,DayOfWeek);
  Write('Today is ');
  Case DayOfWeek OF
    0 : Write('Sunday');
    1 : Write('Mondag');
    2 : Write('Tuesday');
    3 : Write('Wednesday');
    4 : Write('Thursday');
    5 : Write('Friday');
    6 : Write('Saturday');
  End;
  Writeln('.');
End.
```

GetDir procedure

Declaration Procedure GetDir(Drive : Byte; var Dir : DirStr);

Function Returns the current directory on a specified disk drive.

Remark Drive numbers are ordered as follows:

drive 0 = Current disk drive.
- 1 = Drive A:
- 2 = Drive B:
....

See also ChDir,RmDir,MkDir,GetDrive,Dos

Example

```
Program GetDir_Demo;

Uses DOS;

Var
        CurDir : DirStr;
Begin
  GetDir(0,CurDir);
  Writeln('Current dir on current drive is: ',CurDir);
  GetDir(1,CurDir);
  Writeln('Current dir on drive A: is     : ',CurDir);
End.
```

GetDrive function

Declaration Function GetDrive : Byte;

Function Returns the number of the current disk drive.

Remark The drive numbers are ordered as follows:

0 = Drive A:
1 = Drive B:
....

See also SetDrive,GetDir,Dos

Example

```
Program GetDrive_Demo;

Uses DOS;

Begin
  Writeln('Current drive is: ',GetDrive);
End.
```

GetEnv function

Function GetEnv(MatchStr : String) : String; — Declaration

Returns the value of a specific environment string. — Function

If no match is found for MatchStr, GetEnv will return an empty string. — Remark

EnvCount,EnvStr,Dos — See also

Example

```
Program GetEnv_Demo;

Uses DOS;

Begin
  Writeln('Current PATH is: ', GetEnv('PATH'));
End.
```

GetFattr procedure

Declaration Procedure GetFAttr(Fil : PathStr; var Attr : Integer);

Function Gets the attributes of a file.

Remark The file attributes and PathStr are declared in the DOS unit:

```
Const
        ReadOnly        = $01;
        Hidden          = $02;
        Sysfile         = $04;
        VolumeID        = $08;
        Directory       = $10;
        Archive         = $20;
        AnyFile         = $3F;

Type
        PathStr         = String[79];
```

See also SetFAttr,DosError,Dos

Example

```
Program GetFAttr_Demo;

Uses DOS;

Var
        Attrs    : Integer;
Begin
  If (ParamCount = 0) then HALT;
  GetFAttr( ParamStr(1), Attrs );
  If (DosError = 0) then
    Begin
      If (Attrs AND ReadOnly > 0) then
        WriteLn('ReadOnly');
      If (Attrs AND Hidden > 0) then
        WriteLn('Hidden');
      If (Attrs AND SysFile > 0) then
        WriteLn('System');
      If (Attrs AND VolumeID > 0) then
        WriteLn('Disk label');
      If (Attrs AND Directory > 0) then
        WriteLn('Directory');
      If (Attrs AND Archive > 0) then
        WriteLn('Archive');
    End
   ELSE
     Writeln('No file name specified.');
End.
```

GetFTime procedure

Procedure GetFTime(var F; var Time : LongInt); — Declaration

Gets the time and date of a file. — Function

The F parameter is a file variable (Typed, Untyped or Text) that has been either Reset, Rewritten or Appended. — Remark

The returned long integer can be unpacked, using the UnPackTime procedure.

SetFTime,UnPackTime,Reset,DosError,Dos — See also

Example

```
Program GetFTime_Demo;

Uses DOS;

Var
       DiskFile : TEXT;
       Name     : String;
       DatTim   : DateTime;
       PackData : LongInt;
Begin
  Write('Enter name of file to work with: '); Readln(Name);
  {$I-} Reset(DiskFile,Name); {$I+}
  If (IOresult = 0) then
    Begin
      GetFTime( DiskFile, PackData );
      UnpackTime(PackData,DatTim);
      With DatTim DO
        Begin
          Writeln('Date of update/creation is: ',Day,'-
',Month,'/',Year);
          Writeln('Time of update/creation is:
',Hour,'.',Min,',',Sec);
        End;
      Close(DiskFile);
      Writeln('Done.');
    End
  ELSE
    Writeln('Error while opening file.');
End.
```

GetMem procedure

Declaration Procedure GetMem(var P : Pointer; Size : LongInt);

Function Creates a new dynamic variable and sets P to point to it.

Remark P is a pointer variable of any type. Size specifies the size in bytes of the dynamic variable to allocate.

See also FreeMem,Dispose,New,MemAvail,MaxAvail

Example

```
Program GetMem_Demo;

Var
        Data : Pointer;
        F    : FILE;
        R    : Integer;
Begin
  GetMem(Data,1024); { Allocate 1K in the heap }
  Reset(F,'TEMP.DTA');
  BlockRead(F,Data^,1024,R);
  Close(F);
  FreeMem(Data,1024);
End.
```

GetTime procedure

Procedure GetTime(var Hour, Min, Sec, Sec100 : Integer); — Declaration

Returns the current time in the operating system. — Function

The ranges of the returned values are: — Remark

```
Hour        0..23
Min         0..59
Sec         0..58
```

The Sec100 parameter, which has been implemented for compatability with Turbo Pascal, is not supported by the operating system, and will therefore always return 0. Due to the operating system, the returned seconds will always be dividable with two.

SetTime,GetDate,Dos — See also

Example

```
Program GetTime_Demo;

Uses DOS;

Var
        Hour,Minute,Second,Hundreds : Integer;
Begin
  Repeat
    GetTime(Hour,Minute,Second,Hundreds);
    GoToXY(1,1); ClrEol;
    Write(Hour,'.',Minute,',',Second);
  Until keyPressed;
End.
```

GetVerify function

Declaration Procedure GetVerify(var Verify : Boolean);

Function Returns the value of the disk verify flag.

Remark When verify is on (True), TOS will check to see if data written to the disk, has been stored properly. This is done by trying to read the data just written. When off (False), TOS will simply write the data.

See also SetVerify,Dos

Example

```
Program GetVerify_Demo;
Uses DOS;
Var
        Res : Boolean;
Begin
  Write('Current state of disk verify is ');
  GetVerify( Res );
  Case Res OF
    True  : Writeln('ON');
    False : Writeln('OFF');
  End;
End.
```

GotoXY procedure

Declaration Procedure GotoXY(Col, Lin : Integer);

Function Positions the cursor at specific screen coordinates.

Remark On a standard monochrome screen the range for X and Y are:
X 1..80 Y 1..25
Where position 1,1 is the upper left corner.

See also WhereX,WhereY,ClrEos,InsLine,DelLine

Example

```
Program GoToXY_Demo;
Var
        X,Y : Integer;
Begin
  For X := 1 to 80 DO
    For Y := 1 to 25 DO
      Begin
        GoToXY(X,Y);
        If (X*Y) < (80*25) then
          Write('*');
      End;
End.
```

Halt procedure

Procedure Halt [(ExitCode : Integer)];

Declaration

Halts the current program.

Function

If the optional parameter ExitCode is not specified, an exitcode of 0 will be used.

Remark

Exit,ExitProc

See also

Example

```
Program Halt_Demo;

Uses DOS;

Begin
  If (ParamCount = 0) then
    HALT(99);
End.
```

Hi function

Function Hi(N : Integer) : Byte;

Declaration

Returns the high byte of N.

Function

Lo,LoWord,HiWord,Swap

See also

Example

```
Program Hi_Demo;
Begin
  Writeln(Hi($1234));
End.
```

HiWord function

Function HiWord(N : LongInt) : Integer;

Declaration

Returns the high word of N.

Function

LoWord,Hi,Lo,SwapWord

See also

Example

```
Program HiWord_Demo;
Begin
  Writeln(HiWord($12345678));
End.
```

Inc procedure

Declaration Procedure Inc(N [, Count]);

Function Increments an ordinal variable by 1 or by Count.

Remark N and the optional parameter Count are both simple types. If Count is not specified N will be incremented by 1. Otherwise N will be incremented by Count.

See also Dec,Pred,Succ

Example

```
Program Inc_Demo;

Var
        L : LongInt;
Begin
  L := 0;
  Repeat
    Inc(L);     { Increase L by 1 }
    Inc(L,2);   { Increase L by 2 }
  Until (L > 100000);
End.
```

Include function

Declaration Function Include(NewStr, MainStr : String; Index : Byte);

Function Returns MainStr with NewStr inserted at position Index.

Remark MainStr is not affected by the call to Include. Except for this the rules for the Include function are the same as for the Insert procedure.

See also Insert

Example

```
Program Include_Demo;
Begin
  Writeln(Include('IS ','ATARI GREAT',7));
End.
```

Insert procedure

Procedure Insert(New : String; var Main : String; Index : Byte); — Declaration

Inserts New into Main at position Index. — Function

Include,Copy,Delete,Length,Pos — See also

Example

```
Program Insert_Demo;

Var
        S : String;
Begin
  S := 'Hello';
  Insert(' World!',S,6);
  Writeln(S);
End.
```

InsLine procedure

Procedure InsLine; — Declaration

Inserts a blank line on the screen. — Function

All lines under and including the current line will be scrolled down, and the bottom line is lost. — Remark

DelLine,GotoXY — See also

Example

```
Program InsLine_Demo;

Var
        X,Y : Integer;
Begin
  For X := 1 to 80 DO
    For Y := 1 to 24 DO
      Begin
        GoToXY(X,Y); Write(Random(2));
      End;
  GoToXY(30,10);
  Writeln('Press ENTER to go on');
  Readln;
  GoToXY(1,1);
  For Y := 1 to 25 DO
    InsLine;
End.
```

Int function

Declaration Function Int(N : Real) : Real;

Function Returns the integer part of N.

See also Trunc,Round,Abs

Example

```
Program Int_Demo;

Var
        R : Real;
Begin
  R := INT(1234.5678); { R = 1234.00 }
End.
```

IOresult function

Declaration Function IOresult : Integer;

Function Returns the error status of the last IO operation.

Remark It is important to notice that IOresult is used as an error status for several IO routines in the system. Therefore it is not reliable to make a program like this:

```
{ Do some file operations here. }
Writeln('Error code: ',IOresult);
```

Since Writeln also uses IOresult to repport errors. (Thereby setting IOresult to 0 when writing 'Error code: '). Instead do like this:

```
{ Do some file oprations here. }
MyVar := IOresult;
Writeln('Error code: ',MyVar);
```

Example

```
Program IOresult_Demo;

Var
        F : FILE;
Begin
  {$I-} Reset(F,'TEMP.TXT'); {$I+}
  If (IOresult <> 0) then
    ReWrite(F,'TEMP.TXT');
End.
```

KeyPressed function

Function KeyPressed : Boolean; — Declaration

Checks to see if the keyboard has been pressed. — Function

KeyPressed will return True if a key is waiting to be read, otherwise False. — Remark

ReadKey — See also

Example

```
Program KeyPressed_Demo;

Begin
  Repeat
    Writeln('Please press a key.');
  Until KeyPressed;
  Writeln('You pressed ',ReadKey);
End.
```

Length function

Function Length(S : String) : Byte; — Declaration

Returns the length of string S. — Function

The returned value will be in the range of 0..255. — Remark

Copy,Concat,Insert,Include,Delete,Omit,Pos — See also

Example

```
Program Length_Demo;

Var
       S : String;
Begin
  Write('Enter a string: '); Readln(S);
  Writeln;
  Writeln('You entered ',Length(S),' characters');
  Writeln('You entered ',Ord(S[0]),' characters');
End.
```

Ln function

Declaration Function Ln(R) : Real;

Function Returns the natural logarithm of R.

Remark N is a real-type expression. The result is the natural logarithm of R.

See also Exp,ValidReal

Example

```
Program Ln_Demo;

Var
        R : Real;
Begin
  R := LN(100);
End.
```

Lo function

Declaration Function Lo(N : Integer) : Byte;

Function Returns the low byte of N.

See also Hi,HiWord,LoWord

Example

```
Program Lo_Demo;

Var
        X : Byte;
Begin
  X := Lo($1234); { = $34 }
End.
```

LoWord function

Declaration Function LoWord(N : LongInt) : Integer;

Function Returns the low word of N.

See also HiWord,Hi,Lo

Example

```
Program LoWord_Demo;
Begin
  Writeln(LoWord($12345678));
End.
```

MaxAvail function

Function MaxAvail : LongInt; — Declaration

Returns the size of the largest contiguous memory block in the heap. — Function

MemAvail,New,GetMem — See also

Example

```
Program MaxAvail_Demo;

Var
        P : Pointer;
Begin
  If (MaxAvail > 1000) then
    GetMem(P,1000);
End.
```

MemAvail function

Function MemAvail : LongInt; — Declaration

Returns the sum of the sizes of all free memory blocks in the heap. — Function

MaxAvail — See also

Example

```
Program MemAvail_Demo;
Begin
  Writeln('Bytes free in the heap is: ',MemAvail);
End.
```

MkDir procedure

Procedure MkDir(Dir : DirStr); — Declaration

Creates a new directory. — Function

ChDir,RmDir,GetDir,DosError,Dos — See also

Example

```
Program MkDir_Demo;

Uses DOS;

Begin
  MkDir('MYDIR');
End.
```

Move procedure

Declaration Procedure Move(var Source, Dest; Count : LongInt);

Function Move Count bytes from Source to Dest.

Remark Source and Dest can be any variable. Count specifies how many bytes to move.

If Count is greater then the size of Dest, other code or data may be overwritten, causing unpredictable problems. Therefore it is a good idea to use the SizeOf function to specify how many bytes to move:

Example

```
Var
       Big   : LongInt;
       Small : Integer;
Begin
  Move(Big,Small,4);     { The very dangerous way of doing it! }
  Move(Big,Small,SizeOf(Small));  { The safe way of doing it. }
End.
```

Sizeof,FillChar

```
Program Move_Demo;

Var
        Big1  : Packed Array[1..2000] OF Integer;
        Big2  : Packed Array[1..1000] OF Integer;
Begin
  Move(Big1,Big2,SizeOf(Big2));
End.
```

New procedure

Procedure New(var P : Pointer);

Declaration

Creates a new dynamic variable and sets P to point to it.

Function

P is a pointer variable of any type. The size of the allocated memory block corresponds to the size of the type that P points to.

Remark

Dispose,GetMem,FreeMem,MemAvail,MaxAvail

See also

Example

```
Program New_Demo;

Type
        PersonType = Record
                       Name    : String[20];
                       Age     : Byte;
                     End;
Var
        PersonData : ^PersonType;
Begin
  New(PersonData); { Allocate space in heap }
  With PersonData^ DO
    Begin
      Write('Enter name: '); ReadLn(Name);
      Write('Enter age : '); ReadLn(Age);
    End;
  Dispose(PersonData); { Dispose the space }
End.
```

Odd function

Declaration Function Odd(N) : Boolean;

Function Tests to see if the N parameter is an odd number.

Remark N is an integer type value.

Example

```
Program Odd_Demo;

Var
        L : LongInt;
Begin
  Repeat
    Write('Enter a number: '); Readln(L);
    If Odd(L) then
      Writeln('ODD')
    ELSE
      Writeln('EVEN');
  Until (L = 0);
End.
```

Omit function

Declaration Function Omit(S : String; Index, Count : Integer) : String;

Function Returns S with Count characters deleted, starting at position Index.

Remark The rules for the Omit function are the same as for the Delete procedure, except for the fact that the S string is left untouched when using the Omit function.

See also Delete

Example

```
Program Omit_Demo;
Begin
  Writeln(Omit('ATARI IS NOT GREAT',10,4));
End.
```

Ord function

Function Ord(x) : Integer; — Declaration

Returns the ordinal number of an ordinal-type or pointer-type value. — Function

If X is an ordinal-type expression, the result is of type integer and the value is the ordinality of X. If X is a pointer-type expression, the result is of type LongInt, and the value is the address of the dynamic variable pointed to by X. — Remark

Ord4,Chr — See also

Example

```
Program Ord_Demo;

Var
        C : Char;
Begin
  Write('Enter something: ');
  C := ReadKey;
  Writeln;
  Writeln('The ordinal value of ',C,' is ',ORD(C));
End.
```

Ord4 function

Function Ord4(x) : LongInt; — Declaration

Returns the ordinal number of an ordinal-type value. — Function

This function does the same as the Ord function, but the result is always a long integer. — Remark

Ord,Chr — See also

PackTime procedure

Declaration Procedure PackTime(D : DateTime; var Result : LongInt);

Function Packs a 12-byte DateTime record into a 4-byte long integer.

Remark The DateTime record is declared in the DOS unit:

```
Type
     DateTime          = Record
                           Year        : Integer;
                           Month       : Integer;
                           Day         : Integer;
                           Hour        : Integer;
                           Min         : Integer;
                           Sec         : Integer;
                         End;
```

PackTime returns a long integer with the following format:

```
bits     0 -  4  seconds (/ 2)    16 - 20 day of m.
 -       5 - 10  minutes           21 - 24 month
 -      11 - 15  hours             25 - 31 year
```

The returned long integer can be used by SetFTime and UnPackTime.

See also UnPackTime,SetFTime,Dos

Example

```
Program PackTime_Demo;
Uses DOS;
Var
       U : DateTime;    { The unpacked time and date }
       P : LongInt;     { The packed time and date }
Procedure WriteData;
Begin
  With U DO
    Begin
      WriteLn('Time: ',Hour,'.',Min,',',Sec);
      WriteLn('Date: ',Day,'-',Month,'/',Year);
    End;
  Writeln;
End;
Begin { main }
  With U DO
    Begin
      Year := 1990; Month := 12; Day := 24;
      Hour := 12; Min := 0; Sec := 0;
    End;
  Writeln('Before packing:'); WriteData;
  PackTime(U,P);
  Writeln('The packed time and date: ',P);
  UnPackTime(P,U);
  Writeln('After unpacking:'); WriteData;
End.
```

Page procedure

Procedure Page [(var F : text)]; — Declaration

Writes a formfeed character to file F. — Function

If F is omitted, the standard file Output is assumed. Page is equivalent to: Write(F, Chr(12)); — Remark

Example

```
Program Page_Demo;
Var
        T : Text;
Begin
  ReWrite(T,'PRINTER.DTA');
  Writeln(T,'First page.');
  Page(T);
  Writeln(T,'Second page.');
  Page(T);
  Writeln(T,'Third page.');
  Page(T);
  Close(T);
End.
```

ParamCount function

Function ParamCount : Integer; — Declaration

Returns the number of parameters that were passed on the command line. — Function

Use the ParamStr function to obtain the value of the parameters. — Remark

ParamStr,Dos — See also

Example

```
Program ParamCount_Demo;
Uses DOS;

Var
       Res, Count : Integer;
Begin
  Res := ParamCount;
  Writeln('You passed ',Res,' parameters to this program.');
  Writeln;
  If (Res = 0) then
    HALT;
  Writeln('Here is a list of the parameters:');
  Writeln;
  For Count := 1 to Res DO
    Writeln('Parameter #',Count:2,' :=',ParamStr(Count));
End.
```

ParamStr function

Declaration Function ParamStr(Index : Byte) : String;

Function Returns command line parameter number Index.

Remark If Index is 0 or greater than ParamCount, ParamStr will return an empty string.

See also ParamCount,Dos

Example

```
Program ParamStr_Demo; { Displays 1 or more text files. }

Uses DOS;

Var
        DataFile : TEXT;
        Data     : String;
        Count    : Integer;

Procedure DisplayFile( Number : Integer );
Begin
  {$I-} Reset( DataFile, ParamStr(Number) ); {$I+}
  If (IOresult <> 0) then
    Begin
      Writeln('Error while opening file: ',ParamStr(Number));
      EXIT;
    End;
  While NOT(Eof(DataFile)) DO
    Begin
      Readln(DataFile, Data);
      Writeln(Data);
    End;
  Close(DataFile);
End; { DisplayFile }

Begin
  For Count := 1 to ParamCount DO
    DisplayFile( Count );
End.
```

Pi constant

Const Pi = 3.14159265358979324; — Declaration

Returns the value of Pi. — Function

Example

```
Program Pi_Demo;

Begin
  Writeln('Pi is: ',Pi :23:20);
End.
```

Pos function

Function Pos(SubStr, MainStr : String) : Byte; — Declaration

Searches MainStr for the first occurrence of SubStr. — Function

If SubStr does not exist in MainStr, the Pos function will return 0, else Pos will return the position in MainStr where SubStr was found. — Remark

Here are some examples: — Example

```
Pos('HELLO','HELLO WORLD') = 1
Pos('CAL','PASCAL')        = 4
Pos('TS','ATARI ST')       = 0
```

Copy,Insert,Delete,Length

```
Program Pos_Demo;

Var
        MainStr,
        SubStr    : String;
        P         : Byte;
Begin
  Write('Enter main string: '); Readln(MainStr);
  Write('Find what        : '); Readln(SubStr);
  Writeln;
  P := Pos(SubStr,MainStr);
  If (P = 0) then
    Writeln('Not found.')
  ELSE
    Writeln('Found at position ',P);
End.
```

Pred function

Declaration Function Pred(X) : (Same type as parameter)

Function Returns the predecessor of the parameter.

Remark X is an ordinal-type value.

See also Succ,Dec,Inc

Example

```
Program Pred_Demo;

Begin
  Writeln(Pred(2)); { = 1 }
End.
```

Ptr function

Declaration Function Ptr(Address : LongInt) : Pointer;

Function Converts a long integer value to a pointer type value.

Remark The result of Ptr is a pointer that points to the memory location specified by the value of Address.

See also SPtr,Addr

Example

```
Program Ptr_Demo;

Uses DOS;

Var
        SP : Pointer;
        P  : ^integer;
        I  : Integer;
Begin
  Sp := Super(NIL);
  P  := Ptr($4a6);
  I  := P^;
  Sp := Super(Sp);
  Writeln(I,' disk drives are connected to your
system.');
End.
```

Random function

Function Random [(Max : LongInt) : LongInt] | [: Real]; — Declaration

Returns a random number. — Function

If the optional Max parameter is specified, the result of Random will be a long integer in the range of 0..Max-1. — Remark

If no parameter is given to the Random function a random real number, in the interval 0 <= x < 1, will be returned.

Randomize,RandSeed — See also

Example

```
Program Random_Demo;

Var
        Tries ,
        Secret,
        Guess    : Integer;
Begin
  Randomize;
  Writeln('Guess a number.');  Writeln;
  Secret := Random(100);
  Tries  := 0;
  Repeat
    Inc(Tries);
    Write('Enter your guess number ',Tries,': ');
    Readln(Guess);
    If (Guess < Secret) then
      Writeln('Too low.')
    ELSE
    If (Guess > Secret) then
      Writeln('Too high.');
  Until (Guess = Secret);
  Writeln('You guessed it in ',Tries,' attempts.');
End.
```

Randomize procedure

Declaration Procedure Randomize;

Function Initializes the random-number generator with a random value.

Remark Every time a program is started, the random-number generator is initialized with the same value (0). The effect of this is that random numbers generated by a program will be the same every time the program is run.

There are 2 ways of getting around this problem:

1. Call Randomize to re-initialize the random-number generator.
2. Change the value of the RandSeed variable.

See also Random,RandSeed

Example

```
Program Randomize_Demo;

Begin
 { This will be the same every time the program is run. }
  Writeln(Random(1000));

  Randomize;
  { This will not! }
  Writeln(Random(1000));
End.
```

RandSeed variable

Declaration Var RandSeed : LongInt;

Function RandSeed is used as the seed for the random-number generator.

Remark Setting RandSeed to a specific value can make the Random function return the same random numbers over and over again. This feature is useful in Encryption and Simulation algorithms.

See also Random,Randomize

Example

```
Program RandSeed_Demo;

Begin
  RandSeed := 1234;
  Writeln(Random(9999));{ This }
  RandSeed := 1234;
  Writeln(Random(9999));{ and this will always be the same. }
End.
```

Read procedure

Procedure Read [([var F;] v1 [,v2,..vn])];

Declaration

Reads 1 or more components from file F into 1 or more variables.

Function

Read for text files:

Remark

F is a file variable of type text. If omitted the standard file Input is assumed. Each of the variables (v1,v2..vn) must be of type char, integer, real or string.

Read for typed files:

F is a file variable of any type except text. Each of the variables (v1,v2,vn) must be of the same type as the component type of file F.

ReadLn,Write,WriteLn

See also

Example

```
Program Read_Demo;

Uses DOS;

Var
        T : Text;
        C : Char;
Begin
  {$I-} Reset(T,'TEST.TXT'); {$I+}
  If (IOresult = 0) then
    Begin
      While NOT(Eof(T)) DO
        Begin
          Read(T,C);
          Write(C);
        End;
      Close(T);
    End
  ELSE
    Writeln('File not found.');
End.
```

ReadKey function

Declaration Function ReadKey : Char;

Function Reads a character from the keyboard.

Remark If no character is waiting to be read a key must be pressed in order to exit the ReadKey function. Otherwise a character is read from the systems internal keyboard buffer.

See also KeyPressed,UpCase

Example

```
Program ReadKey_Demo;

Begin
  Repeat
    Writeln('Press SPACEBAR to stop this.');
  Until (ReadKey = #32);
End.
```

ReadLn procedure

Declaration Procedure ReadLn [([var F : text;] v1 [,v2,..vn])];

Function Executes the Read procedure, and then skips to the next line in the file.

Remark If no next line can be found in the file, the Eof(F) will return true. If ReadLn is called with only a file variable as parameter, the file pointer will advance to the beginning of the next line in the file.

See also Read,WriteLn,Write

Example

```
Program ReadLn_Demo;

Var
        S : String;
Begin
  Write('What is your name: '); ReadLn(s);
End.
```

ReWrite procedure

Procedure ReWrite(var F [; FileName : String][; BufSize : LongInt]); — Declaration

Creates or resets a file. — Function

F is a file variable of any type. FileName specifies the name of the file on the disk. The optional BufSize can be specified with files of type text to determine the size of a buffer. — Remark

The ReWrite procedure can be used in 3 different ways:

ReWrite(MyFile, 'NAME.EXT');

Creates and/or opens the file MyFile. If the file is of type text it is opened as Write Only.

ReWrite(MyFile, 'NAME.EXT', 10000);

Creates and/or opens the text file MyFile. A 10000 byte buffer is created for the file.

ReWrite(MyFile);

Positions the file pointer at the start of MyFile. MyFile must be open.

Reset,Append — See also

Example

```
Program ReWrite_Demo;

Var
        Data,
        Name : String;
        Out  : Text;
Begin
  Write('Create what file: '); Readln(Name);
  {$I-} ReWrite(Out,Name); {$I+}
  If (IOresult = 0) then
    Begin
      Repeat
        Write('Enter data: '); Readln(Data);
        Writeln(Out,Data);
      Until (Data = '');
      Close(Out);
    End
  ELSE
    Writeln('Could not create ',Name);
End.
```

RmDir procedure

Declaration Procedure RmDir(Dir : DirStr);

Function Removes an existing directory.

See also ChDir,MkDir,GetDir,Dos

Example
```
Program RmDir_Demo;
Uses DOS;
Begin
  RmDir('MYDIR');
End.
```

Round function

Declaration Function Round(N : Real) : LongInt;

Function Rounds a real-type value to an integer-type value.

See also Int,Trunc,Abs

Example
```
Program Round_Demo;
Var
        L : LongInt;
Begin
  L := ROUND(1.40); { L = 1 }
  L := ROUND(1.50); { L = 2 }
End.
```

RunFromMemory function

Declaration Function RunFromMemory : Boolean;

Function Returns True if the program is running from within the editor.

Remark If you are running the compiler/editor from within a debugger, the RunFromMemory function will probarly return an incorrect result.

See also System2

Example
```
Program RunFromMemory_Demo;
Begin
  Write('You are');
  Case RunFromMemory OF
    False : Write(' NOT');
  End;
  Writeln(' running from within the HIDE.');
End.
```

Seek procedure

Procedure Seek(var F; Rec : LongInt); — Declaration

Moves the file pointer to a specific record in file F. — Function

F is a file variable (Typed or Untyped) that has been opened. Rec is the number of the record to seek to. The number of the first record in a file is 0. — Remark

FilePos,FileSize — See also

SeekEof function

Function SeekEof [(var F : text)] : Boolean; — Declaration

As Eof, but skips over all white spaces. — Function

F is a file variable of type text, that has been opened. If F is omitted the standard file Input is assumed. — Remark

Eof — See also

Example

```
Program SeekEof_Demo;

Var
       T : Text;
       X : Integer;
Begin
  ReWrite(T,'SPACES.DTA'); { Make a file with lots
of spaces. }
  For X := 1 to 1000 DO
    Write(T,#32);
  Reset(T);                { Go to start of file. }

  If (SeekEof(T)) then
    Writeln('File contains nothing or only
spaces.')
  ELSE
    Writeln('The file contains data.');

  Close(T); Erase('SPACES.DTA');
End.
```

SeekEoln function

Declaration Function SeekEoln [(var F : text)] : Boolean;

Function As Eoln, but skips over all white spaces.

Remark F is a file variable of type text, that has been opened. If F is omitted the standard file Input is assumed.

See also Eoln

Example

```
Program SeekEoln_Demo;
Uses DOS;
Var
        F : Text;
Begin
  Reset(F,'TEMP.TXT');
  If (SeekEoln(F)) then
    Writeln('First line in file is empty.')
  ELSE
    Writeln('There is data on the first line.');
  Close(F);
End.
```

SetDate procedure

Declaration Procedure SetDate(Year, Month, Day : Integer);

Function Sets the current date in the operating system.

Remark The legal ranges are: Year 1980..2099, Month 1..12, Day 1..31

See also GetDate,SetTime,Dos

Example

```
Program SetDate_Demo;
Uses DOS;
Var
        Year,Month,Day,Dow : Integer;
Begin
  Write('Enter Year   : '); Readln(Year);
  Write('Enter Month  : '); Readln(Month);
  Write('Enter Day    : '); Readln(Day);
  SetDate(Year,Month,Day);
  GetDate(Year,Month,Day,Dow);
  Writeln('Date has been set to:');
  Writeln('Year  = ',Year);
  Writeln('Month = ',Month);
  Writeln('Day   = ',Day);
End.
```

SetDrive procedure

Procedure SetDrive(Drive : Byte); — Declaration

Sets the active disk drive. — Function

The drive numbers are : 0 = Drive A:, 1 = Drive B:, etc. — Remark

GetDrive,ChDir,Dos — See also

Example

```
Program SetDrive_Demo;
Uses DOS;
Var
        Go : Integer;
Begin
  Write('Enter drive number: '); Readln(Go);
  SetDrive(Go);
End.
```

SetFAttr procedure

Procedure SetFAttr(Fil : PathStr; Attr : integer); — Declaration

Sets the attributes of a file. — Function

The file attributes and PathStr are declared in the DOS unit: — Remark

```
Const
        ReadOnly        = $01;
        Hidden          = $02;
        Sysfile         = $04;
        VolumeID        = $08;
        Directory       = $10;
        Archive         = $20;
        AnyFile         = $3F;

Type
    PathStr     = String[79];
```

GetFAttr,Dos — See also

Example

```
Program SetFAttr_Demo;
Uses DOS;
Var
        Name : PathStr;
Begin
  Write('Hide what file?: ');
  ReadLn(Name);
  SetFAttr( Name , Hidden );
End.
```

SetFTime procedure

Declaration Procedure SetFTime(var F; Time : LongInt);

Function Sets the time and date of a file.

Remark The F parameter is a file variable (Typed, Untyped or Text) that has been either Reset, Rewritten or Appended

The Time parameter is a long integer, that has previously been packed using the PackTime procedure.

See also GetFTime,PackTime,Reset,ReWrite,Append,DosError,Dos

Example

```
Program SetFTime_Demo;

Uses DOS;

Var
        DiskFile : TEXT;
        DatTim   : DateTime;
        PackData : LongInt;
Begin
  {$I-} Reset(DiskFile,'TEST.TXT'); {$I+}
  If (IOresult = 0) then
    Begin
      With DatTim DO
        Begin
          Write('Enter Year   : '); Readln(Year);
          Write('Enter Month  : '); Readln(Month);
          Write('Enter Day    : '); Readln(Day);
          Write('Enter Hour   : '); Readln(Hour);
          Write('Enter Minute : '); Readln(Min);
          Write('Enter Second : '); Readln(Sec);
        End;
      PackTime(DatTim, PackData);
      SetFTime(DiskFile, PackData);
      Close(DiskFile);
      Writeln('Done.');
    End
  ELSE
    Writeln('Error while opening file.');
End.
```

SetTime procedure

Procedure SetTime(Hour, Min, Sec, Sec100 : Integer);

Declaration

Sets the current time in the operating system.

Function

The legal ranges of Hour, Min, Sec are:

Remark

```
Hour        0..23
Min         0..59
Sec         0..59
```

The Sec100 parameter, which has been implemented for compatability with Turbo Pascal, is not supported by the operating system, and it will therefore be ignored by the SetTime procedure.

GetTime,SetDate,Dos

See also

Example

```
Program SetTime_Demo;

Uses DOS;

Var
        Hour,Minute,Second,Hundred : Integer;
Begin
  Write('Enter Hours   : '); Readln(Hour);
  Write('Enter Minutes : '); Readln(Minute);
  Write('Enter Seconds : '); Readln(Second);
  SetTime(Hour,Minute,Second,Hundred);
  GetTime(Hour,Minute,Second,Hundred);
  Writeln('Time has been set to:');
  Writeln('Hours   = ',Hour);
  Writeln('Minutes = ',Minute);
  Writeln('Seconds = ',Second);
End.
```

SetVerify procedure

Declaration Procedure SetVerify(Verify : Boolean);

Function Sets the value of the disk verify flag.

Remark When verify is on (True), TOS will check to see if data written to the disk, has been stored properly. This is done by trying to read the data just written.

When off (False), TOS will simply write the data.

Setting the verify flag to False will increase the speed at which data is written to the disk.

See also GetVerify,Dos

Example

```
Program SetVerify_Demo;

Uses DOS;

Var
       State    : Boolean;
Begin
  GetVerify( State );           { Read current status }
  State := NOT(State);          { Reverse the status  }
  SetVerify( State );           { Set the new status  }
End.
```

Sin function

Declaration Function Sin(R) : Real;

Function Returns the sine of the parameter.

Remark R is a real-type expression. R is assumed to represent an angle in radians.

See also Cos,Trunc,ValidReal

Example

```
Program Sin_Demo;

Var
        R : Real;
Begin
  R := SIN(20);
End.
```

SizeOf function

Function SizeOf(var Object) : Integer; — Declaration

Returns the size of an object. — Function

Object can be any variable or type. If the object has been word-aligned, the filler byte(s) are also included in the result that SizeOf returns. — Remark

FillChar,Move — See also

Example

```
Program SizeOf_Demo;

Var
       B : Byte;
       I : Integer;
       L : LongInt;
       R : Real;
       S : String;
Begin
  Writeln('The size of B is: ',SizeOf(b)); { notice the
  alignment byte. }
  Writeln('The size of I is: ',SizeOf(i));
  Writeln('The size of L is: ',SizeOf(l));
  Writeln('The size of R is: ',SizeOf(r));
  Writeln('The size of S is: ',SizeOf(s));
End.
```

SPtr function

Function SPtr : LongInt; — Declaration

Returns the current value of the stack pointer. — Function

Ptr — See also

Example

```
Program SPtr_Demo;

Begin
  Writeln('Value of stack pointer: ',SPtr);
End.
```

Sqr function

Declaration Function Sqr(N) : (Same type as parameter)

Function Returns the square of N.

Remark N is an integer-type or real-type expression. The result of Sqr(X) is equal to X*X.

See also Sqrt,ValidReal

Example

```
Program Sqr_Demo;

Var
        R : Real;
Begin
  R := SQR(10);
End.
```

Sqrt function

Declaration Function Sqrt(N) : Real;

Function Returns the square root of N.

Remark N is an integer-type or real-type expression.

See also Sqr,ValidReal

Example

```
Program Sqrt_Demo;

Var
        R : Real;
Begin
  R := SQRT(4);
End.
```

Str procedure

Procedure Str(x [:width [:decimals]]; var Res : String); — Declaration

Converts a numeric value to a string. — Function

By specifying Width and Decimals it is possible to make Res look like the output generated by Write and WriteLn. — Remark

Val,Copy — See also

Example

```
Program Str_Demo;

Var
        R : Real;
        S : String;
Begin
  R := 1234.5678;
  Writeln(R :10:4);
  Str(R :10:4, S);
  Writeln(S);
End.
```

Succ function

Function Succ(X) : (Same type as parameter) — Declaration

Returns the successor of the parameter. — Function

X is an ordinal-type value. — Remark

Pred,Inc,Dec — See also

Example

```
Program Succ_Demo;

Begin
  Writeln(Succ(1)); { = 2 }
End.
```

Super function

Declaration Function(P : Pointer) : Pointer;

Function Swithes between User mode and Supervisor mode.

Remark If P is NIL: The system switches to Supervisor mode, returning the value of the User stack pointer.
If not:The system switches back to User mode, setting the User stack pointer to P's value. The returned value is the value of the Supervisor stack pointer.

See also Dos

Example

```
Program Super_Demo;

Uses DOS;
Var
        SP : Pointer;   { Used to save stack pointer. }
        P  : ^Integer;
        B  : Byte;
Begin
  Sp := Super(NIL);      { Enter Supervisor mode. }
  P  := Ptr($440);
  B  := P^;
  Sp := Super(Sp);       { Exit Supervisor mode. }
  Write('Floppy seek rate is: ');
  Case B AND 3 OF
    0  : Write(6);
    1  : Write(12);
    2  : Write(2);
    3  : Write(3,' (default)');
  End;
  Writeln(' ms.');
End.
```

Swap function

Declaration function Swap(N : Integer) : Integer;

Function Swaps the high and low bytes in N.

See also SwapWord,Hi,Lo

Example

```
Program Swap_Demo;
Var
        X : Integer;
Begin
  X := Swap($1234); { = $3412 }
End.
```

SwapWord function

Function SwapWord(N : LongInt) : LongInt; — Declaration

Swaps the high and low words in N. — Function

Swap,HiWord,LoWord — See also

Example

```
Program SwapWord_Demo;

Var
        X : LongInt;
Begin
  X := SwapWord($12345678); { = $56781234 }
End.
```

TosVersion function

Function TosVersion : Integer; — Declaration

Returns the version number of the operating system. — Function

The primary version number is in the low part, and the secondary version number is in the high part of the returned value. — Remark

Hi,Lo,Dos — See also

Example

```
Program TosVersion_Demo;

Uses DOS;

Var
       Res : Integer;
Begin
  Res := TosVersion;
  Writeln('TOS version number is: ',LO(Res),'.',HI(Res));
End.
```

Trunc function

Declaration Function Trunc(R : Real) : LongInt;

Function Truncates a real-type value to an integer-type value.

Remark Trunc returns the value of R rounded towards 0.

See also Int,Round,Abs

Example

```
Program Trunc_Demo;

Var
        L : LongInt;
Begin
  L := TRUNC(1234.5678);
End.
```

UnPackTime procedure

Declaration Procedure UnPackTime(Time : LongInt; var Result : DateTime);

Function Unpacks a 4-byte long integer to a 12-byte DateTime record.

Remark The format of the packed long integer is:

```
bits     0 -  4  seconds (divided by 2)
-        5 - 10  minutes
-       11 - 15  hours
-       16 - 20  day of month
-       21 - 24  month
-       25 - 31  years   (since 1980)
```

The format of the returned DateTime record is:

```
Type
        DateTime          = Record
                              Year          : Integer;
                              Month         : Integer;
                              Day           : Integer;
                              Hour          : Integer;
                              Min           : Integer;
                              Sec           : Integer;
                            End;
```

PackTime

UpCase function

Function UpCase(Ch : Char) : Char; — Declaration

Converts the letter Ch to upper case. — Function

UpCase will only return a result different from Ch, if Ch is in the range of 'a'..'z'. — Remark

ReadKey — See also

Example

```
Program UpCase_Demo;

Var
        C : Char;
Begin
  Repeat
    Write('Enter a letter. ');
    C := ReadKey;
    Writeln('You pressed ',UpCase(C));
  Until (C = #13);
End.
```

Val procedure

Procedure Val(S : String; var R; var ErrorPos : Integer); — Declaration

Converts a string to a numeric variable. — Function

R is an integer-type or real-type variable. Upon return ErrorPos holds the position of the first character in S that could not be converted. — Remark

Str,Copy — See also

Example

```
Program Val_Demo;

Var
        R     : Real;
        S     : String;
        Error : Integer;
Begin
  Write('Enter a number: '); Readln(S);
  Val(S,R,Error);
  If (Error = 0) then
    Writeln('OK')
  ELSE
    Writeln('Error in number at position ',Error,' !');
End.
```

ValidReal function

Declaration Function ValidReal(R) : Boolean;

Function Returns true if R is a valid real value.

Remark Use this function to test the results of calculations done with real-type values.

See also

Example

```
Program ValidReal_Demo;

Var
        R : Real;
Begin
  R := Ln(0); { Cannot be done. }
  If (ValidReal(R)) then
    Writeln('OK')
  ELSE
    Writeln('Error in calculation.');
End.
```

WhereX function

Declaration Function WhereX : Integer;

Function Returns the column in which the cursor is currently located.

Remark On a standard monochrome monitor, the range of the returned result will be 1..80.

See also WhereY,GotoXY

WhereY function

Declaration Function WhereY : Integer;

Function Returns the line on which the cursor is currently located.

Remark On a standard monochrome monitor, the range of the returned result will be 1..25.

See also WhereX,GotoXY

Write procedure

Procedure Write [([var F;] v1 [:Width :Deci] [,v2,..vn])]; — Declaration

Writes 1 or more components to file F. — Function

Write for text files:F is a file variable of type text. If omitted the standard file Output is assumed. Each of the components to write must be of type char, integer, real, string or boolean.
Each of the components can be formatted using the two format parameters Width and Deci. Width specifies the minimum length of the component. Deci specifies the number of decimals a real component are to have.
Write for typed files:F is a file variable of any type except text. Each of the components (v1,v2,vn) must be of the same type as the comp. type of file F. — Remark

WriteLn,Read,ReadLn — See also

Example

```
Program Write_Demo;

Var
        R : Real;
        X : Byte;
Begin
  R := 1234.5678;
  For X := 20 to 40 DO
    Begin
      Write(R :X:5); Writeln;
    End;
End.
```

WriteLn procedure

Procedure WriteLn [([var F : text;] v1 [:Width :Deci] [,v2,..vn])]; — Declaration

Executes the Write procedure, and then writes an end-of-line marker to the file. — Function

Writeln can only be used on file variables of type text. — Remark

Write,ReadLn,Read — See also

Example

```
Program WriteLn_Demo;

Var
      · R : Real;
        X : Byte;
Begin
  R := 1234.5678;
  For X := 10 to 30 DO
    WriteLn(R :X:X);
End.
```

Command Line

Command Line Compiler

The HighSpeed Pascal compiler can also be used in a command line system, by using the HSPC command line version. It takes all commands from the command line and from a configuration file "HSPC.CFG". You call it as:

HSPC {Option} FileName {Option}

The call "HSPC -$R- Clock" will make HSPC compile the file Clock.Pas with the range check switch off.

The call "HSPC" alone will display a help screen.

Each option starts with a "/" or a "-", both are accepted.

The options are as follows:

```
-$R+    Range checking
-$S+    Stack checking
-$Mxxx  Memory allocation parameters
-$F+    Force 32 bit calls
-$D+    Debug information
-$I-    No I/O checking (default on)
-$V-    No var-string checking (default on)

-Fxxx   Find runtime error
-B      Build all units
-M      Make modified units
-Q      Quiet compilation

-Dxxx   Define conditionals
-Exxx   PRG & UNI directory
-Ixxx   Include directories
-Oxxx   Object directories
-Uxxx   Unit directories
```

$Switch Option

The -$ or /$ option allows you to enable or disable compiler directives, just as if they were entered into the first line of your program.

More than one directive can be entered at the same time by separating each directive with a comma. Do not use any spaces. The -$M directive will need 4 numbers after the M. These 4 numbers must be separated by commas. The numbers represents the number of KiloBytes wanted for stack, heap min/max and free to TOS (DOS).

Example:

```
HSPC -$R-,S+,M5,10,10,20 DEMO
```

Look in the Appendix "Compiler Directives" for a description of all compiler directives.

Directory Options

Three of the directory lists entered can take more than one path. They are then separated by a semi-colon.

The **-E** option defines where the generated units and programs goes. This allows you to keep all units together someplace far away or on a memory disk. Only one path can be entered.

The **-I** option defines a set of search paths used to search for include files named in $I directives in the source.

The **-O** option defines a set of search paths used to search for object files named in $L directives in the source.

The **-U** option defines a set of search paths used to search for object files units named in the USES clause in the source. If the .UNI file is also found here, it is also written here again, not regarding the -E option.

Mode Options

The **-M** option forces the compiler to check all file dates to see if any units, include files or object files used has been changed. Each unit that that have been changed will be recompiled.

The **-B** option forces the compiler to re-compile all units used, if the source can be found.

If -M or -B is not used, the compiler does not check anything, it just compiles the file requested.

The **-Fxxx** option will start the compiler in find-mode.

The -Q option will hide any output to the screen exept for error messages.

The **-Dxxx** option defines conditional compiler variables. They are used for source control. With it you can make part of your code be compiled dependent on which flags (names) you activate.

Configuration File

The file HSPC.CFG contains initial compiler definitions, that you do not want to enter each time you start HSPC. It is simply a number of options entered line by line. These options are read before the command line is parsed, allowing you to define your own default settings.

Menu

Desk:
Information about copyright and compiler version.

File:
Loading existing files ..
Creating new files ..
Saving the file ..
Printing a file ..
- in the editor
Quitting High-Speed
Execute a program outside High-Speed

Edit:
Cut/Copy/Paste text
Indent and outdent text

Search:
Find and replace text.
Toggle between windows

Compile:
Compile Make or Build a program.
Set the destination of the object code (disk or memory).
Find a run-time error.
Set the primary file.
Get information about the current source file.

Options:
Contains general setting that control how the integrated environment works:
General settings.
Compiler settings.
Linker settings.
Run settings.
Save all options to disk.
Also contains the help system.

FILE

New (^N)
Opens a new editor window with an empty file. This file has noname, you name it when saving it to disk.

Open (^O)
Opens a window with the file you select from the file selector box.

Open Selection (^T)
If you have selected (inverted) a file name in the editor window, you can open this file with "Open Selection".

Close (^U)
Close the active window. If the text has been changed, you will be asked if you want to save it.

Save (^S)
Use this command to save the text in the active window. If your are saving a "New" window you will be asked to name it.

Save As...
This command save the text in the active window, but you will be asked to rename it.

Revert to saved
This command will replace the version of you text in the editor with the latest copy you saved to disk.

Print
Use this command to print the text in the active window to the printer.

Print Selection (^P)
If you have selected (inverted) some text in the editor window, this command will print the selection to the printer.

Execute
Leaves the compiler temporarily, while running another program.

Quit (^Q)
Quits the compiler and close all open windows, if your text in the window(s) has been changed you will be asked if you want to save it/them.

EDIT

Undo ([UNDO])
This command reverts the latest command in the editor.
For example if you have deleted some text "Undo" will bring it back.

Cut (^X)
Cut a selected part of the text to a "clipboard". With the command EDIT/Paste you can insert the text from the clipboard to the editor. The text will be inserted in the editor window (at the cursor position).

Copy (^C)
This command copies a selected part of the text to a "clipboard". With the command EDIT/Paste you can insert the text from the clipboard to the editor. The text will be inserted in the editor window (at the cursor position).

Paste (^V)
With this command you can copy text from the clipboard to the editor window (at the cursor position).

Select All
This command will select (invert) all text in the active editor window.
With this command you can select all text i one window, and with EDIT/Copy copy it to the clipboard, move to another window and with the EDIT/Paste command copy it into this window.

Indent (^K)
With this command you can move a selected part of the text one position to the right.

Outdent (^J)
With this command you can move a selected part of the text one position to the left.

Get info (^I)
This command gives you information about the size of the text in the active window (bytes and number of lines). It also gives you information about the memory size.

User Screen (ESC)
Displays the program output screen.

SEARCH

Find (^F)
With this command you can search for strings in your textfile.

"Find What":
Here you write the string you want to search for.

Search options:
"Words Only":
You can search for "Words Only", which means that the string ("Find What:") you are searching for should be a word for it self, otherwise the search string could be a substring (part of another word).

"Case Sensitive":
You can make this search case sensitive. Which means that the string you are searching for should contain exactly the same upper- and lowercase letter you write in "Find What:".

"From Top":
If you want to start the search from the top of the text, you click here, otherwise click "Ok" the search to start from cursor position.

Find Selection (^F)
With this command you can search for a selected (inverted) string/word from your text.

Find Next
This command will repeat the options given in the "Find" command.

Replace (^A)
This command works like the "Find" command, and furthermore you can replace the string/words you find with another string/word.

"Change To":
Which string/word should the words/strings found be replaced with.

"Replace all without asking":
Shall the replacement be done automatically, or shall the editor ask you everything before it replaces.

Goto line (^L)
This command takes you to the line number you write in the "Goto line:" box. For example "Goto line": 1 will take you to the top of the window (as the "Home" key). The Goto line box default show the actual line.

Find Cursor:
This command will find the place in the text where the cursor is placed.
For example if you have "scrolled" the cursor out off the visible screen, "Find cursor" will position the screen , where the cursor is placed, back.

Cycle Windows:
This command toggles between the open windows ("jumps from window to window").

User screen:
Once you're back in the HIDE after executing the program, you can view the programs output by choosing the SEARCH/User screen (Esc) command. By pressing any key you return to HIDE.

COMPILE

Ω Means hold down the "Alt" button and activate "X".

Run (ΩR)

This command invokes the compiler (with the options given in OPTIONS/Compiler and OPTIONS/linker) on the file in the active editor window (unless Primary file option active). Then it runs your program with the options given in the OPTIONS/Run. If an error occurs, you are automatically placed in the editor window at the error.

Compile (ΩC)

Compiles the file in the active window (unless Primary file option active) , after compilation you can open the "Get Info" window (COMPILE/Get Info) to display compilation information: Code file name, Code size (code/data), whether the program compiled successfully, and available memory.
If an error occurs, you are automatically placed in the editor window at the error.

Make (ΩM)

This command invokes the compilers built-in make.
If a primary file has been named (COMPILE/Primary File) that file is compiled. Otherwise the file in the active window is compiled.
The compiler checks all files upon which the file being compiled depends. If any has been changed they are recompiled.

Build All (ΩB)

This command works like "Make" - except "Build" recompiles all depending files, whether they are changed or not.

Find Error (ΩE)

Finds the location of a run-time error.
When a run-time error occurs under TOS, the offset address in memory where it occurred is given. When you return to HIDE, HighSpeed locates the error for you, using this command. If run-time errors occurs when running within HIDE, the default value for the error address are set automatically.

Destination

Use this option to specify whether the executable code will be stored on: "Disk" (as an .PRG file) or in "Memory" (and therefore lost when you exit the compiler).

Note that any unit recompiled during a "Make" or "Build All" have their .UNI files updated on disk.

If "Destination" is set to Disk, then an .PRG file is created. The .PRG file is placed in the same folder as the source file, or at the place specified in OPTION/Compiler/ Program.

Primary File:

This option specify which .PAS file will be compiled when you use COMPILE/ Make, COMPILE/Build All or COMPILE/Compile.

This option is used when working with a program which uses several units or include files. No matter which file you have been editing COMPILE/Make/Build All/Compile will always operate on the file specified in "Primary File".

Get info (ΩI)

This command gives you information about the compiled file the code size (code/ data) and Total/Free memory. And whether the file has been compiled or not.

OPTIONS

Help ("Help")

The compiler provides context sensitive help at the touch of the (Help) key or by choosing the command OPTIONS/Help.
This will open a help window containing help to the function- or the item- or the word in your source code selected. Any help window can contain one or more underscored keywords which you can double-click (or select and press the Help key), this opens the help window containing help to the keyword. Pressing the (Help) key again will return to the main help window.
Pressing the Undo key closes all the open help windows.

General (ΩG)

"Locate Resident Units":
Here you write the path where the compiler should search for the unit library file *.LIB (if any).
The compiler has to kind of unit files : *.UNI and *.LIB. When you compile a unit the resulting object code is put in a *.UNI file. With the program LIBMAKER.PRG you can join several units into a library.

"Tab width":
Sets the tab size in the editor (1-8).

"Auto indent":
If on, the cursor returns to the starting column of the previous line
when you press Return.
If off, the cursor always returns to column 1.

"Autosave Configuration":
If on, all configuration settings will be saved when quitting the compiler.

"Autosave Files":
If on, the file in the editor will be saved automatically before running.

Compiler (ΩK)

These options can also be specified directly in your source code using compiler directives, "$ directives" (See "Compiler Directives").

"Range check" {$R}
If on, the compiler generates code to check that array-/string subscripts are within bounds.
It also checks that assignments to scalar-type variables do not exceed their defined ranges.
If check failure, the program halts with a run-time error.
If off, no such check is done.

"Stack check" {$S}
If on, the compiler generates code to check that stack space is available for local variables, before each call to a procedure or function. If check failure, the program halts with a run-time error.
If off, no such check is done.

User break, Shift-Shift
Whenever a range or stack check is performed in HIDE, the keyboard is also tested. If the user at that moment presses both shift keys simultaneously, then the range or stack check fails with a runtime error 58. Remember that you may have to have an exit procedure installed in your program in order to properly handle these unexpected breaks. Otherwise you may end up with a locked GEM system.

The Shift-Shift system is not active in stand alone programs.

The $R+ and $S+ checks does make your program a bit slower, but you should only use these features when developing your programs.

"IO Check" {$I}
If on, the compiler generates code to check for I/O errors after every I/O call. If check failure, the program halts with a run-time error.
If off, no such check is done.
{Se also the system function IOResult}

"Use 32 bit fixup" {$F}
Allows you to mak big programs (mere forklaring) !!!!!!!!!!

"Keep names for debugger" {$D}
Add symbols to *.PRG files for later debugging.

"Strict string checking" {$V}
If on, the compiler compares the declared type of a "Var" type string parameter with the type of the actual parameter being passed. If check failure, a compiler error occurs.
If off, no such compare is done.

"$Define" {$Define xxxxx}
With defined symbols you can perform compilation with conditional directives in the source code. Define a symbol by typing it's name here, multiple symbols are separated by ";".
If for example the symbol "Test" is defined, the compiler will generate the code for the "Writeln" in the following example:

```
{$IFDEF Test}
  Writeln('This is a test');
{$ENDIF}
```

Search Path:
"Units":
The path where the compiler finds the unit file(s).
Multiple directories are separated by ";".

"Program":
Here the program stores *.PRG and *.UNI files.
If blank, the files are stored in the directory where the source is found.
Note that the path specified here also ought to be specified under "Units" for the compiler to find the compiled units (*.UNI) again.

"Include":
The path where the compiler can find the include file(s) {$I filename}.
Multiple directories are separated by ";".

"Object":
The path where the compiler finds assembly language routines (*.OBJ files).
When the compiler reach a {L filename} directive it looks in the current directory, if not found, it looks in the "Object" directory.
Multiple directories are separated by ";".

Linker (ΩL)
"$F (Use 32 bit fixup)":
If on, this allows you to make big programs.

"$D (Add symbols to the program file)":
If on, this add symbols to *.PRG file for later debugging.

"$M: Stack":
Specifies (in Kbytes) the size of the stack segment.

"$M: Heap minimum":
Specifies (in Kbytes) the minimum required heap size.

"$M: Heap maximum":
Specifies (in Kbytes) the maximum amount of memory to allocate to the heap.
Note, this value must be greater than or equal to the "Heap minimum".

"$M: Free to DOS":
Default 15 Kb, this enable GEM etc. to work.

"Postfix for program":
Default *.PRG, if you want to compile to a Desk ACCessory, write "ACC" here.

Run (ΩJ)

"Make pause after program run":
If on, the program pauses at the last screen from the program. (Se also EDIT/User Screen).

"Memory for program":
Specifies (in Kbyte) how much memory you want to leave for the programs to be executed by the command FILE/Execute.
Note, this memory will not be reclaimed by the compiler.

Moving around

Moving your cursor with the arrow keys.
"Shift" "->" will take you to the end of a line.
"Shift" "<-" will take you to the start of a line.
"Backspace" deletes the character to the left of the cursor.
"Delete" deletes the character to the right of the cursor.
"Shift" "Up Arrow" will move a page up
"Shift" "Down Arrow" will move a page down.
"Home" will move you to the start of the text.
"Shift" "Home" will move you to the end of the text.

HighSpeed versus other Pascal Compilers

HighSpeed is not exactly equal to other compilers. There will always be some differences, some minor and some major.
The American National Standard (ANS) Institute has defined the standard for Pascal. HighSpeed Pascal will be compared against this one, Turbo Pascal and ST Pascal (Personal Pascal). The comparison is not complete as far as the minor details concerns.

HighSpeed Pascal versus ST- and Personal Pascal

The unit named STPascal implements some of the features special to ST Pascal. Here are some of the ST Pascal types named using underscore and some graphics and GEM routines.

The IOResult and the $I- system is implemented differently in ST Pascal.

The $M directive in HighSpeed tells the compiler all about the memory layout.

HighSpeed Pascal has only the standard loop constructions as REPEAT, FOR and WHILE. ST Pascal implements a non standard loop construction.
Please read the file STPASCAL.DOC in the UNIT directory on the HighSpeed disk, on how to compile STPascal code with HighSpeed.

HighSpeed Pascal versus Turbo Pascal for the PC

Notice one important thing: HighSpeed Pascal is running on a 68000 micro processor, while Turbo Pascal is running on a 80x86 processor. Both compilers uses the CPU's default way of saving data in memory.

The 68000 does not allow reads and writes at odd addresses if the element size is not a byte. Therefore all variables using more space than a byte is allocated on even addresses. Furthermore the 68000 does save words (and others) in a non reversed order in memory, where Turbo Pascal uses the '86 reversed mode. Every word written to a typed file using Turbo Pascal, must then be read in HighSpeed Pascal by a normal read, followed by a N:=Swap(N) instruction. (This could also have been done in Turbo Pascal before the writing). Long integers must be swapped by using both SwapWord, Swap and maybe HiWord instructions.

The word alignment does make record structures look different when made with either HighSpeed or Turbo, even if Turbo uses the $A+ option for word alignments. In Turbo Pascal a string[2] only uses 3 bytes, while a string in HighSpeed always uses an even number of bytes, and that makes a string[2] use 4 bytes. All strings of length 1 to 3 can be moved quickly with a single MOVE assembly instruction.

HighSpeed allows BlockRead and BlockWrite on all typed files in additions to untyped files. The parameters given to them is always a byte count.
Reset and Rewrite does not take any block size parameter.

The use of NIL pointers is not checked in Turbo Pascal. The Atari computer helps HighSpeed here, because it stops a program when it tries to use memory at address zero, where a NIL pointer is physically pointing. Therefore HighSpeed Pascal also checks for the use of NIL pointers. Likewise will the use of uninitialized pointers also sometimes stop the program with a bus or an address error.

HighSpeed Pascal versus ANS Pascal
HighSpeed Pascal cannot tell you if any non standard features are used.

The Put and Get system in ANS Pascal is not implemented.

Procedure and function does not take Procedure and Function parameters in HighSpeed Pascal.

The parameter to New and Dispose cannot specify which variant part to use. You may use GetMem and FreeMem instead.

Pack and Unpack is not implemented.

The @ symbol is an operator in HighSpeed, and the same as the ^ in ANS Pascal.

When reading a char variable when eoln is true, HighSpeed returns a Carriage Return character where ANS Pascal would return a space.

Extensions to ANS Pascal
HighSpeed Pascal cannot tell you if any non standard features are used.

HighSpeed implements strings with dynamic length, and routines to operate on them.

HighSpeed Pascal implements the types ShortInt, LongInt, Single and Double.

HighSpeed implements the unit way of making separate programming.

Labels can be both identifiers and the normal digit sequence.

Integer constants can be written in hexadecimal format when preceded with a $.

All character values can be written in character and string constants when preceded with a # or a ^. A decimal or hex value can be written after the #.

The sequence of Label, Const, Var, Type, Procedure and Function declarations can be in any order.

Three new logical operators: xor, shl and shr. The logical operators also work as bitwise operators.

The @ operator works like the Addr function. In ANS Pascal @ and ^is the same.

HighSpeed Pascal implements value and variable typecast.

An ELSE statement is allowed as the default case in a CASE construction. There is no default in ANS Pascal.

A variable formal parameter in a procedure/function can be typeless.

The list of reserved words can be seen in the WordSymbol list in chapter Language Elements, Reserved Words.

Inside HighSpeed

Here is the information needed for hackers and those who need a bit more specific information on the internal workings of HighSpeed Pascal.
All variables mentioned in this chapter are located in the System unit.

Memory Layout
The memory layout of a loaded user program looks like that normally used on the Atari. The sequence of memory blocks are from bottom and up this:

BasePage, 256 bytes header
Code, all your code and the runtime code
Data, all global variables
Stack, room for the stack pointer
Heap, heap memory used from the bottom.

A program on disk has three key values in its header, the code size, the data size, and the BSS size. A HighSpeed generated program always has a data size of zero. The BSS size is the sum of the global variables the wanted stack size and the minimum heap size.
All $M parameters are placed in address $102 from BasePage as 4 longwords, followed by one unused longword that may be used for any purpose.

When loaded, a program tries to increment the BSS area as much as allowed by the amount of free memory, and as allowed by the $M parameters given. The remaining memory is then released to the free memory pool. The second longword in the BasePage is then updated to point to the last location used.

A desk accessory does not release any memory, and does not try to allocate more than stack and heap-minimum says in the $M parameter.
The last two parameters on the $M list is not used.

These variables points into a loaded program

BasePage:
Pointer to the BasePage.
HeapOrg:
Pointer to the bottom of the heap.

HighStak:
Top of Stack. Actually the same as HeapOrg.

LowStack:
Bottom of stack. The stackcheck routine checks the stack pointer against this one.
The LowStack is 126 bytes above the real bottom. It can be incremented a bit if you need to have a larger margin.

Traps and Exceptions
When a normal program (not a desk accessory) starts, it initializes some traps so that an error message can be given if exceptions does occur. The old traps are saved in STrapNN, where NN is the number of the used trap. NN is either 5 or 102 (hex). Number 5 is for Division by zero and number 102 is for Terminate. Bus and Address errors will still give the normal bombs forcing the program to terminate. This is where the 102 trap gets activated.
Right before the program terminates, it calls all ErrorHandler procedures from the ExitProc list.

Exit Procedures
The system unit implements a variable ExitProc: Pointer. It points to a procedure that is called when a program is about to terminate, no matter what the cause is. The ExitProc is initiated to point to the systems exit handler, which removes the installed traps before the final termination. If this handler is not called before the program terminates, you are asking for trouble. The installed traps will point to non existing code after the program has terminated if they are not removed.

The ExitProc pointer enables you to gain control before the program terminates, just by entering your own exit procedure into the chain. Your exit procedure can then close files and do other cleaning jobs.

When the ExitProc handler starts working, it keeps calling the procedure pointer to by ExitProc, until the last procedure (the systems own exit) has been called.

You use the Exit handler system this way:

```
var
   OldExitProc: Pointer;

 Procedure MyExitHandler;
 begin
   ExitProc:=OldExitProc;
   Cleanup...
 end;
```

```
procedure Initialize;
begin
  OldExitProc:=ExitProc;
  ExitProc:=@MyExitHAndler;
end;
```

The exit handler should restore the ExitProc variable as the first thing. Any errors that occur before the ExitProc is restored will terminate the program right away, skipping the execution of thesystems exit handler.

When the exit handler starts it work, it resets the stack pointer to HighStak. Otherwise a stack overflow might occur.

Heap Manager

The heap is where dynamic data is kept. Data blocks are allocated with New and GetMem, and disposed of again using Dispose and FreeMem. Every block released must be released using the same parameters for the release as for the allocation. If more memory is released than retrieved the heap system will be destroyed.

Heap memory is allocated in blocks of 8 byte at a time. Allocating 1000 integers on the heap will use 1000*8 bytes of the heap, the same space as for 1000 double reals.

When memory blocks are released in the middle of other blocks, there will be a hole, that can later be reused. If there are no holes in the heap, the result of MemAvail will equal the result of MaxAvail.

The free list is kept in the first 8 bytes of a released block. Initially there is one big free hole.

Data formats

Integer Types

Values are saved using as much space as needed to save all the information including a sign bit.

Integers, -32768..32767, are saved in a 16 bit word.

LongInt's, -2147483648..2147483647, are saved in a 32 bit (long) word.

ShortInt's, -128..127, are saved in an 8 bit word.

Subranges of integers are saved using signed values. Subranges that lays between the bounds of ShortInt, Integer or LongInt are stored using one of these formats. A subrange of 0..255 is therefore stored in a 16 bit word. Packed structures does save a subrange of 0..255 as an unsigned 8 bit byte.

Char Types

The predefined type Char, #0..#255 are stored using a 16 bit word, with a zero in the high byte, and the ASCII value of the character in the low byte.

A subrange in the area from #0..#127 is stored in an 8 bit word.

Packed structures does save a Char in an unsigned 8 bit byte.

Enumerated Types

An enumerated type is stored as a byte if it contains less than 129 values, else it is saved using a 16 bit word.

Boolean Types

The Boolean type is an enumerated type of False..True and as such saved in a byte.

Real Types

The Single and Double types are saved in 4 and 8 bytes using the IEEE standard format as it can be found in a 68881 manual.
An extended real is saved in 10 bytes using an internal format. This format is not compatible with the 68881 Floating Point Unit.
This is what you get from the reals:

Type	Bytes used	Range	Useful digits
Single	4	3.4e-38..3.4e38	7-8
Real=Double	8	1.7e-308..1.7e308	15-16
Extended	10	1.1e-4932..1.1e4932	18-19

The trigonometric functions all works using Double precision.

Do not use Reals for counting. In integer types, the LOWER xx bits are saved, always giving you access to add one in order to change the value. The real types only saves the UPPER xx bits of the value, skipping the lesser significant bits.

Calculations on real types does never makes runtime errors. Instead the result returned contains a special mask. By using the function ValidReal, you can test if a returned value is valid.The function returns false if the real value is illegal.

```
Function ValidReal(R: Real): Boolean;
```

The predefined constant NAN contains the special mask used for signaling bad reals.

When writing a value containing a NAN, the string ' NAN' or '-NAN' is written.

Pointer Types

A Pointer is saved using a 32 bit longword. NIL is saved as a longword of zero.

The Ptr function takes a LongInt and type converts it into a pointer. The 32 bit value returned is the same 32 bits as in the 32 bit LongInt.

Set Types

Set types are stored using 1 to 32 bytes. Each byte can hold eight elements. The lower and upper limits of the set are byte aligned before calculating the number of bytes needed.

A set of 6..10 uses 2 bytes because the limits are aligned to the range of 0..15.
The bit number within a byte is calculated as: Number = ElementNumber mod 8.

Array Types

Unpacked elements are all stored on even addresses, forcing byte elements to be interleaved with filler bytes. A packed array does pack Char and Integer subranges of 0..255 into unsigned byte elements.

The array elements are stored sequential in memory (maybe with padding) with the element with the lowest index first in memory. A multi dimensional array is saved with the right most dimension increasing first.

Example:

```
Var A3: array[1..3,1..10,1..20] of integer.
The array A3[1,3] is an array of [1..20] of integer.
The array A3[2] is an array of [1..10,1..20] of integer.
```

Record Types

The first element of the Record is stored first in memory followed by the others in same sequence as written. All variables using more than one byte are word aligned. All variant parts starts at the same address.

As with arrays, packed records does pack Char and Integer subranges of 0..255 into unsigned byte elements.

Files types

Files types uses 18 bytes of memory. This is the memory layout of the file control block:

```
Type
 FileRec = Record
  fInpFlag:   Boolean;     {Uses for input}
  fOutFlag:   Boolean;     {Used for output}
  fHandle:    Integer;     {File handle}
  fBufSize:   Integer;     {Record size or Text: buffer size}
  fBufPos:    Integer;     {Text: buffer position}
  fBufEnd:    Integer;     {Text: buffer last used}
  fBuffer:    Char32KPtr;  {Text: buffer pointer}
  fInOutProc: Pointer;     {Text: IO handler if fHandle is zero}
 end;
```

Text files uses additional space in the heap for its buffers. fBuffer points to this memory. The size of this buffer can be set in the Reset, Rewrite and Append call.

fHandle contains the file handle obtained when opening the file. If zero the fInOutProc points to a device handler.

The filename is not saved in the File control block.

Calling Conventions

Parameters for procedure (and function) calls are moved to the stack using the normal Pascal order of parameters. The first parameter met in the source is first moved onto the stack. This is opposite to the C-language way of doing things. The called routine removes all parameters from the stack, except for the result from a functions.

A procedure call looks like this:

```
MOVE    Param1,-(SP)
MOVE    Param2,-(SP)
MOVE    Param3,-(SP)
JSR     Procedure
```

A function call looks like this:

```
SUB     #nn,SP
MOVE    Param1,-(SP)
MOVE    Param2,-(SP)
MOVE    Param3,-(SP)
JSR     Function
MOVE    (SP)+,Result
```

The function call makes room for the result on the stack first, and removes the result itself after the call.

If the result uses more than 4 bytes, then only the address is passed.

This is the sequence used:

```
PEA     Result
MOVE    Param1,-(SP)
MOVE    Param2,-(SP)
MOVE    Param3,-(SP)
JSR     Function
ADD     #4,SP
```

The procedure and function code looks like this:

```
JSR      StackCheck
LINK     A6,#-StackUsed
MOVEM.L  D3-D7/A2-A5,-(SP)
{Take copy of value parameters}
....
MOVEM.L  (SP)+,D3-D7/A2-A5
UNLK     A6
MOVE.L   (SP)+,A0
ADD      #ParamSize,SP
JMP      (A0)
```

The last 3 lines may look like a single "rts" if the routine takes no parameters. The "movem" is removed if none of the registers d3-d7 or a2-a5 are used by the routine. The StackCheck routine is called if the compiler directive $S+ is active. It looks at the StackUsed parameter, and compares this value added with the stackpointer SP against the system variable StackLow.

If the called procedure is nested, that is if it is declared inside another procedure, the stack frame is also passed as a parameter, the last one. All parameters then moves four bytes up in memory from the A6 register.

A procedure call to a nested procedure looks like this:

```
MOVE    Param1,-(SP)
MOVE.L  A6,-(SP)
JSR     Procedure
```

The procedure can then use code as:

```
MOVE.L  4(A6),A0
MOVE    nn,(A0)
```

in order to read or write variables in other levels. If the nesting level is more than one, a number of "MOVE.L 4(A0),A0" can be inserted between these two lines.

Value Parameters

Value parameters are copied every time the routine gets called. If the size of the variable is less or equal to 4 bytes, its value is pushed onto the stack right away.

When the size is greater than 4 bytes a reference is pushed. A normal Pascal routine then makes it own copy as the first thing in the routine. An assembler routine does not have to take a copy if it does not change the parameter, thereby saving time and code.

Two and four byte parameters are passed using the same storage method as used when storing in normal variables.

Byte values are passed using the normal procedures for bytes on the stack, using a word. Otherwise the stack pointer would be misaligned.

Real types are passed by reference to an extended real, maybe first converted from the other real formats.

Arrays and Records are passed by reference unless they can becontained in four bytes.

Strings are passed by reference.

Sets are passed as references to a maximized set of 256 elements.

Variable Parameters

Variable parameters are always passed by pushing the address of the variable. The routine then uses the exact same location for its formal parameter as the caller does for its actual variable. Both variables must of cause be of exact same type.

Function Results

For function results of Integer, Char, Boolean and any subrange, two or four bytes are allocated on the stack before the call. The result is returned in this space and removed after the call.

For Real types, the caller allocates 10 bytes for an extended real, and pushes the address of this before any parameters. The caller removes this again after the call.

For String types, the caller allocates temporary space for the string, pushes the address of this before any parameters and removes it again after the call.

Other types cannot be returned.

Using Assembly Language

The {$L FileName} compiler directive allows assembly code to be included into your Pascal code. The object format used is the standard one used on the Atari. Thesefiles have the extension ".O" as default. When "Making" a program (compile with update check), the $L file is also checked to see if it has been changed.

HighSpeed Pascal makes some restrictions on what can be done using object files due to the nature of the built in smart linker in the compiler and because the object file first gets included in the internal unit format.

Restrictions:

1. There is no declared data area in HighSpeed Pascal programs. Data areas can therefore not be used. Only the BSS area can be usedfor data.

2. All references to procedures (and functions) in other assembly modules or in the Pascal source must use the JSR, JMP, LEA or PEA formats. BSR and Bcc are only legal within a module, because the assembler fixes these references itself. Also each procedure used for cross module reference must be named in the Pascal source by an external declaration, simply to tell the compiler that another entry in the procedure table is needed.

3. 8 bit fixups are not allowed as in Var1(A0,D0.W).

4. The registers that may be used are D0, D1, D2, A0 and A1. All other used, must be saved and restored by the routine.

5. All names are restricted to 8 characters. All interfaced names cannot be longer than 8 characters. This is why the System variables HighStak and LowStack are named this way. The KeyPressed routineis an assembly file. This one is made by making a normal Pascal routinethat calls the real assembly file KeyPresX.

Below are two ways of implementing a function AddThing declared as shown in following program:

```
Program test;
{$L MyAsm}
Function AddThing(Thing1: Integer; VAR Thing2: Integer): Integer;
EXTERNAL;

Var n,SideVar: Integer;
begin
  n:=AddThing(7,n);
end.
```

This is the standard way of making AddThing using an assembler

```
          EXPORT  AddThing          ;Make AddThing public
AddThing:LINK    A6,#-0             ;= because no locals
         MOVE.L  8(A6),A0           ;Read ^Thing2
         MOVE.W  8+4(A6),D0         ;Get Thing1
         ADD.W   D0,(A0)            ;Update Thing2
         MOVE.W  D0,8+4+2(A6)       ;Return D0 to AddThing
         UNLK    A6
         MOVE.L  (SP)+,A0           ;Get return address
         ADD     #4+2,SP            ;Remove Thing2 and Thing1
         JMP     (A0)               ;and return
```

This is the quick and dirty way of making assembly files, also showing the use of a global Pascal variable SideVar.

```
          EXPORT  AddThing              ;In file MyAsm.S
AddThing:MOVE.L   (SP)+,RetAddr
         MOVE.L   (SP)+,A0              ;Read ^Thing2
         MOVE.W   (SP)+,D0              ;Get Thing1
         ADDQ.W   #1,SideVar            ;Side effect
         ADD.W    D0,(A0)               ;Update Thing2
         MOVE.W   D0,(SP)               ;Return D0 to AddThing
         MOVE.L   RetAddr,A0            ;Get return address
         JMP      (A0)                  ;and return

         BSS
         IMPORT   SideVar               ;Global data
RetAddr: DS.L     1                     ;Local data
```

Inline

Small assembly code subroutines can be placed directly in the source by declaring an inline procedure (or function). They should be short because all the code is inserted every time the procedure or function is called.

The syntax is:

```
InlineDirective =
"Inline" Constant { "," Constant }.
```

Each constant is a 16 bit value.

Following is an implementation of function AddThing using an inline directive:

```
Program test;
Function AddThing(Thing1: Integer; VAR Thing2: Integer): Integer;
  INLINE $205F, { MOVE.L  (SP)+,A0 ;Read ^Thing2}
         $301F, { MOVE.W  (SP)+,D0 ;Get Thing1}
         $D150, { ADD.W   D0,(A0)  ;Update Thing2}
         $3E80; { MOVE.W  D0,(SP)  ;Return D0}
var m,n: Integer;
begin
  n:=AddThing(7,m);
end.
```

This is the input to Borlands Macro Assembler MAS-68K:

```
MOVE.L   (SP)+,A0          ;Read ^Thing2
MOVE.W   (SP)+,D0          ;Get Thing1
ADD.W     D0,(A0)          ;Update Thing2
MOVE.W    D0,(SP)          ;Return D0 to AddThing
END
```

and this is the output from MAS-68K, used in making the inline procedure heading:

```
MAS-68K 68030 Macro Assembler (C) 88 - 90 SoftDesign Muenchen
     Source File: ADDTHING.S
     1 00000000' 205F  MOVE.L  (SP)+,A0  ;Read ^Thing2
     2 00000002' 301F  MOVE.W  (SP)+,D0  ;Get Thing1
     3 00000004' D150  ADD.W   D0,(A0)   ;Update Thing2
     4 00000006' 3E80  MOVE.W  D0,(SP)   ;Return D0 to AddThing
```

Device Driver

It is possible to define your own devices in HighSpeed Pascal. They are activated by Reset, Rewrite and Append calls when referenced by their logical name.

A device is installed with a call to:

```
Device(Name: String; IOprocedure: Pointer).
```

The call Device('MyDriver',@MyDriver) installs the name along with the address of the IO handler.

The IOprocedure must have a heading like this:

```
Function IOproc(var F: FileRec): Integer;
```

By looking at the flags the routine can see why it is called. The function value returned is put into the IOResult variable, forcing a runtime error if $I+ is used.

The routine is called by Write, Writeln, Page, Read, Readln, Close, Eof, SeekEof, Eoln and SeekEoln whenever input or output must be performed.

fInpFlag is set when the driver should read fBufSize bytes (characters) into the buffer fBuffer^. fBufPos is then set to zero, and fBufEnd is set to show how many bytes actually got read. Whenever zero byte is read, Eof is true.

fOutFlag is set when the driver should write fBufPos bytes from the buffer fBuffer^. fBufPos is set to zero if the data is written. It is legal not to do anything if the buffer is not full, that is if fBufPos does not equal fBufEnd.

A skeleton driver:

```
Function MyDriver(var F: FileRec): Integer;
var
  P:  Integer;
  Ch: char;
begin
  MyDriver:=0; {IOResult:=Ok}
  With F do begin
    if fInpFlag then begin
      fBufEnd:=0;
      repeat
        Ch:=ReadKey;
        fBuffer^[fBufEnd]:=Ch; Inc(fBufEnd);
      until (Ch=#10) or (Ch=^Z) or (fBufEnd=fBufSize);
    end else {Doing output}
      begin
        for P:=0 to fBufPos-1 do WriteCh(fBuffer^[P]);
        fBufPos:=0;
      end;
  end;
end;
```

Compiler Errors

These are the "expected" errors of the simple type:

```
';'  expected
':'  expected
','  expected
'('  expected
')'  expected
'['  expected
']'  expected
'.'  expected
'='  expected
':=' expected
'..' expected
END  expected
DO   expected
OF   expected
THEN expected
TO or DOWNTO expected
```

BEGIN expected
Often returned with the cursor on top of garbage text. The compiler expects BEGIN when it cannot find any further declarations in a function.

INTERFACE expected
The word INTERFACE must follow a UNIT heading.

IMPLEMENTATION expected
When in a unit, the compiler expects IMPLEMENTATION when it cannot find any further declarations.

The following are also errors of the "expected" type, but more complex than the previous.

Constant expected
Boolean expression expected
The construct "IF expression THEN" needs a boolean expression. You cannot write IF 2+3 THEN.

Integer constant expected
The real assignment R:=123.4; is ok, but the assignment R:=12345678; is bad because the compiler reads the number as an integer. You can add a '.0' to make it a real.

Integer expression expected
Integer variable expected
Integer or real constant expected
Integer or real expression expected
Integer or real variable expected
Pointer variable expected
Typed Pointer variable expected
File variable expected
Record variable expected
Ordinal type expected
Ordinal expression expected
String constant expected
String expression expected
String variable expected
Identifier expected
Variable expected
Type identifier expected
Field identifier expected
Char expression expected

The rest of the errors are of more varying kinds:

Unknown identifier
Undefined type in pointer definition

'.' expected after module name

Duplicate identifier
Note that the program name is also entered into the symbol table.
This can be used to reference items in other units if redefined by other units. If you have made a Reset procedure, then you can only reset a file by writing System.Reset(aFile,").

Type mismatch
Constant out of range
Constant and CASE types do not match
Operand types doe not match operator
Invalid result type
A function can only return simple, string and real types.

Invalid string length
Invalid subrange base type
Lower bound greater than upper bound

Invalid FOR control variable
The FOR variable must either be declared in the same function or globally. It may not be part of any structure.

Illegal assignment
String constant exceeds line
You may have forgotten an apostrophe in the string. Note that a single apostrophe in a string must be written as two apostrophe's.

Error in integer constant
Error in real constant
Division by zero
Structure too large
Constants are not allowed here
Invalid type cast argument
Invalid '@' argument
Only works on functions, procedures and variables.

Invalid GOTO
Label not within current block
Undefined label
Label already defined

Invalid file type
Cannot read or write variables of this type . Files must be VAR parameters
File components may not be files

Set base type out of range

Error in type
Error in statement
Error in expression

Undefined FORWARD procedure: xx
Undefined EXTERNAL procedure: xx
Invalid external definition, symbol = xx
Invalid external reference, symbol = xx

Too many symbols
Only 32 KByte is set aside for the symbol table for each unit or program.

Too many nested scopes
Also given when too many units are used. Maximum is 63.

Expression too complicated
All data registers in the 68000 processor are used for temporary integers.

Too many variables
Too much code
Split the code into more units.

Bad unit or program name: xx
Unit missing: xx
Unit not found
The linker cannot find a specific unit in the PACSAL.LIB file. Nor can it find the unit in any of the directories specified in the OPTIONS/Compiler dialog.

Incompatible unit versions
The program is trying to use a unit compiled with another version of the compiler than you are currently usoing

Syntax error
Something is wrong!

Unexpected end of text
You may have more begins than ends. Maybe caused by an non-terminated comment.

Line too long
Only 127 characters are allowed on one line.

Invalid compiler directive
Bad object file in file: xx
You cannot make BSR and BRA to externally defined labels. You MUST use JSR and JMP. See the restrictions in "Inside HighSpeed Pascal".

Not enough memory
Too many files ($I or USES)
Each Uses file uses an include level when it is re-compiled.

No room in 16 bit fixup field, try with $F+
An attempt is made to call a fragment of code located more than 32 Kbytes away from the current location in memory.

Duplicate unit name: xx
Conditional variable missing
Misplaced conditional directive
ENDIF directive missing

Runtime Errors

Tos errors are returned as negative numbers. The corresponding positive error number can be found by using the formula NewNo:=31-OldNo.

Runtime errors are generated at runtime and causes the program to terminate

50 Stack overflow.
Generated by the $S+ control code. Check is performed at entry to each procedure and function using this feature.

Fix: Your program has too little stack space, or it has entered a recursive procedure that did not terminate before the stack area was used up. Your procedures may also have large structures (arrays) as local variables, which are located on the stack while the procedure is running.

51 Range check.
Generated by the additional $R+ code.

Fix: You may be trying to assign an invalid value to a variable, either by using an assignment as := or passing it to a procedure/ function.
Another bug could be to try to access a non-existing element of an array (string).

52 BUS error.
Generated by the 68000 CPU, when trying to access invalid memory. The memory right above address zero is also invalid for a normal program. This area can only be accessed in supervisor mode by first using the Super function from the Tos unit.

Fix: The problem could be that you (inadvertently) are using a pointer variable containing NIL (or garbage).

53 ADDR error.
Generated by the 68000 CPU, when trying to access word or longword elements from an odd address. This cannot be done in one instruction, you have to read it one byte at a time.

Fix: You will only see this problem if you make the addresses yourself, by making your own pointer variables with the Ptr function. HighSpeed Pascal always locates its word variables on even addresses.

54 Other errors.
BUS, ADDR and DIV0 errors are hardware traps in the 68000 system. Many other traps are defined. If the program terminates due to a trap other than above three, the error is shown. You can count the number of bombs on the screen to see which trap (if any) was used.

55 DIV0.
Division by zero. Whenever the 68000 CPU makes a division by zero, the error is given. Long integers are divided by internal routines, that also gives error 55 when you try to divide by zero.

56 Heap overflow.
When the heap is so full that a request for a new block cannot be satisfied, the error is given.

Fix: You must test the MaxAvail function instead of the MemAvail function. MaxAvail returns the size of the largest block where MemAvail returns the total.

The following are the errors generated by the file system.

60 File not open.
You may not read from a file not reset.

61 File not open for read.
You can write to a text file in rewrite mode, but not read from it.

62 File not open for write.
You can read from a text file in reset mode, but not write to it. Use append first.

63 No memory for device.
A HighSpeed Pascal device driver needs heap memory for its buffer and name.

64 Read Write error.
A read/write error has occurred.

65 Bad numeric value read.
Non-numeric data has been read into a numeric variable.

Books Pascal/GEM

Title , Author , Description, Publisher, Pages, (x=nothing).

Pascal and GEM books in English:
Oh! Pascal!, Doug Cooper and , Pascal tutorial , WW Norton & Co.

The Atari ST Explored, Braga , Programmer's Technical Guide, Kuma

Computes! Tech. ref. Guide Vol. 2, Sheldon/Lemmon, Guide to GEM AES, Compute!

ST GEM Programmer's Ref., Szcezepanowski/Gunther Abacus.

Programming the 68000 , Williams Sybex.

Pascal and GEM books in German:
Atari ST - Arbeiten mit GEM, Vol. 1, Sender, x, Sybex, pgs. 320

Atari ST - Arbeiten mit GEM, Vol. 2, Danielsson/Volkmann, x, Sybex, pgs. 240

Atari ST GEM, Abraham El Al, Programming GEM with Pascal, C, and Assembler, Dala Becker, pgs. 691

Atari St Programmierpraxis ST Pascal, Wollschlaeger, x, Markt & Technik, pgs. 288

GEM praxis auf dem Atari ST, Tolksdorf, x, Heise, pgs. 167

GEM Programmier Handbuch, Balma/Fitler, x, Sybex, pgs. 520

Softwareentwicklung auf dem Atari ST, Gei·/Gei·, Programming GEM with Pascal and C, Häthig, pgs. 424

Vom Anfänger zum GEM Profi, GEM programming on the ST and PC, Gei·/Gei·, x, Häthig, pgs. 300

Index

Symbols

A

B

C

D

E

F

G

H

I

J

K

L

M

N

O

P

Q

R

S

T

U

V

V